Got Predictability? Experiences with Fault-Tolerant Middleware

Tudor Dumitraş and Priya Narasimhan

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

Abstract. Unpredictability in COTS-based systems often manifests as occasional instances of uncontrollably-high response times. A particular category of COTS systems, fault-tolerant (FT) middleware, is used in critical enterprise and embedded applications where predictability is of paramount importance. Our prior empirical study, which used a client-server microbenchmark, suggested that hard bounds for the maximum latency are hard to establish *a priori*, but that the unpredictability may be confined to less than 1% of the requests. In this paper, we present empirical data, from 7 different three-tier, FT-middleware applications, that shows strong evidence supporting this "*magical 1%*" hypothesis. We conducted a controlled experiment with 7 teams of students from a graduate-level course at Carnegie Mellon University. Each team, starting from a common three-tier architecture, independently implemented and evaluated an original application using middleware (either CORBA or EJB) and a custom-implemented fault-tolerance mechanism (relying on either state-machine or primary-backup replication) for the middle-tier server. This experiment shows that unpredictability may not be avoidable, even in the absence of faults, and that, in some cases, the random latency outliers are larger than the time needed to recover from a fault. The data also reveals a statistically-significant result that, across all 7 applications, unpredictability is confined to the highest 1% of the recorded end-to-end latencies and is not correlated with the request rate, the size of messages exchanged or the number of clients. This suggests that strict predictability is hard to achieve in FT-middleware systems and that developers of critical FT applications should focus on guaranteeing bounds for statistical measures, such as the 99^{th} percentile of the latency.

1 Introduction

Fault-tolerant (FT) middleware incorporates a collection of mechanisms, which usually rely on commercial off-the-shelf (COTS) operating systems, middleware libraries and replication toolkits, for fortifying distributed applications against failures and outages. In addition to establishing a common programming abstraction across different platforms, FT middleware aims to provide transparency with respect to failures, recovery and replication [1]. Typically used in the most critical enterprise and embedded systems, FT middleware has higher predictability requirements than most technologies for platform interoperability. We naturally

expect that faults, which are inherently unpredictable, will have a disruptive effect on the performance of the system. However, FT middleware is usually assumed to behave in a predictable manner in the fault-free case.

In order to enforce the timeliness or other quality-of-service goals of a distributed application, the response times of the distributed application must be bounded and predictable for a given configuration of the system. However, recent studies have independently reported that the maximum end-to-end latencies of both CORBA and Fault Tolerant CORBA (FT-CORBA) middleware can be several orders of magnitude larger than the average latencies and might not follow a visible trend – even in the absence of faults [2, 3]. This problem has been observed in many systems, especially when combining several third-party COTS components [4].

In our previous work [5], we sought to isolate this uncontrollable behavior and to determine its root cause by studying the behavior of a client-server micro-benchmark that uses the middleware [6]. While we have determined that unpredictability comes from multiple sources and cannot be easily eliminated, we show that it is limited to a very small number of remote invocations, and we suggest a simple rule of thumb called *"the magical 1%": After discarding the highest 1% of the measured latencies, the maximum latency can be correlated with the configuration parameters of the middleware and follows observable and predictable trends*. This suggests that unpredictability is confined to a few (less than 1%), very large outliers and that we can easily establish bounds for the 99^{th} percentile of the latency. This simple statistical technique enables us to guarantee, with a high 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. The notion of a percentile-based approach is not far-fetched and is commonly used for specifying QoS guarantees in the IT industry [7].

In this paper, we investigate whether the magical 1% occurs in three-tiered enterprise systems as well, and we provide more conclusive evidence of the magical 1% hypothesis. We compare the behavior of seven realistic, three-tier applications developed during a semester-long (15-week) graduate course at Carnegie Mellon University. The applications were developed and tested independently by different teams. We collect data from experiments conducted in a local-area network setting to emphasize that unpredictability in FT middleware occurs even without the (expected) asynchrony of wide-area networks. We discover that, despite differences in the middleware (CORBA or EJB), replication mechanism (state-machine or primary-backup), message sizes, request rates or database-access patterns, all applications except one have unpredictable maximum-latencies that can be up to three orders of magnitude higher than the average latencies. For all seven applications, we also find a statistically-significant result that the 99^{th} percentile latency cannot exceed the average latency by a factor of more than 20, which allows us to establish upper bounds based on the middleware's configuration parameters. In addition, fault-injection experiments suggest that, for stateless middle-tier servers, the fault-induced high latencies

may be comparable to the maximum latencies from the fault-free case. This result suggests that strict predictability is hard to achieve in COTS-based, FT-middleware and that developers of fault-tolerant applications should focus on statistical measures such as the 99^{th} percentile of the latency.

2 Problem Background

The Fault Tolerant CORBA (FT-CORBA) standard [8] specifies ten parameters that can be tuned to achieve the required levels of performance and fault-tolerance for every application object. However, the standard remains silent on how these parameters should be set or re-tuned over the application's lifetime [1]. Even for a static configuration with fixed values of these parameters, the end-to-end latencies are hard to bound because they exhibit skewed and sometimes bimodal distributions [3]. 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 average latencies [2]. Thaker [4] reports that, even during fault-free experiments, many systems produce a few numbers of outliers several orders of magnitude larger than the average values. This result has been reproducibly borne out across a plethora of 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), and even our own own MEAD fault-tolerant middleware system.

In our effort to understand the sources of unpredictable behavior in fault-tolerant middleware, we have previously reported a phenomenon called "the magical 1%" [5]. This study was performed with a two-tier client-server micro-benchmark of our own design and implementation. While analyzing the behavior of the system in the fault-free case, we discovered strong empirical evidence that the system's unpredictability arose from merely 1% of the remote invocations. Our study showed that the occurrence of very high latencies could not be regulated through parameters such as the number of clients, the replication style and degree or the request rates and could not be isolated to any single component of the system. However, once we filtered out a selective "magical 1%" of the raw latency measurements, the end-to-end latency became bounded and predictable.

We have since sought to understand whether the magical 1% effect was a universal phenomenon for a larger class of fault-tolerant middleware applications. Typical three-tier, enterprise systems have a front-end, a middle-tier implementing the business logic and a back-end database. These applications incur latencies on the order of seconds and tens of seconds, due to the back-end database and the computationally-intensive middle-tier. If the response times of the database and the middle tier are usually predictable, the overall system will also be more predictable because these two components dominate the end-to-end latency. In this paper, we examine whether the "magical 1%" phenomenon extends to three-tier, enterprise middleware-based systems.

3 Experimental Method

In the Spring 2006 (January-May 2006) semester, 35 students enrolled in the "Fault-Tolerant Distributed Systems" graduate class at Carnegie Mellon University were asked to design and implement, from scratch, a realistic three-tier enterprise system and to collect data to analyze the system's behavior. The goal of this course is to teach students the use of state-of-the-art practices in middleware, fault-tolerance patterns (such as replication, transactions, high performance optimizations) and software engineering, with the ultimate aim of teaching them how to develop reliable distributed systems.

The students formed 7 teams, each team focusing on a different application of their own choice and design. The projects varied in size (6-12 kSLOC), in scope and in the application domain (online game, e-commerce) that each respectively targeted. No effort was made to influence the students to favor one kind of application over another. To eliminate bias, the students were deliberately informed that their objective was not to prove or disprove the magical 1% effect for their respective system, but that they should simply report their observations as-is.

3.1 Common Architectural Considerations

To provide some common architectural ground across all the applications, the teams were required to adhere to a three-tier architecture, to use a middleware platform (either CORBA or EJB), and to render the middle-tier fault-tolerant through the use of replication (either state-machine or primary-backup replication). In addition, the empirical practices for measurement, data collection and statistical analysis were common across all of the teams, e.g., all teams measured end-to-end latency in a similar way. Despite this enforced commonality, the teams had significant freedom in their design choices for replication style, middleware platform, programming language, etc. Even the mechanisms for ensuring consistent replication were left to the students to design and implement.

To expose the students to state-of-the-art practices, teams were required to model their replication infrastructure after a commercially available fault-tolerance standard, namely the Fault-Tolerant CORBA standard [8]. As shown in Figure 1, the clients connect to a server (middle tier), which performs all the business-logic processing and uses a database in the backend to store all of the critical state. Effectively, the middle-tier servers are stateless, which is advantageous for checkpointing and recovery. The clients and servers communicate using CORBA or EJB middleware. A Replication Manager controls the mechanisms used for replicating the middle-tier servers: It creates the servers, registers them with either the CORBA Naming Service or the Java Naming and Directory Interface (JNDI), maintains a list of available replicas, provides a reference to a functioning replica for failover after a fault and re-launches the crashed replicas. A Fault Detector monitors the heartbeats of all the server replicas and notifies the Replication Manager when a fault occurs. The clients use the Replication Manager and the Naming Service for bootstrapping to obtain an initial reference to a server.

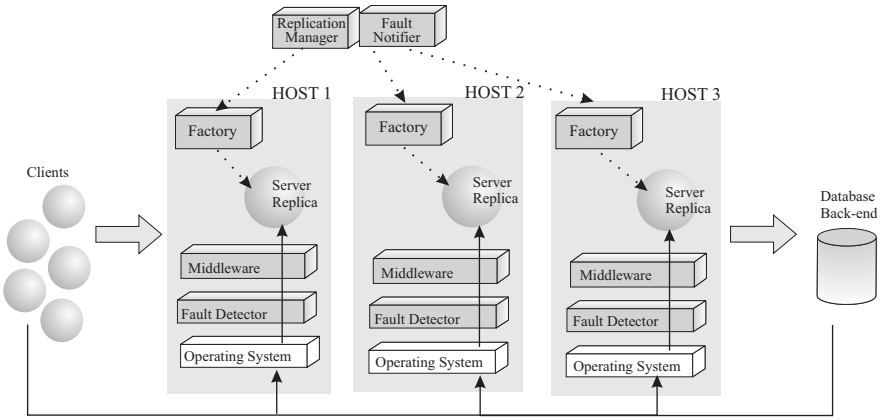


Fig. 1. Common architectural theme for the 7 developed applications.

Warm-passive (primary-backup) replication [9] is most commonly used in enterprise middleware systems. The clients become aware of a replica crash after receiving an exception or a fault-notification, the latter from the Replication Manager. Then, they query the Replication Manager for a new server replica and re-send their request. Alternatively, the systems could use the active (state-machine) replication style [10]. In this case, the client sends the request to all the server replicas, and receives replies from all of them. In case of a replica crash, the client continues to work with the subset of replicas that are still running. The trade-off between these approaches is that active replication has shorter fault-recovery times, while warm passive replication uses less resources (*e.g.*, bandwidth, CPU time) [11]. Because the middle-tier servers are effectively stateless, the issue of state restoration does not arise as a trade-off between state-machine and primary-backup replication.

Sufficient knowledge and guidelines were provided to the students to enable them to implement consistent replication. For example, students exercised care in ensuring that duplicate operations were not processed by replicas. The replication of a server naturally gives rise to duplicate messages entering and leaving the replicas. Duplicate messages by themselves do not affect consistency, it is the duplicate processing of them at a client or a server replica that can threaten consistent replication. Consider a request that increments a value in the database by a fixed amount. If this request is processed by two different server replicas – *e.g.*, by both the new and the old primary replicas after a failover in warm-passive replication or in active replication – the final result will be incorrect. To address this issue, all requests are uniquely numbered and, before processing an invocation, each server replica verifies whether the result of the current request is already stored in the database. When there are multiple clients, yet another identifier uniquely representing each client is embedded into each request to distinguish different clients may legitimately try to invoke the same server method.

To allow the students to remain focused on the fault-tolerance of the middle-tier server, the students were allowed to make a number of assumptions. For instance, they could assume that the Replication Manager, the Naming Service and the database would never fail. Students were encouraged not to cache data in the middle tier because this would violate the statelessness assumption of the middle-tier servers, thereby introducing state-management problems.

To ensure that all of the projects were evaluated in the same environment, all experiments were performed so that each entity in the system (clients, servers, database) ran on a different machine in a cluster connected by a local-area network (LAN). The machines used for the experiments were dual-processor Pentium 4s at 2.8 GHz with 2GB memory, running SUSE Linux (kernel 2.6).

The students defined their projects by specifying the requirements of their applications, choosing the appropriate middleware for their system (CORBA or EJB) as well as the replication style. They were also free to choose between the Java and C++ programming languages for implementing their systems; however, all the 7 teams in this case elected to use Java. The students were not informed about the data-analysis requirements until after the systems were fully implemented. This helped us to eliminate any bias or incentive to report artificial data, rather than the observed output of the applications.

Differences from our Prior Work [5]. The results we previously obtained arose from a micro-benchmark that did not employ a database in the back-end. The servers were stateful, and used both active and passive replication. In contrast, the study presented in this paper uses stateless middle tiers that store their persistent state in a database. In our previous work, we tested our own MEAD middleware [6], which implements transparent replication, giving the clients the illusion that they are still using regular CORBA invocations. The replication mechanisms from the 7 projects are not transparent and require application support. In MEAD, we use group communication and membership for coordinating among server replicas; the Replication Manager is distributed to avoid a single point of failure. The 7 projects make simplifying assumptions (as is to be expected of a 15-week course endeavor) about the fault model and use a centralized Replication Manager. We did not use the Naming Service in our previous experiments, although MEAD has this capability. MEAD and its micro-benchmark are written in C++, while all the 7 projects here used Java. Most importantly, our previous micro-benchmark involved a test application of our design and with which we were intimately familiar. The advantage of this multi-team study is that we were largely observers and analyzers of the data that the students collected, and were, thus, able to be more objective and fair across all applications.

3.2 Project Descriptions

The seven projects were implemented in Java, using CORBA (two projects) or EJB (five projects) middleware. Each project has a three-tier architecture, with a client issuing all the requests, a stateless middle tier implementing the business

Table 1. Comparison of the seven projects implemented for the Fault-Tolerant Distributed Systems course (Spring 2006). The students were free to choose the application, the programming language, the middleware and the replication mechanism. All the teams chose to program in Java.

Project (Members)	Middleware	Replication Style	Request Size	Reply Size	DB Access
1: Su-Duel-Ku (5)	EJB	Warm Passive	4 b	≈200 b	100%
2: Blackjack (5)	EJB	Warm Passive	≈30 b	≈56 b	100%
3: FTEX (5)	EJB	Warm Passive	≈30 b	≈50 b	100%
4: eJBay (6)	EJB	Warm Passive	116 b	98 b	100%
5: Mafia (4)	EJB	Warm Passive	≈41 b	4 b	100%
6: Park'n Park (5)	CORBA	Warm Passive	3 b	4 b	75%
7: Ticket Center (5)	CORBA	Active	≈16	4 b	100%

logic and a MySQL database that stores the persistent state. The middle tier is replicated for fault-tolerance, using active (one project) or warm-passive (six projects) replication. Each project team had between 4 and 6 members. The characteristics of the seven projects are summarized in Table 1.

Su-Duel-Ku. Team 1 has implemented an online game where two or more Sudoku players can pit their intelligence against each other. Sudoku is a puzzle-solving game designed for one player; Su-Duel-Ku allows up to five players to work concurrently on the same board, while the server ranks the players and determines the winner of each confrontation. This application allows clients to create a new Sudoku board, to solve a board and to list all existing boards. and warm-passive replication to preserve the game continuity. The names of the players and the boards are stored in the database; the middle tier uses stateless entity beans. Due to its original idea, the project was mentioned in a local newspaper [12].

Blackjack. Team 2 has implemented a gaming application where users play Blackjack online. Users can create online profiles (stored in the database), place bets and play against the house.

FTEX. Team 3 has implemented the infrastructure for an electronic stock exchange (*e.g.*, NASDAQ, Island, Archipelago). Users can create online profiles, list the current orders for a stock and place buy and sell orders (either market-price or limit); the application matches buy and sell orders automatically. The user profiles and the details of all the transactions are stored in the database.

eJBay. Team 4 implemented a distributed auctioning system, similar to eBay, that allows users to buy and sell items in an auction plaza. The application allows posting items for sale and bidding for them; the user profiles and the information related to auctions (including text-file descriptions) are stored in the database.

Mafia. Team 5 implemented an online version of the popular "Mafia" game, where users create character profiles and communicate through instant messaging. The application stores the persistent state of the game in the database.

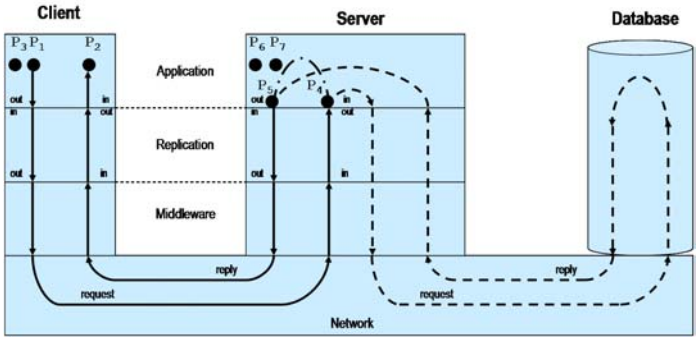


Fig. 2. Lifetime of a request in a 3-tier application. We record the time elapsed during all the different stages of the request to construct the latency profile of each application.

Park'n Park. Team 6 implemented a system for managing parking lots. The application keeps track of how many spaces are available in the lots and recommends alternative locations when a lot is full.

Ticket Center. Team 7 has implemented an online ticketing application for express buses, allowing users to search schedules and available seats, buy and cancel tickets and check reservation status.

3.3 Data Collection and Analysis

Figure 2 shows all the stages in the traversal of an end-to-end invocation in a three-tier system: The client issues a request and sends it to the server. In turn the server, depending on the semantics of the invocation, decides if it is necessary to contact the database to complete the request. After processing and receiving the information needed from the database, the server sends the reply back to the client. For closely monitoring the behavior of the applications, we defined the following seven probe points for monitoring the flow of requests:[1]

$\mathbf{P}_1$: Time (in μs) when each request is issued (client-side);
$\mathbf{P}_2$: Time (in μs) when each reply is received (client-side);
$\mathbf{P}_3$: Name of each invocation (client-side);
$\mathbf{P}_4$: Time (in μs) when each request is received (server-side);

[1] Since Java does not provide a method for accurate time measurement, the timestamps were recorded using a Java Native Interface (JNI) invocation of the `gettimeofday()` system call. This Linux system call provides a very accurate timer, based on the CPU cycle counter, allowing us to record timestamps with microsecond precision. We pre-allocate buffers in memory to store the probe data and to flush these buffers to the disk only at the end of each experiment in order to minimize the interference from the experimental harness.

$\mathbf{P}_5$: Time (in μs) when each reply is completed (server-side);
$\mathbf{P}_6$: Name of each invocation (server-side);
$\mathbf{P}_7$: Time (in μs) when each request is received (server-side).

We compute the end-to-end latency of each request by calculating the difference between the timestamps recorded by probes P_2 and P_1 for that request. By examining the difference between P_5 and P_4, we further decompose the latency into two components: *middleware()*, representing the time spent inside the middleware layer and network delay between client and middle tier, and *server()*, representing the response time of the database and of the business logic. For each request i:

$$\begin{aligned} latency(i) &= P_2(i) - P_1(i) \\ server(i) &= P_5(i) - P_4(i) \\ middleware(i) &= latency(i) - server(i) \end{aligned}$$

Each team selected a representative workload for its project and measured the end-to-end latency for these workloads. There is a significant spread between observations recorded in each experiment, with latency ranges[2] up to 200s. However, we need a method for comparing the latencies of different applications that does not depend on the scale of these observations, i.e., we would like to assess the number and the magnitude of the high latencies from each experiment whether these latencies are measured in seconds or milliseconds. We therefore compute the standard deviation σ for each sample[3] and we characterize each measurement in terms of the number of standard deviations away from the mean. This measure is called the *z-score*:[4]

$$z = \frac{latency(i) - \overline{latency}}{\sigma}$$

We determine the extreme latencies using the 3σ statistical test: any observation that deviates from the mean with more than 3σ is considered an outlier (*i.e.*, all outliers have $z > 3$). 3σ is a statistical test widely used in engineering for quality control or for identifying measurement errors [13]. The 3σ test helps us detect unlikely values in our measurements: if the recorded latencies followed a normal distribution, only 0.1% of the observations would fail the 3σ test. Note that this does not mean that all the outliers are unacceptably high or that they indicate an incorrect operating mode; this test simply helps us isolate the very large latencies for further analysis.

[2] The *range* of a measurement sample is the difference between the largest and the smallest observation. This measure of data spread depends on the scale in which the measurements are made and on the number of observations.

[3] σ is the non-biased standard deviation error:$\sigma[X] = \sqrt{\frac{n\sum x^2 - (\sum x)^2}{n(n-1)}}$

[4] The z-score of an observation is an adimensional measure that can be compared directly with z-scores from measurements expressed on a different scale.

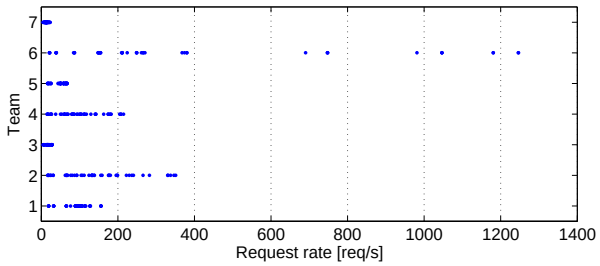


Fig. 3. Distribution of request rates for the 7 teams. The request rates depend on the response time, the number of clients and the client-side think time.

Each team has tested their application in *48 different configurations*, varying the number of clients, the request rate and the message sizes[5]. Two-way replication was used for the middle-tier server, in active or warm-passive mode depending on the design of each application. The students tested different combinations of several numbers of clients, inter-request ("think") times and sizes for the reply messages, varying one parameter at a time.

Number of clients. The teams ran experiments with 1, 4, 7 and 10 clients. This parameter affects the amount of server-side concurrency, which should lead to an increased average latency as requests compete for access to the database and for processing time inside the application server. The number of clients may also influence the maximum latency and, therefore, the size of the outliers. As each client connection requires the server to keep volatile (session) state, an increasing number of clients leads to higher memory requirements, potentially leading to server overload.

Request rates. For our experiments, the application can be modeled as a closed queuing system [14] where the server-side request rate is influenced by the response time, the client-side think time between requests and the number of clients. The teams simulated different request rates by varying the number of clients and by introducing inter-request think times of 40 ms, 20 ms and 0 ms (no pause). We compute the request rate by counting the number of requests arriving each second at probe P_4. Figure 3 shows the request rates issued by the seven teams during the experiments; the discrepancies are due to the different response-time profiles of their applications. The request rate also affects the average latency due to server-side concurrency. If the incoming load exceeds the server's capacity, the server loses its ability to process requests in a timely manner; this manifests in the form of successive invocations that are rejected or that exhibit a very high latency.

Size of the reply messages. Each team ran experiments with the original messages used by their applications, but also with reply messages carrying mod-

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

ified payloads of exactly 256, 512 and 1024 bytes. This was achieved by adding a parameter to each invocation from the workload and padding the messages up to the required size. The reply size may affect the average response time, due to message fragmentation and reassembly in the middleware/protocol layers, but the dependence will not be linear. The thresholds for fragmenting messages depend on the protocols used and on the network configuration. In [5], we reported that this is the only parameter that seems to affect the number and size of outliers. However, large numbers of high-latency spikes occurred in the case of reply messages with a payload of 64 KB. In this paper, we tried to eliminate one of the leading causes of unpredictability by focusing on the behavior of small messages.

3.4 Challenges and Fallacies

We encountered several challenges in analyzing and interpreting the experimental results. These challenges are derived from the fact that students from different teams interpreted the empirical requirements in slightly different ways, dedicated different amounts of work for this phase of the project (as is typical of students taking a course), and used systems with widely different robustness characteristics. None of this is surprising in hindsight – we served as independent observers and these discrepancies were an inherent side-effect of our intentionally electing not to be too familiar with the internals of the systems.

In some cases, the teams made a number of honest mistakes that rendered their data slightly different. Some of these problems were easily detected; for instance, teams 5, 6 and 7 ran the experiments with only 100, 2243 and 1000 invocations, respectively, instead of the 10,000 required. Other variations were subtle and harder to detect. Team 2 used 32-bit integers to store their timestamps, which led to an overflow for the long-running experiments with 1024-byte reply messages. Team 3 recorded their timestamps in milliseconds instead of in microseconds. Team 6 switched the incoming and outgoing probes at the client-side, which resulted in apparently negative values for the end-to-end latencies. Because of the allowed server-side concurrency, the order in which requests were recorded in probes P_4 and P_5 was not the same, which complicated the calculations of the middleware and server latencies. We have corrected or excluded all the corrupted data that we were able to detect; in the following sections, we candidly share all the instances where we believe that some data inconsistencies might have biased the results. We believe that these observations will be useful to others attempting similar multi-team experimental case studies.

Even though the students were instructed to document all the configuration parameters (*e.g.*, the size of the requests from their workloads) relevant to their experiments, some teams chose to report "variable" instead of the actual size used. For some of these cases, we report in Table 1 an estimate based on the team's qualitative verbal description. The database server crashed once (due to the high request-load during the hours before the course's deadline for handing in the results), and the folder where the students were uploading their data exceeded its quota. However, the students were aware of these problems and we

have made every effort to ensure that these events do not influence the data presented here (except, maybe, the highest outlier observed for team 2).

Most importantly, we were concerned that some students might be tempted to adjust their data in order to establish the magical 1% hypothesis that we are trying to test. We informed the students repeatedly that their task was not to confirm or refute the "magical 1%" effect, but to formulate an honest and well-documented opinion on the behavior of their system. We believe that we have succeeded in eliminating this bias. In fact, one of the teams concluded in their final report that their results strongly contradict the "magical 1%"; upon closer examination, it became apparent later that this statement was based on a data-representation error, where the maximum and 99^{th} percentile latencies were plotted on different graphical scales.

4 Experimental Results and Analysis

For all 7 projects, the average response time is well correlated with some of the variables of our controlled experiment. We have observed four trends for the average latency:

- it increases linearly with the number of clients and it scales well with the reply size (for teams 1, 3 and 5);
- it increases linearly with both the number of clients and the reply size (for teams 2 and 7);
- it is significantly higher for the experiments with 10 clients and it is otherwise unaffected by the reply size and client concurrency (for team 4);
- it is significantly higher for the experiments with 10 clients and it increases linearly with the reply size (for team 6).

The message size usually affects the projects with a low latency, where network delays have a significant impact on performance. We can observe this influence in the case of project 6, which has a very low latency and sustains request rates an order of magnitude higher than the other applications, as shown in Table 2. This is probably due to the fact that, for some of the requests from its workload, the middle-tier server does not need to contact the database and does not incur the additional latency (see Table 1). Due to this property and to the fact that this is a CORBA application, project 6 is closest to our experimental setup from [5].

Figure 4 summarizes our experimental results. On the left-hand side, we compare the average, the 99^{th} percentile and the maximum latency for each configuration, sorted by increasing average latencies. While the latencies of the seven applications have widely different magnitudes (summarized in Table 2, we can observe that, in most cases, the 99^{th} percentile closely follows the predictable trend of the mean, while the maximum latency seems uncorrelated and may be several orders of magnitude higher than the average. Team 1 is an exception, as the maximum latency seems to follow the trend of the mean as well. This is the only project that has achieved such predictability.

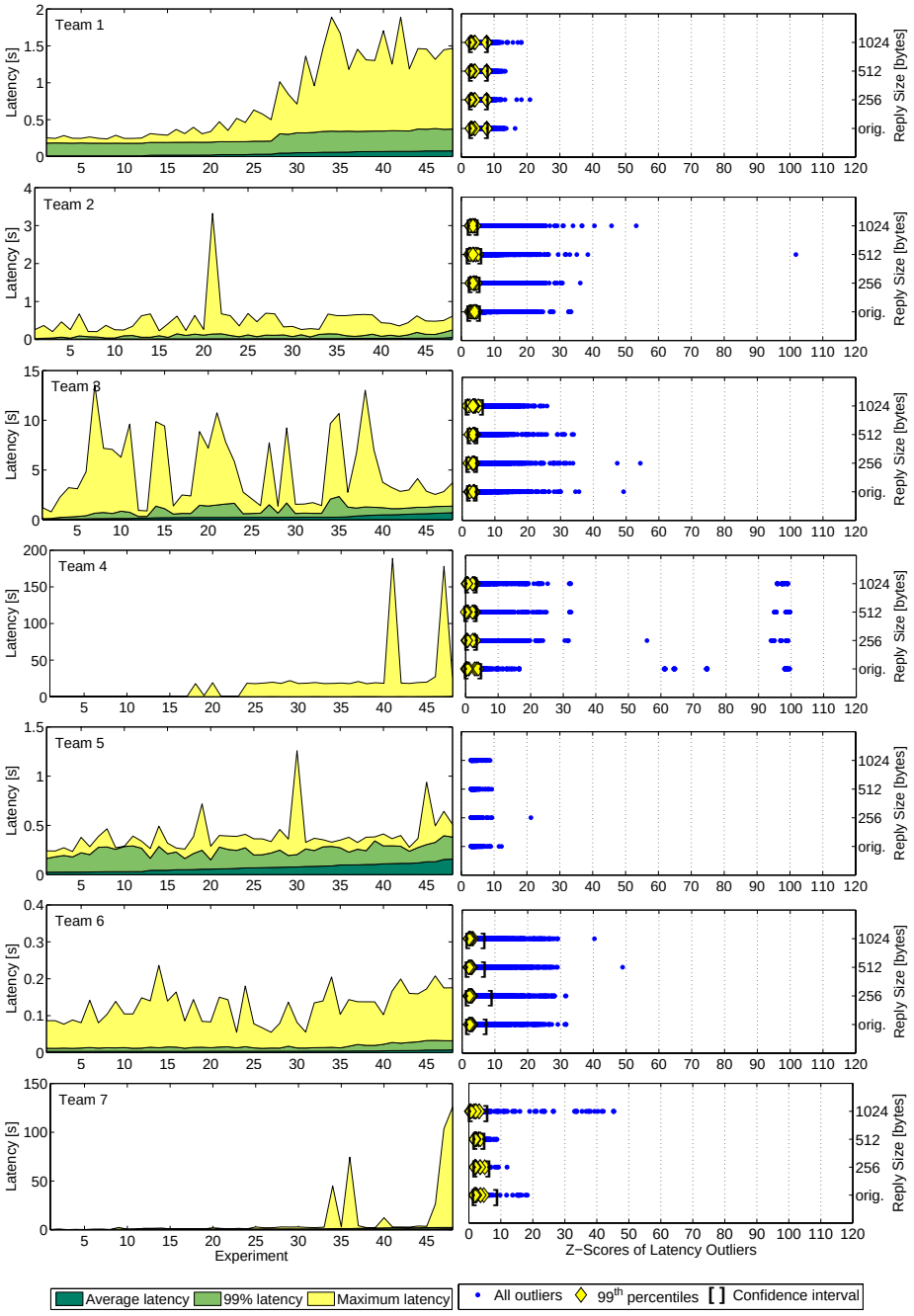


Fig. 4. The magical 1% in the seven projects. The plots on the left-hand side compare the average, the 99$^{\text{th}}$ percentile and maximum latency for each experiment. The experiments are sorted by increasing average latencies. The maximum latencies are usually unpredictable, but by filtering out the highest 1% of the recorded latencies we eliminate this unpredictability (as indicated by the fact that the 99$^{\text{th}}$ percentile closely follows the trend of the average latency). The right-hand side plots indicate the z-scores of all the outliers and of the 99$^{\text{th}}$ percentile latency (together with the corresponding confidence interval). The 99$^{\text{th}}$ percentile always has a low z-score.

Table 2. Characteristics of end-to-end latency and outliers in the 7 projects.

Project	1	2	3	4	5	6	7
Max. req. rate [req/s]	160	350	28	215	67	1250	24
Latency range [ms]	5–1900	4–3400	32–11600	3–190000	8–1300	1.5–300	10–125000
Number of outliers [%]	2.70%	1.68%	1.50%	0.33%	1.85%	0.86%	0.43%
Size of outliers (z-score)	20.95	101.78	54.37	99.51	21.24	48.91	45.28

The large maximum latencies are due to a few outliers. On the right-hand side of Figure 4 we plot the z-scores of all the outliers detected using the 3σ test; naturally, all the data points from the figure have a z-score higher than 3. A z-score indicates the difference between the outlier and the average latency from the corresponding experiment. We observe in the figure that Team 1 did not record any outliers with z-score > 25, while Team 4 recorded multiple outliers with z-scores between 80 and 100. The plots also suggest that the very large latencies are not very common; for all the teams, the vast majority of outliers have z-scores below 20.

We also plot the z-scores of the 99^{th} percentile latency and the corresponding confidence intervals at the 99% confidence level (*i.e.*, we plot the 99% confidence interval of the 99^{th} percentile latency). This gives us a statistically-significant ($p < 0.01$) indication that the 99^{th} percentile of the latency has a low z-score (less than 10 in all the cases). We could not compute the confidence interval for team 5, due to the small size (only 100 remote invocations per client) of the data sets produced by the students. Figure 4 seems to suggest that, with the exception of a couple of outliers, the average and maximum latencies are correlated and that project 5 has the smallest outliers; however, this may be an artifact of the limited number of requests used for these experiments.

The outliers come in occasional bursts, but they occur uniformly throughout the entire run-time of each experiment. This suggests that the number of clients and the request rates were not high enough to overload the middle-tier servers.

4.1 The Magical 1%

The example of team 1 suggests that the 3σ test is conservative and that some of the outliers detected using this method are not very large. Our goal is to remove *all* the extreme latencies and to obtain a predictable latency profile. According to the magical 1% hypothesis, we investigate if the unpredictability is confined to the highest 1% of the remote invocations. We turn our attention to the 99^{th} percentile of the latency.

Across all the experiments in our data set, there is a high correlation between the 99^{th} percentile and the average latency ($r = 0.91$). A two-tailed t-test [13] indicates that this value is statistically-significant. This correlation, which can also be observed in Figure 4, suggests that all the average-latency trends dis-

cussed above apply to the 99$^{\text{th}}$ percentile as well and that the resulting latency profile is predictable.

The confidence intervals for the z-scores of the 99$^{\text{th}}$ percentile help us establish soft latency bounds. The results are strikingly similar across all project teams:[6] the high limit of the confidence interval has a z-score between 4.53 and 8.78. *With a statistical confidence level of 99%, the data indicates that the 99th percentile of the latency cannot have a z-score higher than 10 and cannot be more than 20 times larger than the average latency.* In other words, the 99$^{\text{th}}$ percentile would never fail a 10σ test.

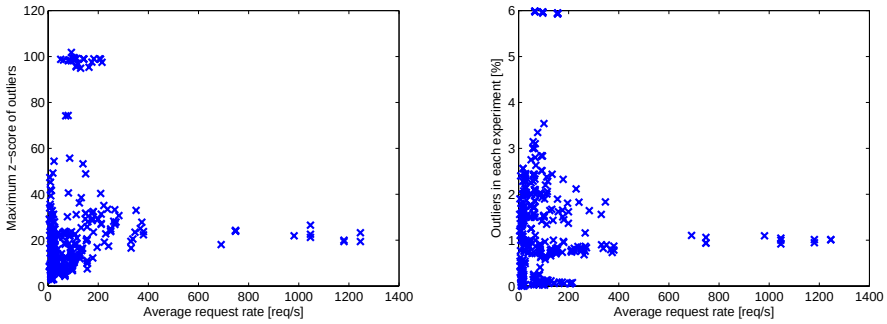
4.2 Sources of the Unpredictability

We examine the *server* and *middleware* components of all the outliers recorded in our experiments (see Section 3.3) to determine what caused the high latency. With one exception (team 6), all the applications may produce outliers due to either the latency of contacting the database or to the processing performed within the FT middleware. The component responsible for most of the outliers varies among the seven applications: for teams 1 and 2 most outliers originate in the database queries, while for team 3 most of them originate in the middleware. For team 6, all the outliers recorded originated in the middleware.

Other factors may also influence the production of outliers:

- The response time for certain invocations increases in time, as objects accumulate in the database, and this effect is amplified by a growing number of clients. However, this affects the average latency as well and does not explain the discrepancy between the trends of average and maximum latency. Moreover, the students have identified this problem and tried to compensate for it: team 1 decided to use a workload that does not add information in the database, and team 3 cleared the database between experiments.
- Team 4 discovered that, by increasing the Java heap size, the number of large outliers can be drastically reduced. This is likely due to the fact that request processing causes a high memory churn on the server, which forces the garbage collector to run more frequently.
- Different applications may produce either many small outliers or a few large ones. Table 2 shows that team 1 has generated the largest number of outliers (2.7% of all invocations) among the 7 projects, which contrasts with the apparently-predictable behavior of this application. The only outlier with a z-score higher than 100 was recorded in project 2, while project 4 (even after increasing the Java heap size) has produced an outlier 3556 times larger than the average latency from the corresponding experiment. Team 4 has also produced the fewest outliers (0.33% of all invocations) among all projects. This inverse proportionality between the number and size of outliers is consistent with our observations from [5].

[6] We exclude Team 5 because we could not compute the confidence intervals due to the small sample sizes reported.



(a) Relative size of outliers (z-score). (b) Number of outliers.

Fig. 5. Impact of the request-rate on the outliers.

We also investigate if outliers are correlated with certain values of the configuration parameters, to determine if certain operating modes of the applications are more unpredictable than others. Table 2 suggests that outliers usually account for around 1% of the remote invocations, except for project 1 which produces more outliers. The relative size of the outliers also varies among projects. Four projects (2, 3, 4 and 7) exhibit maximum latencies two orders of magnitude higher than the average, and project 4 has recorded outliers three orders of magnitude higher that the average.

Figure 5 shows the impact of the request rates on the numbers and sizes of outliers in all the experiments. The data points with request rates larger than 600 req/s correspond to team 6, which was the only project able to sustain such high throughput. The data points with very large outliers (z-score ≈ 100) from Figure 5(a) come from teams 2 and 4, and the large numbers of outliers (up to 6% of the requests, in some experiments) from Figure 5(b) correspond to team 1. Examining the correlation coefficients and the individual scatter plots for each team (not included here for lack of space) reveals that there is no significant correlation between the request rates and the z-scores of the outliers. The impact on the number of outliers differs for each project: for teams 1 and 5 there is no significant correlation, teams 2, 4 and 7 display low, negative correlation ($r \in [-0.39, -0.57]$) and team 6 has low, positive correlation ($r = 0.61$). The only case with a high, positive correlation ($r = 0.8$) between request rates and the number of outliers is team 4.

Table 3. Impact of the reply size.

[bytes]	**original**	**256**	**512**	**1024**
% outliers	1.37%	1.37%	1.32%	1.28%
z-score	99.51	99.01	101.78	98.95

Table 4. Impact of the # clients.

Clients	**1**	**4**	**7**	**10**
% outliers	1.31%	1.70%	1.35%	0.98%
z-score	53.25	101.78	99.18	99.50

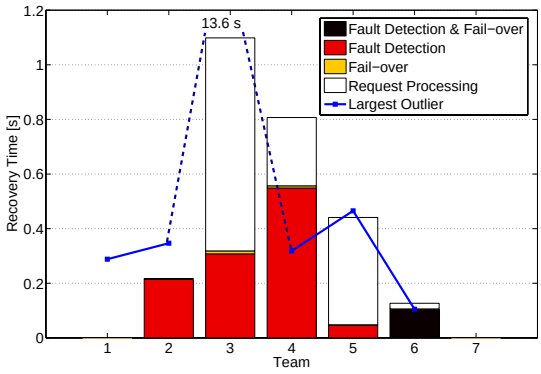


Fig. 6. Recovery time after a crash fault.

The results are similar for the impact of the reply size (Table 3) and number of clients (Table 4) on the outliers. The number of outliers is uniformly distributed among all the tested values of these two parameters. The corresponding maximum sizes of outliers are comparable for all these values, with the exception of the experiments with a single client connection, which exhibit lower z-scores. The impact of the reply size on the relative z-scores of the outliers can also be observed in Figure 4. These results suggest that, in general, the request rate, the number of clients and the message size do not have a significant impact on the generation of outliers.

4.3 Comparison with Fault-Recovery Time

So far, we have shown that the 7 FT applications may have unpredictable response times even in the absence of failures. It is interesting to compare these random high latencies occurring during the normal operation mode with the time the recovery time needed after a crash fault. A single fault in a middle-tier server does not induce an outage because the servers are replicated. No requests are lost and the clients do not have to reconnect, but they experience a high latency while the fault-tolerant infrastructure carries out recovery actions. After each crash, we launch a new server replica to restore the 2-way replication of the middle tier and to prepare the application for handling the next fault.

The students conducted a series of fault-injection experiments by provoking 10-20 crash faults in the middle tier while 1 client was connected. Based on preliminary results, the teams have optimized their system for improved recovery times by maintaining object references to all the server replicas and by keeping open TCP connections to these replicas in order to avoid time-consuming name lookups and the overhead of connection establishment during the failover process. Figure 6 shows the recovery times after this optimization stage; each bar represents the average round-trip time of the requests issued when faults were injected. We break down the recovery time into components corresponding to fault-detection, failover and normal request processing. Since this phase of the

project was designed as an opportunity to obtain bonus points, these results are reported unevenly: teams 1 and 7 did not provide recovery-time numbers, while team 6 lumped together the fault-detection and failover times.

Figure 6 indicates that the largest contributor to the recovery time in these systems is the delay introduced by fault detection. As all the applications use stateless middle-tiers, the failover process is very fast. In consequence, the normal processing time of the request brings a significant contribution to the fault-induced latency outliers. We compare these fault-induced outliers with the outliers recorded in the fault-free experiments with 1 client connection. The fault-induced outliers are significantly higher for team 4, and they are comparable with the fault-free outliers for teams 2, 5 and 6. Team 3, however, has recorded 453 outliers larger than its recovery time; the largest such outlier (13.6 s) is one order of magnitude bigger than the recovery time. This indicates that, under certain circumstances, high latencies occurring randomly during normal operation may have higher impact on availability than hardware crash faults.

The low recovery times achieved by these applications are due to the stateless nature of the replicated servers, which does not mandate a long fail-over process. Enterprise three-tier systems usually store volatile state, such as sessions or cached content, in the middle tiers, and they keep their persistent objects in a database [15]. Because volatile state can be recreated after a fault and does not need to be synchronized, the low recovery times reported by these applications are realistic. This result brings a new aspect into the cost/benefit trade-off of FT middleware. It suggests that it may not be worth optimizing the failover process for achieving a very low recovery time, since comparable outages are expected to occur during normal operation. It does not diminish the usefulness of FT middleware because the fault-tolerance techniques prevent data loss in fault scenarios similar to the ones examined here. Moreover, without FT middleware, the outage following a hardware crash would much longer due to the need to bootstrap a new version of the system.

5 Implications of the Magical 1%

The unpredictability of end-to-end response times has been observed and documented for many COTS-based systems [4]. The unpredictability of FT middleware [3, 2, 4, 5] undermines many critical systems that rely on the fault-tolerant mechanisms to increase their reliability and availability under any conditions and to help them deliver a predictable behavior continuously. Unfortunately, in such complex systems the unpredictability has multiple sources and it proved resilient to our attempts to eliminate it by choosing the best COTS components available and by carefully configuring the system [5].

5.1 Eliminating the Root Causes of Unpredictability is Impractical

While the typical source of outliers depends on the application, most of the seven teams have recorded that the outliers may originate either from the interactions with the database server or from the middleware itself. In our experiences with

the MEAD system, where we had a finer-grained monitoring infrastructure, we have observed outliers originating in all the components of the system: the group communication protocols, the middleware, the replication mechanism and even the microbenchmark server, which has under 100 LOC (the group communication accounted for the vast majority of outliers) [5]. The only parameter settings that show a correlation with the outliers are the message size for MEAD (a large number of small outliers occur for reply messages larger than 64 KB) and the request rate for Team 4 (the number of outliers increases linearly with the request rate, but there is no correlation with the size of outliers).

It is tempting to draw conclusions on which design choices affect the performance and predictability of the system. The seven applications have similar architectures, but the resulting latency profiles are widely different. Each application has a certain propensity to produce outliers, but, as these outliers may originate in different components of the applications, we are unable to suggest a general method for preventing them. Since only two teams used CORBA instead of EJB and only one team used active replication, we cannot make a rigorous comparison between the impact on system predictability of these choices.

Team 1 has achieved a latency profile that seems to be fairly predictable, but this may be a result of the limited number of configurations – 48, compared with the 960 we reported in [5] – that each team was able to test during the 15-week course. Based on the evidence presented here and in the related literature, we conclude that there is no silver bullet that will render a FT middleware application predictable and that, in general, the maximum end-to-end latency is hard to bound. Most likely, this behavior is the result of combining COTS components which: (i) were not built and tested together, and (ii) were designed to optimize the common case among a wide variety of workloads, rather than to enforce tight bounds for the worst-case behavior. This unbounded behavior comes in addition to the unpredictability related to the potential occurrence of faults and it is not negligible compared to the fault-recovery times. How to enforce strict predictability, in order to establish guarantees for the maximum latency of FT middleware, remains an open research question.

5.2 Statistical Predictability for Enterprise Applications

The magical 1% hypothesis states that, for many FT middleware systems, the unpredictability – if it exists – is limited to less than 1% of the remote invocations. We emphasize that 1% is a rule of thumb and not the statistical limit suggested by the experimental data; for instance, our results from the MEAD experiments show that the distribution of the highest 1% of response times has a long tail and that 0.1% might be enough to remove the unpredictability. However, for all the systems that we have examined, removing the magical 1% resulted in a predictable latency profile. This hypothesis holds true for different operating systems (TimeSys and SUSE Linux), middleware platforms (CORBA and EJB), programming languages (C++ and Java), replication styles (active and warm passive) and applications. This is the magic of the unruly 1%.

This indicates that *statistical predictability* is within our reach: upper bounds for the 99$^{\text{th}}$ percentile of the latency can be established with a high confidence, which allows the developers of FT middleware applications to establish and honor service-class guarantees. Reliable percentile-based guarantees are desirable in most enterprise and soft real-time systems.

For example, a predictable latency profile would enable the automatic negotiation of service-level agreements between service providers. According to the Web-Services Agreement specification [16], service providers advertise a list of standard offerings that clients can select from. Clients requesting access to the service will typically communicate a description of their workload and the desired QoS guarantees. In return, the service provider will communicate the price and, potentially, a penalty function for violating the QoS guarantees. The standard also allows the service to describe several options and to assign priorities among these options. Service providers must make informed business decisions during the SLA negotiation, such as how many new clients they can admit before compromising the quality of the service provided to their existing clients, or what class of service it can reliably deliver based on the observed load on the system. These decisions can only be based on an accurate model of the system behavior, which seems achievable with the statistical predictability obtained by removing the magical 1% of invocations.

5.3 Limitations of Statistical Predictability

Statistical predictability is not relevant in all the situations. We have verified the magical 1% hypothesis in multiple configurations on a local-area network, which is a typical setting for fault-tolerant middleware systems. This hypothesis may not hold in a wide-area network with high propagation delays, or in an environment with intermittent network connectivity, such as a wireless network. virtual, rather than physical computing resources. Moreover, certain applications (*e.g.*, embedded, real-time systems) will not be able to use percentiles; in such cases, nothing short of predictable worst-case behavior will be sufficient to ensure safety.

6 Summary of Findings and Conclusions

In this paper, we present an experiment evaluating the unpredictability of enterprise applications using fault-tolerant middleware. We compare seven three-tier applications using CORBA or EJB middleware and active or warm passive replication mechanisms. We identify one project that seems to have achieved predictable latencies (Su-Duel-Ku) and one project where the results were inconclusive due to incomplete data (Mafia); the remaining five projects exhibit unpredictable and sometimes very high latencies (up to three orders of magnitude higher than the average response time). Some of the high latencies observed may be due to contention between projects for the LAN bandwidth and

for the database server. However, these conditions are typical for enterprise infrastructures where multiple applications share communication and computing resources.

We present strong empirical evidence supporting the hypothesis that unpredictability is confined to the highest 1% of the remote invocations: the 99^{th} percentile of the latency cannot have a z-score higher than 10 and is typically at most 20 times larger than the mean These bounds for the majority of the remote invocation latencies are significant at the 99% confidence level. For two of the projects (Blackjack and Mafia) and with a single client connection, the recovery time after a fault is comparable to the maximum latencies recorded in the fault-free case.

While the unpredictability of FT middleware is unsettling, the knowledge that this unpredictability is limited to a small number of remote invocations – typically less than 1% – enables the establishment of high-confidence bounds on the 99^{th} percentile of the latency. This style of statistical predictability is relevant for many enterprise applications. As we propose an easily-verifiable hypothesis, which would provide significant benefits for designing and deploying fault-tolerant applications, we encourage the research community to verify the validity of the magical 1% rule in other middleware systems.

Acknowledgments. We would like to thank the students from the Spring 2006 installment of the Fault-Tolerant Distributed Systems course at Carnegie Mellon University for their efforts in developing and evaluating the 7 applications. We are also grateful to Arun Iyengar for asking the question that motivated this research.

References

1. Felber, P., Narasimhan, P.: Experiences, approaches and challenges in building fault-tolerant CORBA systems. IEEE Transactions on Computers **54** (2004) 497–511
2. 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)
3. Zhao, W., Moser, L.E., Melliar-Smith, P.M.: End-to-end latency of a fault-tolerant CORBA infrastructure. Performance Evaluation **63** (2006) 341–363
4. http://www.atl.external.lmco.com/projects/QoS/.
5. Dumitraş, T., Narasimhan, P.: Fault-tolerant middleware and the magical 1%. In: ACM/IEEE/IFIP Middleware Conference, Grenoble, France (2005) 431–441
6. 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
7. Alistair Croll: Meaningful Service Level Agreements for Web transaction systems. LOOP: The Online Voice of the IT Community (2005)
8. Object Management Group: Fault Tolerant CORBA. OMG Technical Committee Document formal/2001-09-29 (2001)

9. Budhiraja, N., Schneider, F., Toueg, S., Marzullo, K.: The primary-backup approach. In Mullender, S., ed.: Distributed Systems. ACM Press - Addison Wesley (1993) 199–216
10. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys **22** (1990) 299–319
11. Dumitraş, T., Srivastava, D., Narasimhan, P.: Architecting and implementing versatile dependability. In de Lemos, R., Gacek, C., Romanovsky, A., eds.: Architecting Dependable Systems III. Springer-Verlag, LNCS 3549 (2005) 212–231
12. Hentges, R.: Puzzling: Sodoku has grabbed the short-attention span of a nation. Pittsburgh Tribune Review (2006) http://www.pittsburghlive.com/x/pittsburghtrib/search/s_447266.html.
13. National Institute of Standards and Technology: (Engineering statistics handbook) http://www.itl.nist.gov/div898/handbook/index.htm.
14. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: Computer System Analysis Using Queueing Network Models. Prentice Hall (1984)
15. Wu, H., Kemme, B.: Fault-tolerance for stateful application servers in the presence of advanced transactions patterns. In: Symposium on Reliable Distributed Systems, Orlando, FL (2005) 95–108
16. Global Grid Forum: Web services agreement specification (WS-Agreement). Draft, version 11 (2004)