Middleware, Fault-Tolerance and the Magical 1%
A Study of Unpredictability

Tudor Dumitraş, *Student Member, IEEE,* and Priya Narasimhan, *Member, IEEE*

Abstract—We present an extensive empirical study of unpredictability in 16 distributed systems, ranging from simple transport protocols to fault-tolerant, middleware-based enterprise applications, and we show that the inherent unpredictability in these systems arises from a *magical 1%* of the remote invocations. In the *normal, fault-free operating mode* most remote invocations have a predictable end-to-end latency, but the maximum latency follows unpredictable trends and is comparable with the time needed to recover from a fault. The maximum latency is not influenced by the system's workload, cannot be regulated through configuration parameters and is not correlated with the system's resource consumption. The high-latency outliers (up to three orders of magnitude higher than the average latency) have multiple causes and may originate in any component of the system. However, after selectively filtering 1% of the invocations with the highest recorded response-times, the latency becomes bounded with high statistical confidence ($p < 0.01$). We have verified this result on different operating systems (Linux 2.4, Linux 2.6, Linux-rt, TimeSys), middleware platforms (CORBA and EJB), programming languages (C, C++ and Java), replication styles (active and warm passive) and applications (e-commerce and online gaming). Moreover, this phenomenon occurs at all the layers of middleware-based systems, from the communication protocols to the business logic. The magical 1% suggests that, while the emergent behavior of middleware systems is not strictly predictable, enterprise applications could cope with the inherent unpredictability by focusing on statistical performance-indicators, such as the 99th percentile of the end-to-end latency.

1 Introduction

Modern distributed 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. These two problems are related because an effective tuning is rendered more difficult if the outcomes of configuration actions are hard to predict. *How to design dependable systems in spite of such inherent unpredictability remains an open question.*

Conventional wisdom about enterprise applications using commercial-off-the-shelf (COTS) components is that well-designed middleware systems provide sufficient predictability when running on top of certain real-time operating systems. Fault-tolerant (FT) middleware, used in the most critical enterprise and embedded applications, encounters even higher predictability expectations. Faced with a multitude of specialized configuration options—some of which demand an in-depth understanding of FT semantics[1]—practitioners often suspect that unpredictable behavior is caused by misconfiguration, rather than fundamental reasons. We naturally expect that faults, which are inherently unpredictable, will have a disruptive effect on the performance of the system. Moreover, we show that *even in the fault-free case it is hard, and usually impossible, to enforce any hard bounds on the end-to-end latency of CORBA or EJB applications with replicated servers*.

Existing approaches for the autonomic management of computer systems [2–7], including both commercial products and research prototypes, focus on predicting and tuning the average system behavior and do not address the residual unpredictability of well-configured, replicated servers in the normal, fault-free operating mode. In this paper, we test the hypothesis that the worst-case behavior of FT middleware is not truly random and that the fault-free unpredictability has natural limitations that can be leveraged when designing complex distributed systems. We conduct three separate studies. To assess the predictability of real-world middleware, we analyze black-box measurements of 5 middleware systems and 4 communication protocols, collected at Lockheed Martin's Advanced Technology Labs (henceforth called the *ATL trace*). To study unpredictability in depth, we present a controlled experiment trying to achieve a predictable configuration of the MEAD middleware [8]—which we had built, previously—where we dissect the sources of unpredictability through white-box monitoring of the system's components (henceforth called the *MEAD trace*). To analyze the impact of design and implementation on the unpredictability of the resulting system, we present a programming experiment where we perform gray-box comparisons of 7 CORBA and EJB applications—developed independently, under

• *T. Dumitraş and P. Narasimhan are with the Department of Electrical & Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213.*
E-mail: tudor@cmu.edu *and* priya@cs.cmu.edu

1. For instance, the FT CORBA standard [1] specifies ten low-level parameters for tuning the replication mechanisms: replication style, membership style, consistency style, fault-monitoring style, fault-monitoring granularity, location of factories, initial number of replicas, minimum number of replicas, fault-monitoring interval and timeout, and checkpoint interval.

Table 1
Summary of findings.

	Finding	Implications
Unpredictability of Real-World Middleware: The ATL Trace (Section 4)		
I.	The maximum latency of middleware systems does not follow a visible trend, and can be three orders of magnitude higher than the mean latency.	We cannot establish hard bounds for the end-to-end latency of middleware systems.
II.	The relative size of the highest response times and the number of high-latency requests are negatively correlated.	The fault-free unpredictability is due to a few large outliers.
III.	The 99^{th} percentile of the end-to-end latency is at most one order of magnitude higher than the mean and is correlated with the workload and the environmental conditions.	The fault-free unpredictability of real-world middleware is limited to 1% of the remote invocations. The 99^{th} latency percentile is predictable.
Sources of Fault-Free Unpredictability: The MEAD Trace (Section 5)		
IV.	The maximum latency cannot be regulated by adjusting the workload or the FT configuration, and it is not correlated with system-level metrics such as the time spent in kernel mode, the page faults or the size of the resident set.	Unpredictability cannot be eliminated by carefully tuning the system.
V.	The latency outliers may originate in any layer of the middleware stack.	The maximum latency is inherently unpredictable.
Impact of Design and Implementation on Unpredictability: The FTDS Trace (Section 6)		
VI.	The latency of middleware-based applications exhibits considerable variability, and infrastructure-level design choices, such as the middleware platform or the replication style, do not account for the fault-free unpredictability.	It is unlikely that the fault-free unpredictability can be eliminated by focusing only on the middleware infrastructure.
VII.	For enterprise applications with stateless middle tiers, the maximum fault-free latency is often comparable with the recovery time after crash faults.	The recovery time is not critical, as comparable outages are expected to occur during normal operation.

our supervision, during the "Fault-Tolerant Distributed Systems" course at Carnegie Mellon University—and we correlate their fault-free and fault-induced unpredictability with programming practices (henceforth called the *FTDS trace*).

For all 2402 system configurations examined, we find a statistically-significant ($p < 0.01$) result that the 99^{th} percentile of the end-to-end latency is at most one order of magnitude larger than the mean latency. We call this phenomenon *the "magical 1%": the unpredictability of COTS-based FT middleware is limited to at most 1% of the remote invocations*. This result suggests that, while strict predictability is hard to achieve in FT-middleware using COTS components, developers of fault-tolerant applications could focus instead on providing quality-of-service (QoS) guarantees for statistical measures such as the 99^{th} latency percentile.

Contributions. To the best of our knowledge, while related work has focused on evaluating the performance of fault-tolerant systems, this is the first broad empirical study (covering 16 different systems) to study the predictability of fault-tolerant middleware under fault-free and fault-induced conditions. In our study, we follow the old paradigm of empirical science stating that an experimentally-derived scientific law must necessarily be simpler than the data it tries to describe [9]. Consequently, our goal is to establish pragmatic rules for handling and reasoning about unpredictability, rather than to develop detailed regression models for the latency of the systems under examination. We find extensive evidence that middleware systems have unbounded latencies, owing to a few large outliers that have various causes and that may originate in any component of the system. This is not the result of an anomaly and is the normal, expected behavior of such complex systems, in the absence of faults and despite using the best components available. Furthermore, we show that, for *all* the applications and configurations examined, removing a "magical 1%" of the highest recorded latencies yields predictable latency profiles. Our detailed findings are summarized in Table 1.

Section 3 explains the design of our experiments and introduces the statistical tools that we use to analyze the collected data. We present our empirical study in Sections 4–6, and we summarize the results in Section 7. In Section 8 we discuss the implications of the magical 1% effect for designing fault-tolerant distributed systems, and in Section 9 we review the related work. The Appendix contains supplemental material.

2 Problem Statement and Goals

Middleware [10] is a layer of software (*e.g.*, CORBA, EJB, .NET) residing between an application program and the underlying operating system, network or database. Its goals are to help in transparently enforcing the separation of concerns between the functional and non-functional aspects of the application, to facilitate the portability across different platforms and to provide useful horizontal services (*e.g.*, directory look-up, security, transactions). Middleware allows distributed applications to make remote procedure calls (RPC), abstracting away the platform and protocol specific details for network programming. Additionally, fault-tolerant (FT) middleware [11] aims to provide transparency with respect to component failures, through mechanisms for recovery and replication. For example, FT middleware tolerates hardware failures by replicating the application on multiple physical hosts, using techniques such as active or passive replication [12, 13].

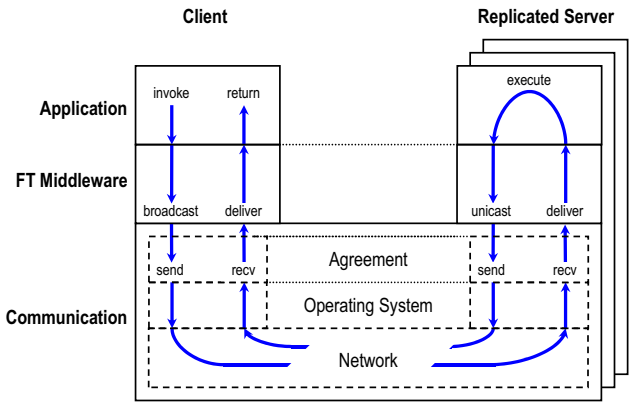


Figure 1. Anatomy of a remote procedure call. Some layers might be absent or might be collapsed into a single layer in particular implementations. Conversely, FT middleware systems might include additional components (*e.g.*, naming services, fault detectors, recovery managers), which are not invoked frequently during fault-free executions. There may also be additional tiers, *e.g.*, caching proxies between the clients and the business logic or databases in the back-end.

Figure 1 shows an idealized model of FT middleware. An RPC call traverses several layers of software, developed independently and optimized for the common case among a wide variety of workloads. The middleware layer marshals the invocation parameters, broadcasts them to the replicated servers, collects the replies and provides the return value to the client application. The communication among physical hosts is implemented using the Internet's transport protocols (*e.g.*, TCP [14]) or using group-communication protocols (*e.g.*, Spread [15], which enables reliable broadcasts and ordered deliveries of messages among a group of processes). Because middleware-based systems are assembled from multiple COTS components, the unpredictability can originate in any layer of the system. Moreover, the operations performed by one layer might dominate the end-to-end latency, hiding the unpredictability of the other layers.

Reportedly, the maximum end-to-end latency of CORBA and FT-CORBA middleware can be several orders of magnitude larger than the mean values and might not follow a visible trend [16, 17]. However, most of these findings have yet to be validated with extensive experimental or field data; for instance, it remains unclear if this unpredictability can be eliminated by fine-tuning the system. Our *first goal* in this paper is to reexamine the conventional wisdom that well-designed middleware systems behave in a sufficiently predictable manner by assessing the predictability of real-world middleware. We evaluate just how much predictability we can obtain by carefully choosing a good configuration of FT middleware components (operating system, middleware, group communication protocols and replication mechanisms). We conduct these experiments in a local-area network (LAN) setting to emphasize that unpredictability in FT middleware occurs even without the (expected) asynchrony of wide-area networks.

Our *second goal* in this paper is to assess the feasibility of autonomic management for enterprise applications by investigating the root causes of unpredictability and by formulating a practical rule for identifying the inherent unpredictability in the absence of faults. Aside from its statistical significance, we impose two additional requirements for this rule to be useful in practice:

- The rule must be broadly applicable to different systems, configurations, workloads and environments;
- The rule must be simple and easy to apply.

The second point is important because middleware systems incorporate complex mechanisms for adapting to a wide range of workloads and environmental conditions. We must therefore separate the inherent unpredictability from the effects of these mechanisms.

These empirical findings teach us important lessons about the behavior of complex, COTS-based distributed systems. Therefore, a *meta-goal* of this paper is to discuss the implications of inherent unpredictability and its limitations for the design and management of dependable enterprise systems.

In this paper, we also have some *non-goals*:

- We do not create complete behavioral profiles for each of the systems examined;
- We do not characterize the fault-induced unpredictability;
- We do not test the validity of our rule for systems co-located on the same physical host or systems communicating across wide-area networks.

Hypothesis. We test the following hypothesis:

> *The inherent unpredictability of FT middleware has natural limitations, which can be exploited when designing dependable enterprise systems.*

We consider that a system is unpredictable when it behaves according to one of the following three criteria (formalized in Appendix A.1):

- C1 The maximum latency is *too large*, compared with most of the other requests. Specifically, we analyze latencies that are more than one order of magnitude larger than the mean latency.
- C2 The latency profile includes *too many* requests with a high response time. Specifically, we estimate the number of requests that exceed the mean latency by more than 3 standard deviations (the 3σ test [18]).
- C3 The latency is *not influenced* by configuration parameters or by the environment. Specifically, we perform an analysis of variance (ANOVA [18]) to determine the correlation between these parameters and the end-to-end latency.

3 Experimental Methods

We compare the end-to-end (client-side) latency of 12 middleware and FT middleware systems—Java RMI,

JacORB, TAO, MEAD, JBoss and seven middleware-based applications—and of their underlying communication protocols—UDP, TCP, SCTP and Spread. These systems and protocols are described in more detail in Sections 4–6. We measure their mean, 99^{th} percentile and maximum latency.

We combine three experimental traces, ATL, MEAD and FTDS, which have been collected independently.[2] The ATL Trace (Section 4) contains black-box measurements of 9 systems, allowing us to assess the predictability of real-world middleware. In the MEAD trace (Section 5), we instrument a state-of-the-art FT middleware, MEAD, and we collect white-box observations in order to investigate the root causes of unpredictability. The FTDS trace (Section 6) was collected during a programming experiment, where the students of a graduate-level university course have designed and evaluated 7 fault-tolerant, middleware-based applications. While these applications range from e-commerce to online gaming, their design started from a common system architecture, which allows us to make gray-box comparisons and to assess the impact of design and implementation practices on unpredictability.

The systems covered by these three traces are closely related: the MEAD middleware (ATL and MEAD traces) uses the TAO object-request broker and the Spread group-communication protocol (ATL trace), while most of the middleware-based applications (FTDS trace) rely on the JBoss application server (ATL trace). This analysis—encompassing all the layers of the middleware stack (see Figure 1), evaluated both in isolation and when combined into a coherent system—provides deeper insights into the sources and limitations of unpredictability. Furthermore, as each of the three traces is likely to emphasize different system behaviors, combining these datasets allows us to provide a better coverage of unpredictability than previous studies. For example, the ATL and MEAD experiments use *micro-benchmarks*, designed to provide full testing coverage of the middleware features, while the FTDS experiments use *macro-benchmarks*, with realistic workloads exercising the business logic of their corresponding applications. The MEAD and ATL systems are two-tier, client-server applications. The FTDS applications use three tiers: the clients, which issue requests, the middle-tier servers, which implement the application's business logic, and the back-end database, which stores the persistent data.

3.1 Design of Experiments

We conduct controlled experiments by changing a single configuration parameter at a time, while making a reasonable effort to keep the remaining settings identical. There are no additional loads on the processors and no extra traffic on the network, in order to avoid interference with the experiments. Except for Section 6.2, all our results were recorded in the absence of faults.

The data from the ATL trace corresponds to client-side measurements, which do not affect the observed latency. In the MEAD trace, most data represents client-side measurements; we conduct a separate set of experiments for assessing the contribution of each server-side component to the overall unpredictability (Section 5.2). In the FTDS trace, we use only two server-side probes. Because of these precautions, it is unlikely that our instrumentation biases the results presented in this paper. Some systems (*e.g.* MEAD) are examined in two different experimental traces and produce similar outcomes, which confirms the repeatability of our results.

Impact of environmental conditions. We assess the environmental impact on the system unpredictability by conducting experiments in a wide variety of testbeds. The ATL experiments employ several operating systems: Linux 2.4 (minor versions 2–20), Linux 2.6 (minor versions 8–23), Linux-rt[3] (patches rt1–rt17) and TimeSys 3.1.[4] In the MEAD trace we also use TimeSys 3.1, and in the FTDS trace we use SUSE Linux (kernel 2.6). These experiments use a variety of hardware configurations, with CPUs ranging from Pentium III running at 850 MHz to 64-bit dual-core Xeon running at 3.0 GHz, and with 100 Mbps or 1 Gbps LANs (see Appendix A.1 for details).

Impact of FT configurations. We assess the effect of two configuration options for the FT mechanisms: the *replication style* (MEAD and FTDS traces) and the *replication degree* (MEAD trace). In the MEAD trace, we test each system configuration using both the active and the warm-passive replication styles;[5] for each FTDS application, the replication style is a static design choice. We use up to 3 server replicas in the MEAD trace and 2 server replicas in the FTDS trace.

Impact of workloads. We assess the impact of the workload by varying the *message payload* (ATL, MEAD and FTDS traces), the *number of clients* (MEAD and FTDS trace) and the *request rate* (MEAD and FTDS traces). We use payloads between 3 bytes – 4 MB. We use up to 22 clients in the MEAD trace and up to 10 clients in the FTDS trace. We induce different request rates by varying the "think time" between invocations.

Impact of faults. Finally, we compare the fault-free unpredictability with the recovery time needed after a crash fault (MEAD and FTDS traces). We inject faults by periodically crashing and restarting a server replica.

2. The ATL trace is available at `http://www.atl.lmco.com/projects/QoS`, while the MEAD and FTDS traces can be downloaded from `http://www.ece.cmu.edu/~tdumitra/FT_traces`. Preliminary, non-comparative analyses of the MEAD and FTDS traces have been published as [19] and [20], respectively.

3. Linux kernel patched to enable preemptibility (http://www.kernel.org/pub/linux/kernel/projects/rt/).

4. Linux-based commercial operating system, with a fully-preemptible kernel, protection against priority inversion, $O(1)$ task-scheduling complexity, and millisecond timer granularity.

5. In *warm-passive (primary-backup) replication* [13], one replica is actively processing requests while several backups are waiting to take over after a fault. In *active (state-machine) replication* [12], all the replicas accept and process the same requests, and the system tolerates faults as long as at least one replica remains functional.

Table 2
Systems evaluated in the ATL, MEAD and FTDS traces.

Experimental configurations		System type	Programming language	Faults tolerated	Latency range Mean latency	Full range
ATL Trace (Section 4)						
UDP [14]	89	Message-oriented transport protocol	C	—	47–832 µs	37 µs–11.6 ms
TCP [14]	423	Message-oriented transport protocol	C, Java	message loss	54 µs–35.9 ms	42 µs–209.9 ms
SCTP [21]	9	Message-oriented transport protocol	C	message loss, single link-failure	260–469 µs	251–879 µs
Spread [15]	60	Message-oriented group communication	C	message loss, node or link failure, network partition	0.3–8.3 ms	0.3–2014.5 ms
Java RMI [22]	30	Distributed-object middleware	Java	message loss	0.4–9.3 ms	0.4–118 ms
JacORB [23]	60	Distributed-object middleware (CORBA)	Java	message loss	0.2–8.4 ms	0.1–372.9 ms
TAO/TCP [24]	103	Distributed-object middleware (CORBA)	C++	message loss	87 µs–9.5 ms	75 µs–16.4 ms
TAO/SCTP [24]	9	Distributed-object middleware (CORBA)	C++	message loss, single link-failure	607–894 µs	595–1571 µs
MEAD [8]	68	Distributed-object middleware (CORBA)	C++	message loss, node or link failure, network partition	1–12.5 ms	0.8–2014.1 ms
JBoss [25]	15	Component middleware (EJB)	Java	message loss	0.7–2.4 ms	0.6–328.2 ms
MEAD Trace (Section 5)						
MEAD [8]	1200	Distributed-object middleware (CORBA)	C++	message loss, node or link failure, network partition	2–675 ms	1.7–5800 ms
FTDS Trace (Section 6)						
1: Su-Duel-Ku	48	Enterprise application (EJB based)	Java	message loss, single node-failure	10–77 ms	4–1890 ms
2: Blackjack	48	Enterprise application (EJB based)	Java	message loss, single node-failure	7–54 ms	4–3322 ms
3: FTEX	48	Enterprise application (EJB based)	Java	message loss, single node-failure	48–697 ms	32 ms–13.5 s
4: eJBay	48	Enterprise application (EJB based)	Java	message loss, single node-failure	15–125 ms	3 ms–189 s
5: Mafia	48	Enterprise application (EJB based)	Java	message loss, single node-failure	23–157 ms	8–1259 ms
6: Park'n Park	48	Enterprise application (CORBA based)	Java	message loss, single node-failure	3–7 ms	1.5–236 ms
7: Ticket Center	48	Enterprise application (CORBA based)	Java	message loss, single node-failure	61–1018 ms	10 ms–125.6 s

3.2 Data Summary

Table 2 summarizes our experiments. The systems from the ATL, MEAD and FTDS traces cover a broad spectrum of latency profiles.

4 Real-World Unpredictability: The ATL Trace

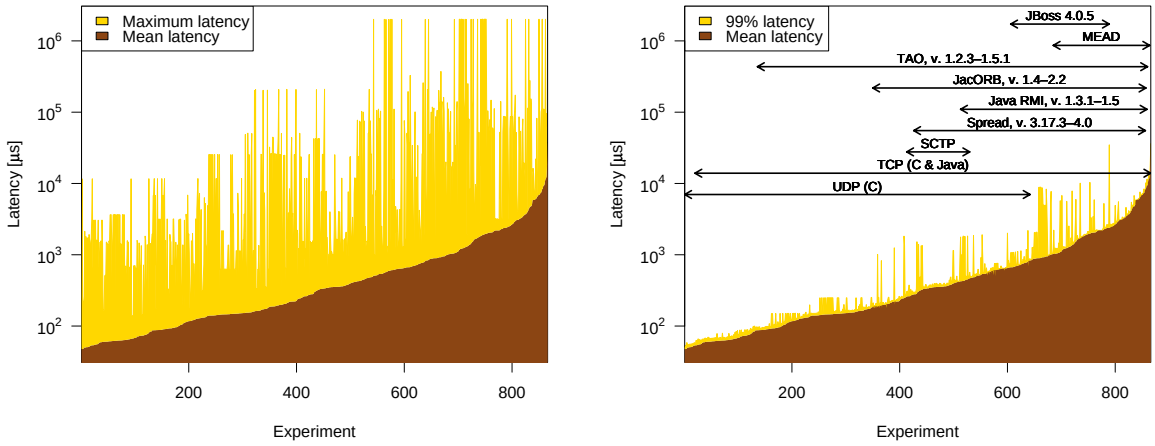
The ATL trace contains observations from 866 configurations of 4 communication protocols and 5 middleware systems. Most experiments focus on the Internet's *transport protocols* [14]—the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP)—which are widely used for sending data between two physical hosts. The Stream Control Transmission Protocol (SCTP) [21] is a newer transport protocol, which can be configured to use two parallel connections between endpoints and to fail-over transparently if one of the redundant connections is lost. Spread [15] is a package of *group-communication protocols*, which enforce extended virtual synchrony (EVS) [26] among two or more physical hosts. This model mandates that the same events (application messages or membership changes) are delivered in the same order at all of the nodes of the distributed system, despite lost messages or node and link crashes. These communication protocols are message-oriented and do not have RPC semantics.

Middleware systems rely on one or several of these protocols for initiating remote invocations (see Figure 1). Java RMI [22], JacORB [23], TAO [24] and MEAD [8] are *distributed-object middleware* systems, which facilitate the development of distributed, object-oriented applications by providing location transparency (the methods of remote and co-located objects are invoked in similar ways). Java RMI allows communicating with remote Java objects, while JacORB, TAO and MEAD implement the Common Object Request Broker Architecture (CORBA), which can connect software components written in different programming languages; for instance, JacORB uses Java, while TAO uses C++. MEAD, described in more detail in Section 5, enhances TAO by transparently providing fault-tolerance to legacy CORBA applications. JBoss [25] implements the Enterprise Java Beans (EJB) architecture for *component middleware*. In this architecture, components implement the business logic of the application, while an application container (*e.g.* JBoss) assembles and configures the final application by deploying its components and by configuring the connections among them. All these systems rely on the TCP protocol, with the exception of TAO—which can be configured to use either TCP or SCTP—and MEAD—which uses Spread.

4.1 Fault-Free Unpredictability

Figure 2a shows the high discrepancy between the mean[6] and maximum latency recorded in the ATL trace. It is hard to find a correlation between these two metrics, especially because the maximum values seem to be randomly high. For most systems from the ATL trace, the maximum latency is 2–3 orders of magnitude

6. The highest mean latency from the ATL trace was recorded for TCP, in the only experiment that used message payloads up to 4 MB.



(a) Before. (b) After.

Figure 2. The "haircut" effect of removing 1% of the end-to-end latency outliers in the ATL trace. The experiments are sorted by the mean latencies recorded, and (b) indicates the range of experiments covering each system.

higher than the mean latency (note the logarithmic scale on the Y-axis of Figure 2a). The maximum latency is unpredictable, according to Criterion ℂ1. After removing 1% of the highest recorded latencies in each experiment, we get the "haircut" effect displayed in Figure 2b: the randomness seems to disappear, and the 99th percentiles do not deviate significantly from the mean values.

Finding I: The maximum latency of communication protocols and middleware systems does not follow a visible trend, and can be three orders of magnitude higher than the mean latency.
Implications: We cannot establish hard bounds for the end-to-end latency of middleware systems, even in the absence of faults.

The difference between the maximum latency and the 99th percentile suggests that the unpredictability is due to a few large outliers. We now investigate how many such outliers are recorded in each experiment (Criterion ℂ2). Figure 3 shows that, in most of the 866 experiments from the ATL trace, fewer than 1% of the recorded latencies are considered outliers, according to the 3σ test. Even fewer requests incur the pathologically-high latencies illustrated in Figure 2. Figure 4 shows that, among all experiments, the relative size of the largest outliers (ℂ1) is negatively correlated with the outlier count (ℂ2).

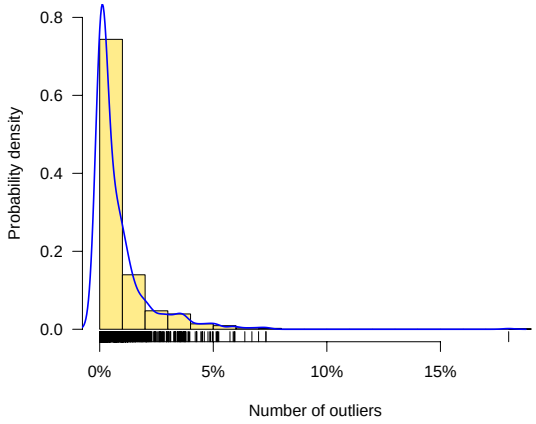


Figure 3. Distribution of outlier counts.

Finding II: The size and the number of outliers are negatively correlated.
Implications: The fault-free unpredictability is caused by a few large-latency outliers.

In order to interpret these findings in the context of the entire ATL trace, we evaluate the impact of the workload and of the environmental conditions on each system's unpredictability (ℂ3). The analysis of variance confirms that the mean latency recorded depends on the system: for UDP, TCP and SCTP, it is on the order of hundreds of microseconds, while for Spread and the 5 middleware systems it is on the order of milliseconds. Moreover, the mean latency depends on the experimental testbed and it increases linearly with the message payload (result significant with $p = 0.001$).

The results are subtly different for the maximum latency, which depends on the system and on the testbed, but not on the message payload. This suggests that, while the systems from the ATL trace do have different latency profiles, their maximum latencies are unpredictable (ℂ3). Similarly, the experimental testbed and the system, but not the payload, have a significant impact on the number of 3σ outliers.

4.2 The Magical 1%

While the maximum latency is unpredictable, we explore the effect of removing just 1% of the highest measured latencies in each configuration. This is equivalent to assessing the predictability of the 99th percentile.

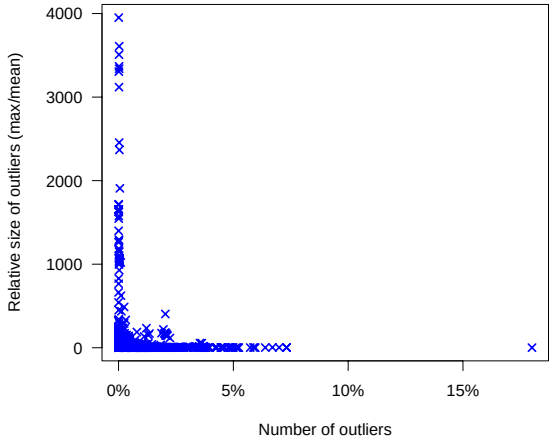


Figure 4. Counts and sizes of outliers.

In the ATL trace, the 99th latency percentile is at most 15× larger than the maximum latency. This indicates that the 99th percentile is predictable according to ℂ1. Moreover, the 99th percentile closely follows the trend of the mean latency, with a correlation coefficient $r = 0.82$. Like the mean latency, the 99th percentile depends on the system and the experimental testbed, and it increases linearly with the message payload. ANOVA identifies these parameters as the main sources of variability, which indicates that the 99th percentile is predictable according to ℂ3. By eliminating a "magical 1%" of the highest recorded latencies, we have removed the fault-free unpredictability from the ATL trace.

Finding III: The 99th percentile of the end-to-end latency is at most one order of magnitude higher than the mean and is correlated with the workload and the environmental conditions.
Implications: The fault-free unpredictability of real-world middleware is limited to 1% of the remote invocations. The 99th latency percentile is predictable.

Additional results are included in Appendix A.2.

5 Sources of Unpredictability: The MEAD Trace

The ATL trace shows that the maximum latency of real-world middleware is unpredictable, for a wide range of systems and environmental conditions, but it does not elucidate the mechanisms that produce this unpredictability. We would like to know, for instance, if all the high-latency outliers have a common cause and if we can eliminate the unpredictability by carefully tuning the system. We therefore focus on a single system, and we investigate whether we can achieve a predictable configuration by employing some of the best open-source components available and by exhaustively exploring the configuration-parameter space. We analyze the unpredictability of the Middleware for Embedded Adaptive Dependability (MEAD), one of the most complex systems evaluated in the ATL trace. Our intimate knowledge of this system, which we have designed and implemented, also allows us to study in-depth the sources of the fault-free unpredictability and of the magical 1%.

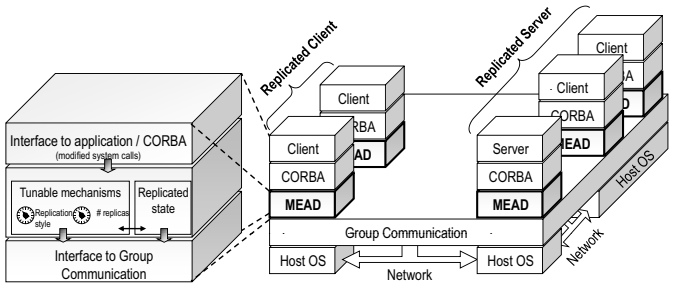


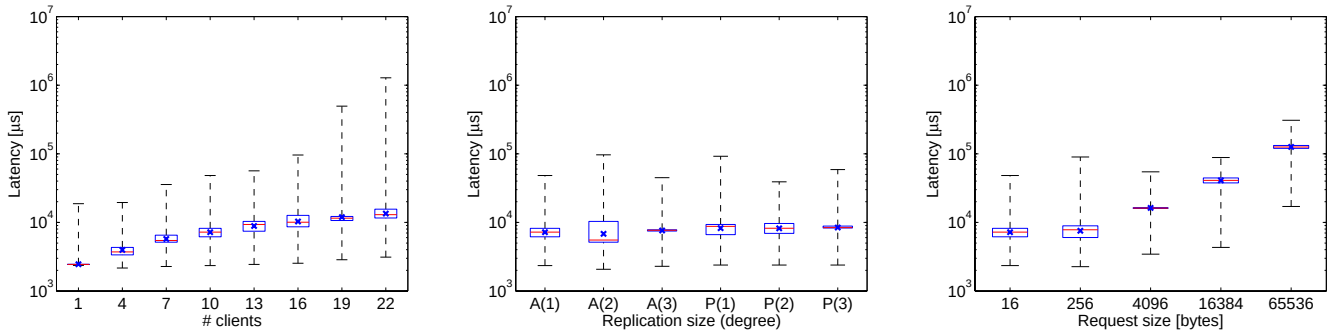
Figure 5. The architecture of MEAD.

The MEAD system [8], illustrated in Figure 5, is an extension of the FT CORBA standard [1]. MEAD provides transparent, tunable fault-tolerance to distributed middleware applications. The system uses library interposition [27] for transparently intercepting and redirecting system calls, and includes a fault-tolerance advisor, whose task is to identify the most appropriate configurations (including the replication style and number of replicas) for the current state of the system. MEAD supports CORBA applications that use the TAO real-time ORB (v. 1.4) [24], which provides excellent mean latency, compared with other middleware systems (see Table 2). MEAD implements both active and passive replication, relying on the Spread (v. 3.17.3) group communication toolkit [15] to enforce the EVS guarantees. We carefully tune Spread's timeouts for our networking environment, to provide fast, reliable fault detection and consistent performance. We configure the Spread daemon with a real-time scheduling policy in order to avoid any unnecessary delays in message delivery.

In the MEAD trace, we collect observations for 1200 configurations, in the absence of faults. We conduct controlled experiments, varying the workload (the payload size, the number of clients, the request rate) and the FT configuration (the replication style and degree). Our micro-benchmark achieves between 20 and 4177 requests/s.

5.1 Configuration Predictability

Figure 6 shows the impact of the configuration parameters on the latency distributions. F or an increasing number of clients (Figure 6a), the minimum latencies are similar, while the average latency, as well as the latency of most of the samples, increases sub-linearly with the number of clients (note the logarithmic Y-axis of the figure). The maximum latency increases as well, but without revealing a clear trend, and it can be two orders of magnitude higher than the medians and means of the corresponding experiments (ℂ1). By varying the replication parameters and the reply size (Figure 6c), we



(a) Number of clients. (b) Replication style and degree. (c) Message payload.

Figure 6. The effect of varying one parameter at a time on the end-to-end latency.

also observe that the maximum latency increases and decreases in an uncorrelated way with respect to the parameter varied. The mean latency, however, increases linearly with the message payload.

The very large latencies are seen for only a few requests. Furthermore, the high-latency outliers seem to come in bursts, which breaks the defenses of many fault-tolerant systems that assume only single or double consecutive timing-faults. This emphasizes that it is impossible to isolate and control the unpredictability by adjusting the parameters of the system configuration.

The experiments with 64 KB payloads produce half of the outliers recorded in the MEAD trace (C2), but the size of these outliers is relatively small when compared to the other cases. As in the ATL trace (Finding II), the occurrence of very large outliers is negatively correlated with a high number of outliers.

These results indicate that the replication style, the replication degree, the number of clients, the request rates and, up to 16KB, the message payload, do not influence the number of outliers produced in the MEAD trace. Increasing the message size to 64 KB introduces large numbers of outliers, perhaps due to the higher amount of work performed inside the operating-system kernel for segmenting and re-assembling these large messages. In general, however, we cannot obtain bounded maximum latencies by calibrating these configuration parameters.

We try to determine if the unpredictability recorded in our experiments can be correlated with the behavior of an operating-system mechanism by comparing number and size of outliers with various server-side resource-usage statistics. The server processes are never swapped out of physical memory and that they generate between 1675 and 1680 major page-faults (requiring a memory page to be reloaded from disk), because the experiments are conducted in isolation and the benchmark is very repetitive in nature.

In the experiments with 16 KB and 64 KB message payloads, both the client and the server spend around 25% of the time in kernel mode, compared to 10% for the other cases. However, these execution times do not indicate the occurrence of unpredictability, as unusually large numbers of outliers occur only for the 64 KB case. The minor page-fault rate increases with the number of clients, and the resident set grows with the payload size, because of the need to allocate larger memory buffers. This predictable behavior contrasts the unpredictability of the end-to-end response times, suggesting that the virtual memory system is not the source of the overall recorded unpredictability.

The size and number of outliers seems to be inversely proportional to the average number of context switches on the server hosts. Moreover, the clients exhibit the same trend. The occurrence of context switches can be explained by the regular OS daemons running on each host (e.g., `sshd`), as well as a few network daemons specific to the Emulab testbed and the daemon of the Spread group communication system.[7] A potential explanation of the negative correlation between context switches and outliers is that the normal operation mode of our system is characterized by a large number of context switches between the program and the Spread daemon, and that fewer context switches indicate that one of these processes is blocked (*i.e.*, waiting), which generates the outliers.

Finding IV: The maximum latency cannot be regulated by adjusting the number of clients, the request rates or the replication style and degree, and it is not correlated with system-level metrics such as the amount of time spent in kernel mode, the number of page faults or the size of the resident set.
Implications: Unpredictability cannot be eliminated by carefully tuning the system.

5.2 Sources of Fault-Free Unpredictability

We also examine whether a single module of our system might be responsible for producing all these outliers.

7. The Spread daemon performs many computationally-intensive operations in order to enforce the extended virtual synchrony of the distributed system, using a significant amount of CPU time.

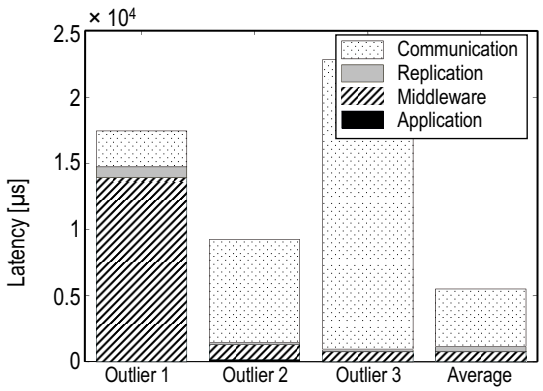


Figure 7. The outliers might originate in any layer.

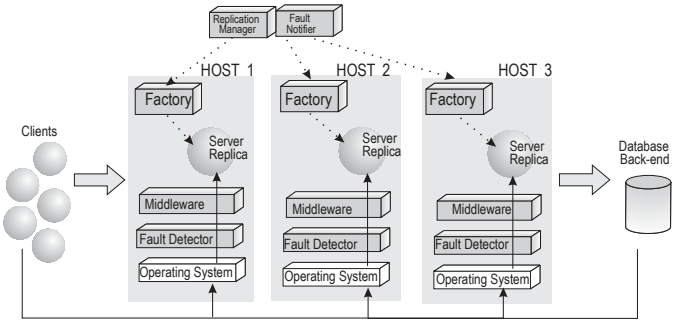


Figure 8. Architecture of the FTDS applications.

The experimental harness in the MEAD trace is composed of a simple CORBA application (under 100 source lines-of-code), an object request broker (middleware), a replication module and a group communication protocol (Figure 1). Figure 7 presents the breakdown of the end-to-end latency by components for some of the outliers observed, as well as for the average latency. We can see that Outlier 1 originated in the ORB, while the source of Outlier 3 is the group communication module (which may be due to either the delay in physical network medium or to the synchronization overhead for enforcing the extended virtual synchrony guarantees). For Outlier 2, the application took 10 times longer than on average to process the request; although this latency is small, this is another example of a rare event that may result in an outlier. These effects are unlikely to be caused by the operating system's scheduler, as the experiments were conducted in isolation, where each client and server replica had a dedicated host and there were no other processes competing for the operating system's resources.

Finding V: The latency outliers have multiple root causes.
Implications: The maximum latency is inherently unpredictable.

Additional results are included in Appendix A.3.

6 Impact of Design and Implementation: The FTDS Trace

The ATL and MEAD traces illustrate the unpredictability of middleware and FT middleware systems, but these experiments were conducted using micro-benchmarks with response times on the order of milliseconds. In a complete enterprise application, the latency due to business-logic operations and to storing the persistent data in a back-end database might dominate the end-to-end latency and might mask the unpredictability of the middleware. Moreover, the systems from the ATL and MEAD traces are fully operational and do not allow us to evaluate the impact of the design choices *ex post facto*.

The FTDS trace contains observations from 336 configurations of 7 realistic enterprise applications, developed during the Spring 2006 (January-May 2006) semester by the students enrolled in the "Fault-Tolerant Distributed Systems" (FTDS) graduate class at Carnegie Mellon University. These applications are similar in scope and complexity to other benchmarks widely used for evaluating middleware systems, such as Pet Store [28], TPC-W [29] or RUBiS [30], but they cover a wider range of behaviors.

The common architecture of the applications is modeled after the FT CORBA standard [1]. The 7 applications are described in Table 3. Each application relies on a middleware platform, either EJB (projects 1–5) or CORBA (projects 6 and 7). As shown in Figure 8, the clients connect to a server (middle tier), which performs all the business-logic processing and uses a MySQL database in the back-end to store all of the critical state. Effectively, the middle-tier servers are stateless, which simplifies their checkpointing and recovery. The middle tier is replicated for fault-tolerance, using warm-passive replication (projects 1–6) or active replication (project 7). 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 fail-over after a fault and relaunches 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 to obtain references to the server objects during initialization or during crash-fault recovery. The application designs assume that the Replication Manager, the Naming Service and the database never fail. The maximum request rates for the 7 applications are between 24–1250 requests/s.

6.1 Application Level Unpredictability

This design and implementation processes produced 7 applications that exhibit significant variability among the latency measurements, with ranges up to 200 ms. The FTDS applications have widely different latency profiles: Park'n Park achieves the highest request rates and the

Table 3
Characteristics of the 7 applications from the FTDS Trace.

Project (developers)	Replication style	Description	Message payloads requests / replies
1: Su-Duel-Ku (5)	Passive	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. Owing to its original idea, the project was mentioned in a local newspaper [31].	4 b / ≈200 b
2: Blackjack (5)	Passive	Gaming application, where users play Blackjack online. Users can create online profiles (stored in the database), place bets and play against the house.	≈30 b / ≈56 b
3: FTEX (5)	Passive	Electronic stock exchange (*e.g.* NASDAQ). Users 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.	≈30 b / ≈50 b
4: eJBay (6)	Passive	Distributed auctioning system, similar to eBay. Allows posting items for sale and bidding for them; the user profiles and the information related to auctions are stored in the database.	116 b / 98 b
5: Mafia (4)	Passive	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.	≈41 b / 4 b
6: Park'n Park (5)	Passive	Application for managing parking lots. Keeps track of how many spaces are available in the lots and recommends alternative locations when a lot is full.	3 b / 4 b
7: Ticket Center (5)	Active	Online ticketing application for express buses, allowing users to search schedules and available seats, buy and cancel tickets and check reservation status.	≈16 b / 4 b

lowest average latencies, Su-Duel-Ku has the highest number of outliers, and eJBay, has both the lowest number of outliers (0.33%) and longest-running request (190 s). Four applications, Blackjack, FTEX, eJBay and Ticket Center, exhibit maximum latencies two orders of magnitude higher than the average, and one application (eJBay) produces outliers three orders of magnitude higher that the average. Project 4 is the only one that exhibits a significant correlation ($r = 0.8$) between the incoming request rates and the number of outliers. The message payload and the number of clients do not have an impact on the outliers; maximum sizes of outliers are comparable for all the values of these parameters.

Figure 9 compares the outlier distributions (ℂ2) for the seven FTDS applications. It is interesting to note that applications eJBay and Ticket Center, which have nearly identical outlier distributions, are based on radically different technologies: one is an EJB application using warm-passive replication, and the other is a CORBA application using active replication. Su-Duel-Ku is the only application from the FTDS trace that has predictable (ℂ1) response times, as the maximum latency follows the same trend as the mean latency.

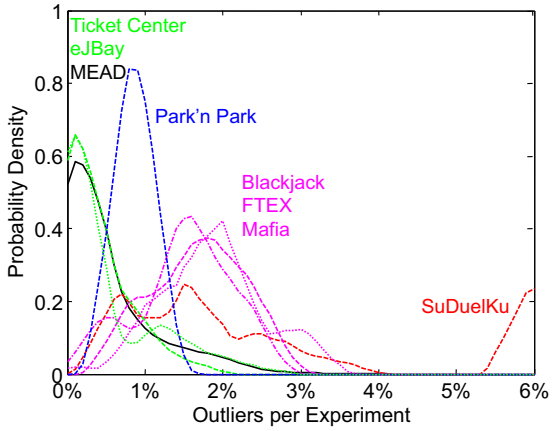


Figure 9. Distributions of outlier counts.

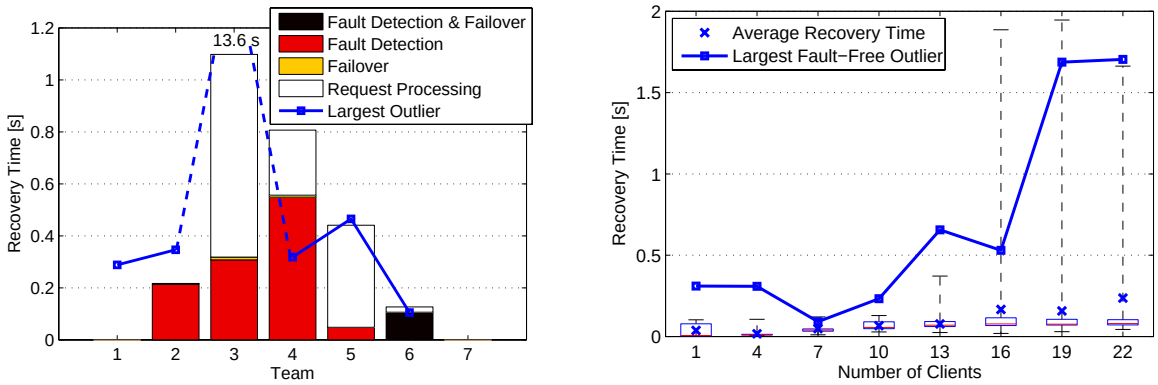
Finding VI: The latency of middleware-based applications exhibits considerable variability, and infrastructure-level design choices, such as the middleware platform or the replication style, do not account for the fault-free unpredictability.
Implications: It is unlikely that the fault-free unpredictability can be eliminated by focusing only on the middleware infrastructure.

With one exception (Park'n Park), all the applications produce outliers due to either the latency of contacting the database or to the processing performed withing the FT middleware. The component responsible for most of the outliers varies among the seven applications: for Su-Duel-Ku and Blackjack most outliers originate in the interactions with the database, while for FTEX the majority of outliers originate in the middleware. For Park'n Park, all the outliers recorded originate in the middleware. This suggests that the application-level unpredictability confirms Finding V.

Additional results are included in Appendix A.4.

6.2 Comparison with Fault-Induced Unpredictability

We have shown that FT middleware systems have unpredictable response times even in the absence of faults. We now compare these random high latencies occurring during the normal operation mode with time needed to recover from a crash fault.



(a) Recovery time for FTDS trace.

(b) Recovery time for MEAD trace.

Figure 10. Comparison between recovery-time after a crash fault and fault-free outliers.

The developers of the FTDS applications conducted fault-injection experiments, by provoking 10–20 crash faults in the middle tier while 1 client was connected. Based on preliminary observations, the 7 teams optimized their recovery time by maintaining object references to all the server replicas and keeping open TCP connections to these replicas in order to avoid time-consuming name lookups and the overhead of connection establishment during the fail-over process. In Figure 10a, we plot the recovery times of the FTDS applications after this optimization stage; each bar represents the average round-trip time of the requests issued when faults are injected. We break down the recovery time into components corresponding to fault-detection, fail-over and normal request processing.[8]

The largest contributor to the recovery time in these applications is the delay introduced by fault detection. The fault-induced outliers are significantly higher for project 4 (eJBay), and they are comparable with the fault-free outliers for projects 2 (Blackjack), 5 (Mafia) and 6 (Park'n Park). Project 3, however, has recorded 453 outliers larger than its recovery time; the largest such outlier (13.6 s) is one order of magnitude higher than the recovery time. This indicates that, under certain circumstances, high latencies occurring randomly during normal operation may have a higher impact on availability than the equally-infrequent hardware faults.

We also compare these results with the recovery times for MEAD, running in active replication mode with three-way replication and 4 KB reply messages (Figure 10b). Every 10 seconds, we induce a crash-fault, and we subsequently restart the crashed server replica, injecting up to 35 faults during an experiment. The recovery time seems to increase with the number of clients, but, for more than 16 clients, we observe a higher variability and some large recovery-time outliers. We can also see that the outliers recorded during the fault-free experiments with corresponding configurations are comparable with the recovery times.

Finding VII: For enterprise applications with stateless middle tiers, the maximum fault-free latency is often comparable with the time needed to recover from crash faults.
Implications: It is worth reexamining the cost/benefit trade-off of optimizing the fail-over process for low recovery times, as comparable outages are expected to occur during normal operation.

The low recovery times achieved by these applications are due to the stateless nature of the replicated servers, which enables optimizations of the fail-over process. Enterprise three-tier systems usually store volatile state, such as sessions or cached content, in the middle tiers, while keeping their persistent objects in a database. Because volatile state can be recreated after a fault and does not need to be synchronized, the low recovery times reported here are realistic.

7 Summary of Results

The latency of middleware systems is influenced by many factors, such as the environmental conditions, the workload, the functional characteristics of the application (*i.e.*, the business logic), the middleware infrastructure, the fault-tolerance mechanisms, the configuration parameters, etc. With two exceptions—SCTP, from the ATL trace, and Su-Duel-Ku, from the FTDS trace—the maximum latency is unpredictable in all the configurations analyzed in our 16-system study. Figure 11 compares these maximum latencies with the confidence intervals of the 99th percentile latency, for all the systems evaluated in this paper. The results are strikingly similar across the ATL, MEAD and FTDS traces:[9] *with a*

8. Because this phase of the project was designed as an opportunity to obtain bonus points, these results are incomplete: teams 1 (Su-Duel-Ku) and 7 (Ticket Center) did perform fault injection, while team 6 (Park'n Park) lumped together the fault-detection and fail-over times.

9. We exclude Project 5 (Mafia) because we could not compute the confidence intervals due to the small sample-sizes (100 invocations) reported.

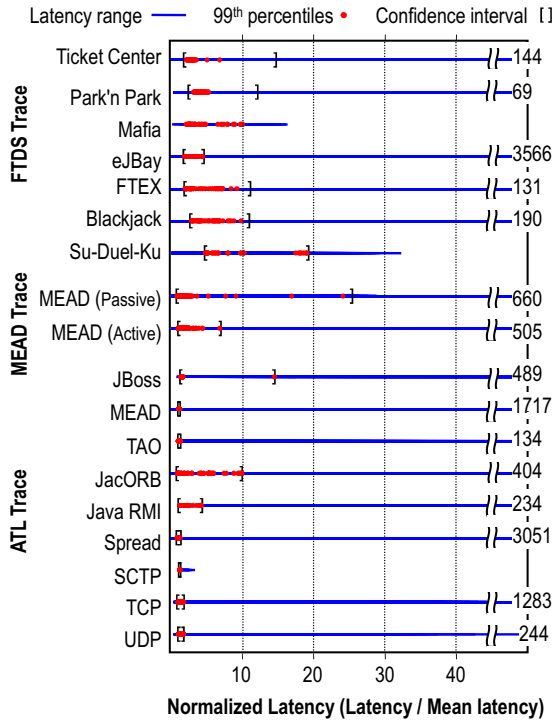


Figure 11. Bounds for the 99th percentile latency.

statistical confidence level of 99% (i.e. $p < 0.01$), *the 99th percentile of the latency is at most* 25× *larger than then mean latency.* This shows that only 1% of the requests issued in each configuration are responsible for the unpredictable behavior. By removing this magical 1% we eliminate the pathologically large end-to-end response times that render the maximum latency unpredictable.

8 Discussion

Computer systems are not naturally occurring objects. They are designed and built by human engineers, and it is difficult to accept that their behavior cannot be explained by misuse, defects or designer intent. However, academic and industry experts forecast that the future ultra-large-scale systems will be characterized by *emergent behaviors*, which are not localized to any component and are difficult to predict using our current techniques [32]. Such emergent behaviors are best studied empirically, much in the way we study the natural sciences. The magical 1% is an example of emergent behavior in middleware systems, and our broad empirical study allows us to answer a number of open questions.

Can we reconcile fault-tolerance and real-time requirements? Nearly a decade ago, we postulated the existence of a fundamental trade-off between the goals of fault-tolerance (which aims for predictable recovery from faults) and of real-time (which aims for end-to-end temporal predictability) [33]. For instance, these separate goals require different orders of operations, and the consistency semantics of the data might need to be traded against timeliness during the composition of fault-tolerance and real-time. This insight motivated the initial design of the MEAD system, which enables fine-grained tuning of these system-level properties.

Comparing the ATL, MEAD and FTDS traces sheds new light on this trade-off. While fault-tolerant systems have a higher mean latency, needed to ensure agreement among multiple hosts, the maximum latency is not necessarily more unpredictable than for the other systems. However, the agreement phase, which typically dominates the end-to-end latency, can hide the unpredictability of the other layers. For example, the group communication protocol accounts for most of the outliers observed in the MEAD trace. This only happens for high replication degrees; for instance, for some applications from the FTDS trace (which use two-way replication) the main source of unpredictability is the communication with the back-end database. This suggests that fault-tolerance mechanisms do not always diminish temporal predictability, but they might shift the leading source of latency outliers to a different system component.

Are enterprise applications more predictable than their building blocks? The communication protocols and middleware systems from the ATL and MEAD traces tend to produce larger outliers than the applications from the FTDS trace (except for project 4, eJBay). This suggests that, in a complete enterprise application, the computationally-intensive business logic and the back-end database can partially mask the unpredictability of the communication protocols and of the middleware. Even so, the maximum latency of these applications might be 2–3 orders of magnitude higher than the average. This unpredictability cannot be eliminated by carefully configuring the system, and the only effective technique for obtaining a predictable latency profile is to remove 1% of the highest latencies.

Can we achieve strict predictability? Our results might seem surprising given the fact that many embedded systems, designed from the ground up, can successfully enforce hard real-time properties. Therefore, our empirical study encompasses all the layers of the middleware stack—examined both in isolation and as parts of a complex system—and tries to pinpoint the sources of unpredictability in middleware systems. In both MEAD and FTDS traces we have observed that the typical source of outliers is system-dependent and that outliers might originate in any middleware layer. While the group communication protocol produces most of MEAD's outliers, the middleware and the replication mechanisms occasionally induce abnormally high latencies as well. Furthermore, the unpredictability is not confined to a single tier of a distributed application. In the FTDS trace, the typical source of outliers is either the database, for projects 1 and 2, or the middle tier, for projects 3 and 6; the other projects produce outliers that equally originate in both tiers.

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. Our broad empirical study shows that unpredictable maximum latencies are part of the normal behavior for a diverse group of middleware systems. More research is needed to eliminate this inherent unpredictability, but it seems that, presently, fault-tolerant middleware must cope with such unbounded behavior, which comes in addition to the unpredictability related to the potential occurrence of faults.

How can we design systems that cope with the inherent unpredictability? Existing approaches for the autonomic management of computer systems [2–7] focus on predicting the average system behavior. Furthermore, the existence of the magical 1% suggests that *statistical predictability* is easily achievable in middleware-based systems. When establishing service-level agreements (SLAs), providers must make informed business decisions, such as how many new clients they can admit (before compromising the existing quality of the service), or what class of service they can reliably deliver (based on the observed load on the system). We have shown that latency percentiles can be predicted with high confidence, which enables service-class guarantees based on such statistical measures. For instance, Amazon.com provides service-level agreements that focus on the 99.9^{th} latency percentile [34]. Our research provides a solid scientific foundation for the engineering choices made in such Internet-scale infrastructures, which guarantee percentile-based performance metrics.

This is the magic of the unruly 1%.

What are the limitations of statistical predictability? The magical 1% hypothesis holds true for different operating systems (Linux 2.4, Linux 2.6, Linux-rt, TimeSys), middleware platforms (CORBA and EJB), programming languages (C, C++ and Java), replication styles (active and warm passive) and applications. We have tested this hypothesis in clusters connected with a local-area network, which is the typical setting for fault-tolerant middleware systems. Our rule of thumb may not be effective in environments with high propagation delays, such as wide-area networks, or with intermittent connectivity, such as wireless networks. Moreover, certain applications (*e.g.*, embedded real-time systems) will not be able to rely on percentiles; in such cases, nothing short of predictable worst-case behavior will be sufficient.

9 Related Work

Anecdotal evidence and recent studies [16, 17] suggest that the maximum end-to-end latencies of CORBA and FT-CORBA middleware are usually several orders of magnitude larger than the average latency and do not follow a visible trend. However, most of these findings have not been validated with extensive experimental or field data, covering multiple systems and middleware platforms. Fault-tolerant middleware, such as MEAD, are often based on the extended virtual synchrony model [26]. This model mandates that the same events are *eventually* delivered in the same order at all of the nodes of the distributed system, but without enforcing any timeliness guarantees. Thomopoulos *et al.* [35] introduce a stochastic model for latency under extended virtual synchrony, which predicts long-tailed probability distributions for the latency of "safe" messages needed for implementing replication. Gutierrez-Nolasco *et al.* [36] present a formal model of the Spread protocols that MEAD uses for preserving the EVS guarantees; however, the model is focused on correctness, rather than performance predictions. Szentivanyi [37] presents a performance evaluation of FT middleware and suggests techniques for improving this performance.

Because of the inherent unpredictability, existing approaches for the autonomic management of computer systems [2–7], including both commercial products and research prototypes, focus on predicting and tuning the average system behavior. Aguilera *et al.* focus on debugging the average performance of large-scale distributed systems by identifying the node-activity patterns that have a high impact on the mean latency [2]. Narayanan *et al.* describe a Resource Advisor for SQL Server 2005, which can predict the effect of changes in available buffer-memory or transaction rates on the throughput and on the average latency [3]. We proposed an approach called versatile dependability, which provides knobs for tuning system-level properties rather than internal fault-tolerance mechanisms. For instance, we demonstrated a knob that tunes MEAD's scalability, while enforcing predefined bounds on the mean latency [4]. Because of the heterogeneity of COTS-based distributed systems, Mansour *et al.* [5] suggest eliminating the performance dependencies between sub-components by introducing isolation points that limit and contain the effects of ill-behaving requests, in order to improve the overall system predictability. Thereska *et al.* present a framework for managing cluster-based storage systems with facilities for predicting the impact of data placement and encoding choices on throughput and mean response-time [6]. Dageville *et al.* present the architecture of Oracle's self-tuning solutions, aimed at tuning SQL statements and at improving the overall throughput of the database [7].

10 Conclusions

In this paper, we examine the predictability of 16 middleware systems and communication protocols. By exhaustively exploring 2402 configurations of fault-tolerant middleware systems, we show that unpredictability, manifesting as unbounded maximum latencies, cannot usually be eliminated by selecting a good system configuration. The end-to-end latencies have skewed distributions, with maximum values several orders of magnitude larger than the averages. The number of clients, the

replication style and degree or the request rates neither inhibit nor augment the occurrence of latency outliers. Middleware-based systems produce such outliers even in the absence of faults, and the magnitude is comparable with the time needed to recover after a crash fault.

We also present strong empirical evidence of a "magical 1%" effect: after removing 1% of the highest measured latencies for each configuration, the resulting latency profile becomes predictable. Moreover, the magical 1% is the only effective rule for isolating the inherent unpredictability of the systems examined. We show that the 99^{th} percentile of the end-to-end latency closely follows the trend of the mean and that it can be bounded with a high statistical confidence. While such percentile-based guarantees are clearly inappropriate for hard real-time systems, they can be of immense benefit to enterprise applications and service providers—which do not focus on the worst-case behavior of the system but require quantifiable assurances in the normal operation mode.

ACKNOWLEDGMENTS

The authors would like to thank Gautam Thaker for allowing us to analyze the data from the ATL trace. Thanks are also due to David O'Hallaron, Asit Dan, Daniela Roşu, Jay Wylie, Arun Iyengar, Jean-Charles Fabre, and the 35 students from the Spring 2006 installment of the Fault-Tolerant Distributed Systems course.

REFERENCES

[1] Object Management Group, "Fault Tolerant CORBA," Sep 2001, OMG Technical Committee Document formal/2001-09-29.

[2] M. K. Aguilera *et al.*, "Performance debugging for distributed systems of black boxes," in *Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct 2003, pp. 74–89.

[3] D. Narayanan, E. Thereska, and A. Ailamaki, "Continuous resource monitoring for self-predicting DBMS," in *Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Atlanta, GA, Sep 2005.

[4] T. Dumitraş, D. Srivastava, and P. Narasimhan, "Architecting and implementing versatile dependability," in *Architecting Dependable Systems III*, R. de Lemos, C. Gacek, and A. Romanovsky, Eds. Springer-Verlag, LNCS 3549, 2005, pp. 212–231.

[5] M. Mansour and K. Schwan, "I-RMI: Performance isolation in service oriented architectures," in *ACM/IEEE/IFIP Middleware Conference*, Grenoble, France, Nov/Dec 2005.

[6] E. Thereska *et al.*, "Informed data distribution selection in a self-predicting storage system," in *International Conference on Autonomic Computing*, Dublin, Ireland, Jun 2006.

[7] B. Dageville and K. Dias, "Oracle's self-tuning architecture and solutions," *IEEE Data Engineering Bulletin*, vol. 29, no. 3, pp. 24–32, Sep 2006.

[8] P. Narasimhan, T. Dumitraş *et al.*, "MEAD: Support for real-time, fault-tolerant CORBA," *Concurrency and Computation: Practice and Experience*, vol. 17, pp. 1527–1545, 2005.

[9] G. Leibniz, *Discourse on Metaphysics*, 1686.

[10] S. Vinoski, "An overview of middleware," *Lecture Notes in Computer Science*, vol. 3063, pp. 35–51, Jan 2004.

[11] P. Felber and P. Narasimhan, "Experiences, approaches and challenges in building fault-tolerant CORBA systems," *IEEE Transactions on Computers*, vol. 54, no. 5, pp. 497–511, 2004.

[12] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[13] N. Budhiraja, F. Schneider, S. Toueg, and K. Marzullo, "The primary-backup approach," in *Distributed Systems*, S. Mullender, Ed. ACM Press - Addison Wesley, 1993, pp. 199–216.

[14] W. R. Stevens, *UNIX Network Programming*, 3rd ed. Addison-Wesley, 2003.

[15] Y. Amir, C. Danilov, and J. Stanton, "A low latency, loss tolerant architecture and protocol for wide area group communication," in *International Conference on Dependable Systems and Networks*, New York, NY, Jun 2000, pp. 327–336.

[16] A. S. Krishna *et al.*, "CCMPerf: A benchmarking tool for CORBA Component Model implementations," *The International Journal of Time-Critical Computing Systems*, vol. 29, no. 2–3, Mar 2005.

[17] W. Zhao, L. E. Moser, and P. M. Melliar-Smith, "End-to-end latency of a fault-tolerant CORBA infrastructure," *Performance Evaluation*, vol. 63, no. 4, pp. 341–363, 2006.

[18] National Institute of Standards and Technology, "Engineering statistics handbook," http://www.itl.nist.gov/div898/handbook/index.htm.

[19] T. Dumitraş and P. Narasimhan, "Fault-tolerant middleware and the magical 1%," in *ACM/IEEE/IFIP Middleware Conference*, Grenoble, France, Nov/Dec 2005, pp. 431–441.

[20] ——, "Got predictability? Experiences with fault-tolerant middleware," in *ACM/IEEE/IFIP Middleware Conference*, Newport Beach, CA, Nov 2007.

[21] R. Stewart and C. Metz, "SCTP: New transport protocol for TCP/IP," *Internet Computing, IEEE*, vol. 5, no. 6, pp. 64–69, Nov/Dec 2001.

[22] Sun Microsystems, "Java remote method invocation specification," 2004. [Online]. Available: http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf

[23] G. Brose, "JacORB: Implementation and design of a Java ORB," in *Distributed Applications and Interoperable Systems*, Cottbus, Germany, Sep/Oct 1997, pp. 143–154.

[24] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time Object Request Broker," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr 1998.

[25] M. Fleury and F. Reverbel, "The JBoss extensible server," in *ACM/IEEE/IFIP Middleware Conference*, Rio de Janeiro, Brazil, Jun 2003, pp. 344–373.

[26] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal, "Extended virtual synchrony," in *IEEE International Conference on Distributed Computing Systems*, Tokyo, Japan, 1994, pp. 56–65.

[27] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann Publishers, 2000.

[28] I. Singh, B. Stearns, M. Johnson *et al.*, *Designing Enterprise Applications: Java 2 Platform*, 2nd ed., 2002, http://java.sun.com/blueprints/guidelines/.

[29] D. Menascé, "TPC-W: A benchmark for e-commerce," *IEEE Internet Computing*, vol. 6, no. 3, pp. 83–87, May/Jun 2002.

[30] C. Amza *et al.*, "Specification and implementation of dynamic web site benchmarks," in *IEEE Workshop on Workload Characterization*, Austin, TX, Nov 2002, pp. 3–13, http://rubis.objectweb.org/.

[31] R. Hentges, "Puzzling: Sodoku has grabbed the short-attention span of a nation," *Pittsburgh Tribune Review*, Apr 2006, http://www.pittsburghlive.com/x/pittsburghtrib/search/s_447266.html.

[32] L. Northrop *et al.*, *Ultra-Large-Scale Systems: The Software Challenge of the Future*. SEI Carnegie Mellon University, Jun 2006.

[33] P. Narasimhan, "Trade-offs between real-time and fault tolerance for middleware applications," in *Workshop on Foundations of Middleware Technologies*, Irvine, CA, Nov 2002.

[34] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Symposium on Operating Systems Principles*, Stevenson, WA, Oct 2007, pp. 205–220.

[35] E. Thomopoulos, L. E. Moser, and P. M. Melliar-Smith, "Latency analysis of the Totem single-ring protocol," *IEEE/ACM Transactions on Networking*, vol. 9, no. 5, pp. 669–680, Oct 2001.

[36] S. Gutierrez-Nolasco *et al.*, "Exploring adaptability of secure group communication using formal prototyping techniques," in *Workshop on Reflective and Adaptive Middleware*, Toronto, ON, Oct 2004.

[37] D. Szentivanyi, "Performance studies of fault-tolerant middleware," Ph.D. dissertation, University of Linköping, 2005.

[38] B. White *et al.*, "An integrated experimental environment for distributed systems and networks," in *USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec 2002, pp. 255–270.

[39] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

APPENDIX A
SUPPLEMENTAL MATERIAL

A.1 Experimental Methods

The differences between the ATL, MEAD and FTDS traces are summarized in Table 4. To prevent a systematic bias, the systems from the three traces were instrumented independently and data was collected by different experimenters. We were directly involved in the data collection only for the MEAD trace. To minimize the interference from the experimental harness, we preallocate buffers in memory to store the probe data, and we flush these buffers to the disk only at the end of each experiment. Moreover, the client-side probes do not affect the latency observations because timestamps are recorded before a request is issued and after a reply is received. Except for Sections 5.2 and 6, all the data presented in this paper represents client-side measurements.

We analyze 2402 different configurations of middleware systems and communication protocols. In each configuration we make N_{conf} measurements of the response times $R_{conf}(i)$, with $i \in [1, N_{conf}]$. We compute the mean latency $\overline{R_{conf}}$, the standard deviation σ_{conf} and the following statistical measures:

$$\text{Probability density} \quad \int_s^t \text{PDF}(x)\,\mathrm{d}x = \Pr[s \le R_{conf} \le t]$$

$$99^{\text{th}} \text{ percentile} \quad \tilde{R}_{conf} : \int_0^{\tilde{R}_{conf}} \text{PDF}(x)\,\mathrm{d}x = 0.99$$

The raw latency measurements for the ATL trace, collected between 2001–2008, are no longer available. Instead, we analyze the histograms of these measurements, which split the range of the observed latency into equally-sized bins and provide the counts of measurements that fall into each bin. $\max R_{conf}$, $\overline{R_{conf}}$ and σ_{conf} have been computed originally, from the raw data, and the histograms represent an estimation of the probability density function. We estimate $\tilde{R}_{conf}$ through linear interpolation in the bin of the 99th percentile. Unfortunately, this estimation is imprecise for some configurations with wide histogram bins; for instance, in some cases all the measurements fall into a single bin, and linear interpolation does not provide meaningful results because the characteristics of the original distribution are lost. To ensure the precision of estimation for the 99th percentile, we disregard all the configurations where the 99th percentile falls into the same bin as either the 75th percentile or the maximum of R_{conf}. Furthermore, we exclude from the ATL data the fault-injection experiments, the configurations with additional processor or network loads, and the large number of experiments that were performed on a single physical host. For the MEAD and FTDS traces, we perform the statistical analysis directly on the raw measurement data.

Criterion $\mathbb{C}$1. Because the 16 systems have different latency ranges, we assess whether the latency is too large by normalizing the latency values: $\hat{R}_{conf}(i) = R_{conf}(i)/\overline{R_{conf}}$. This metric indicates whether all the observed latencies are on the same order or magnitude, which, in practice, would mean that the system configuration examined is sufficiently predictable because it doesn't produce excessively large latencies. We therefore consider that a system is unpredictable when we record latencies that are more than one order of magnitude larger than the mean latency: $\exists R_{conf}(i) : \hat{R}_{conf}(i) > 10$. We note that this is a pragmatic test, rather than a statistically-rigorous one.

Criterion $\mathbb{C}$2. We estimate the number of requests with an unusually-high latency using the 3σ statistical test: any observation that deviates from the mean with more than 3σ is considered an *outlier*. 3σ is a statistical test widely used in the engineering disciplines for quality-control or for identifying measurement errors [18]. The 3σ test helps us detect latency values.

Criterion $\mathbb{C}$3. We evaluate the impact of a configuration parameter on the latency by performing an *analysis of variance* (ANOVA) [18]. ANOVA summarizes how much of the variance in the data is accounted for by the impact of the configuration parameter and how much is random error. Formally, ANOVA tests the *null hypothesis* that these two variability components are estimates of the true variance. If the probability p of this hypothesis is small (*e.g.*, $p < 0.01$), there is statistically-significant evidence that the values of the configuration parameter influence the latency.

Impact of environmental conditions. We assess the impact of the environment on the system unpredictability by conducting experiments in a wide variety of testbeds. Most experiments from the ATL trace use the Emulab testbed [38], in two hardware configurations:

- `pc850`: Pentium III, running at 850 MHz, with 256 MB of RAM and 100 Mbps LAN.
- `pc3000`: 64-bit, dual-core Xeon, running at 3.0 GHz, with 2 GB of RAM and 1 Gbps LAN.

The ATL trace uses six additional testbeds. Moreover, the ATL experiments employ several operating systems: Linux 2.4.2–2.4.20, Linux 2.6.8–2.6.23, TimeSys 3.1. TimeSys is a Linux-based, commercial operating system, with a fully-preemptible kernel, protection against priority inversion, $O(1)$ task-scheduling complexity, and a fine timer granularity. In some configurations, the Linux-rt patches (rt1–rt17) are applied to the Linux kernel, in order to enable preemptibility.

In the MEAD trace, we also use Emulab `pc850` nodes, with TimeSys 3.1. Our experiments use 25 hosts: up to 22 for the clients and up to 3 for the replicated server. The experiments from the FTDS trace utilize a university cluster, where the experimental nodes were connected by a 100 Mbps LAN. Each machine has a dual-processor Pentium 4 at 2.8 GHz with 2GB memory, and runs SUSE Linux (kernel 2.6). Each entity in the system (clients, servers, database) uses a dedicated physical node.

Table 4
Qualitative comparison of the three experimental traces.

	ATL	MEAD	FTDS
Benchmark	Micro-benchmark	Micro-benchmark	Macro-benchmark
Fault tolerance	None (most systems), single link-failures (SCTP, TAO/SCTP), node failures (MEAD)	Link and node failures, with no single point of failure	Node failures, with single point of failure (the Replication Manager)
Programming language	C, C++, Java	C++	Java
Middleware platform	Message-oriented protocols, CORBA, EJB	CORBA	CORBA, EJB
Operating system	TimeSys 3.1, Linux 2.4.2–2.4.20, Linux 2.6.8–2.6.23 (including Linux-rt)	TimeSys 3.1	SUSE Linux 2.6
Data collection	Third party	Authors	Under the authors' supervision

Impact of FT configurations. We assess the effect of two configuration options for the FT mechanisms: the *replication style* (MEAD and FTDS traces) and the *replication degree* (MEAD trace). We vary the replication style as follows:

- MEAD trace: for MEAD, the replication style is a configurable parameter.[10] We test each system configuration using both active and passive replication.
- FTDS trace: the replication style is a static design choice in all 7 applications. 6 applications use passive replication, and 1 uses active replication.

The replication degree indicates the number of server replicas, including the primary for warm passive replication. A system with a replication degree of n will be able to tolerate $n-1$ crash faults without interruption. We use the following replication degrees:

- MEAD trace: 1, 2 or 3 server replicas.
- FTDS trace: 2 server replicas (middle tier).

Impact of workloads. We assess the impact of the workload by varying the *message payload* (ATL, MEAD and FTDS traces), the *number of clients* (MEAD and FTDS trace) and the *request rate* (MEAD and FTDS traces). The three traces cover the following payloads:

- ATL trace: $2^2, 2^3, \ldots 2^{16}$ bytes (*i.e.*, up to 64 KB), for most experiments; up to 4 MB for one experiment (TCP); only up to 1 KB for other experiments (*e.g.*, SCTP, TAO/SCTP).
- MEAD trace: 16 bytes, 256 bytes, 4 KB, 16 KB, 64 KB.
- FTDS trace: the original message sizes used by each application, and modified payloads of exactly 256, 512 or 1024 bytes.

We use the following client loads:

- MEAD trace: 1, 4, 7, 10, 13, 16, 19 or 22 clients.
- FTDS trace: 1, 4, 7 or 10 clients.

We induce different request rates by introducing an artificial "think-time" between requests. The request-rates recorded are specific to each benchmark, owing to the differences in latency profiles. We introduce the following inter-request think times:

- MEAD trace: 0 ms, 0.5 ms, 2 ms, 8 ms or 32 ms between requests;
- FTDS trace: 0 ms, 20 ms or 40 ms;

Impact of faults. We compare the fault-free unpredictability with the recovery time needed after a crash fault (MEAD and FTDS traces). We inject faults by periodically crashing a server replica. A single crash does not a induce complete system failure; 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. This high-latency outlier, measured at the client-side, represents the recovery time of the system. After each crash, we launch a new server replica to preserve the initial replication degree and to prepare the application for handling the next faults.

Challenges and fallacies. Our goal in this paper is to test the hypothesis of limited unpredictability using objective metrics of computer-system evaluation. However, in addition to providing us with empirical data about enterprise applications, the FTDS programming experiment represents a behavioral study requiring subjective interpretation. The 7 student 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 or too deeply involved with the day-to-day implementation of the projects.

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 more subtle; for instance, 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. We have corrected or excluded all the corrupted data that we were able to detect; in Section 6, we candidly share all the instances where we believe that some data inconsistencies might have biased the results. We believe that these

10. With MEAD, the replication style can be changed dynamically, at run-time [4], but this functionality was not used in the experiments presented in this paper.

observations will be useful to others attempting similar programming experiments.

Most importantly, we were concerned that some students might be tempted to adjust their data in order to establish the limited-unpredictability hypothesis that we are trying to test. We informed the students repeatedly that their task was not to confirm or refute this hypothesis, 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 hypothesis; upon closer examination, it became apparent that this statement was based on a data-representation error, where the maximum and 99$^{\text{th}}$ percentile latencies were plotted on different graphical scales.

A.2 Real-World Unpredictability: The ATL Trace

The maximum latency is not always unbounded. For instance, the experiments with the SCTP protocol and the TAO/SCTP middleware, using two redundant connections between hosts, have produced latency ranges that are comparable with the mean latency (see Table 2). These results might not be representative, however, because the experiments were conducted in a single testbed, where the hosts were connected using two crossover cables, and used message payloads only up to 1 KB. Conversely, the 99$^{\text{th}}$ percentile of the latency is not always close to the mean. Figure 12 shows the latency distribution for a JBoss experiment. In this case, the distribution is bimodal: most response times are either less than 10 ms or between 35–45 ms. The second mode is two orders of magnitude smaller than the first one (note the logarithmic Y-scale in the figure), which means that it corresponds to only about 1% of the latency measurements. In such a bimodal distribution, the mean latency occurs in the first mode, while the 99$^{\text{th}}$ percentile occurs in the distant second mode. This observation

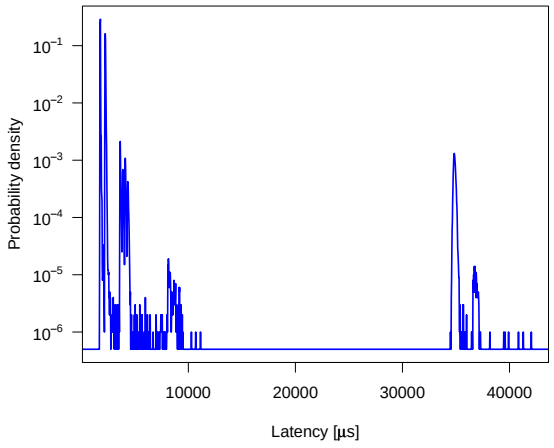


Figure 12. Bimodal latency distribution for JBoss 4.0.5, using EJB 3.0 APIs with a 64 KB payload. $\max \hat{R}_{\text{JBoss}} = 14.5$

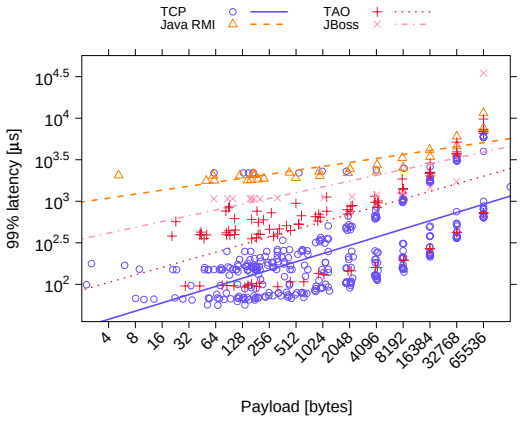


Figure 13. Fitted models for the 99$^{\text{th}}$ percentile.

cannot be generalized either: in the remaining experiments concerning JBoss the latency has a long-tailed distribution, and, in all the other configurations from the ATL trace, the 99$^{\text{th}}$ percentile is predictable according to Criterion ℂ1.

This unpredictability of the maximum latency indicates the lack of *real-time* capabilities for most systems from the ATL trace. Under a real-time schedule, these systems are liable to miss deadlines because their worst-case latency is unknown in advance. In order to enforce real-time guarantees in the absence of a statically-computed schedule that includes all possible tasks, it is sometimes necessary to *preempt* a task that is executing in order to allow a real-time task to complete before its deadline [39]. Most COTS operating systems are not fully preemptible, but the ATL trace includes experiments with UDP, TCP, Spread and TAO on preemptible operating systems (TimeSys and Linux-rt). We compare these configurations against experiments with the same 4 systems, performed on similar testbeds without preemptibility. ANOVA does not identify a significant reduction of unpredictability (in terms of Criterion ℂ1).

For all configurations but one (the experiment illustrated in Figure 12), it is up to $\max R_{conf} \leq 10 \times \overline{R_{conf}}$. Excluding JBoss,[11] the 99$^{\text{th}}$ percentile can be described by a generic model, which does not depend on the system (significant with $p = 0.001$):

$$\tilde{R}_{conf} \approx 1.04 \cdot \overline{R_{conf}} + 263 \text{ µs}$$

Figure 13 compares the fitted least-squares lines[12] with the observed values of the 99$^{\text{th}}$ percentile, for several systems from the ATL trace. The residuals are small, and the least-squares lines have different slopes for each system, which accounts for the different latency ranges discussed above.

11. The experiments that deviate the most from this model are the ones with bimodal latency distributions, such as the one illustrated in Figure 12.

12. Note that these simple regression models do not take into account the testbed, which is a significant factor.

A.3 Sources of Unpredictability: The MEAD Trace

The highest request rates in the MEAD trace were recorded in active replication mode, with 3 server replicas, 22 clients and 16-byte messages. The largest outlier ($\max \hat{R}_{\text{MEAD}} = 660$ occurs in warm-passive replication, with 2 server replicas, 4 KB replies, 16 clients connected and a low request rate (under 1000 req/s, corresponding to a 32 ms client think-time).

We try to determine if the unpredictability recorded in our experiments can be correlated with the behavior of an operating-system mechanism by comparing the relative numbers (Figure 14) and sizes (Figure 15) of outliers with various server-side resource-usage statistics. For all the different message sizes tested, we plot the correlation between the outliers and:

- The time spent executing in user mode;
- The time spent executing in kernel mode;
- The combined execution time;
- The number of minor page-faults (which do not require disk I/O);
- The number of context switches;
- The size of the resident set (the number of pages the process has in physical memory).

For the configurations with a passive replication style the numbers reported represent the measurements for the primary server; for active replication we report the averages across all the server replicas.

In general, we observe that there is very little correlation between these metrics and the reply sizes. While on the client-side the experiments with large reply sizes took longer to complete because the program has to do more work (both in kernel and user mode) to handle these messages, the same is not true for the server side. For the configurations with 64 KB reply-messages, the servers handled between 1 and 22 clients, which accounts for the large variations in run-time visible in Figures 14 and 15.[13]

The minor page-fault rates are grouped in clusters, which correspond roughly to the number of clients connected. We can observe similar clusters for the resident-set sizes, which correspond to the size of the reply messages. Figure 14 shows that, even among experiments with the same size of reply messages, the number of outliers tends to decrease as the number of context switches increases; a similar trend can be noticed in Figure 15 as well.

As in the ATL trace, the maximum end-to-end latencies are unpredictable and do not follow any discernible trends (Finding I). This is illustrated in Figure 16a. The establishment of QoS guarantees on the maximum latency would be virtually impossible. However, the trends for the 99$^{\text{th}}$ percentile latency are obvious in Figure 16b, and the dependence on the message sizes and request rates is evident. The 99$^{\text{th}}$ percentile follows the trend of the mean, with correlation coefficient $r = 0.97$ (Finding III).

13. To emphasize this phenomenon, we have removed 6 configurations with run-times larger than 10 min ($6 \times 10^8 \mu s$) from the figures. These configurations have produced relatively fewer (less than 17) and smaller (z-score < 14) outliers, with one exception where we have recorded 200 outliers. All these configurations correspond to the experiments with 16 KB reply sizes.

A.4 Impact of Design and Implementation: The FTDS Trace

The 35 students from the FTDS course were asked to design and implement a realistic enterprise system and to collect data to analyze the system's behavior. The goal of this course is to teach students the use of good practices in middleware, fault-tolerance patterns (such as replication, transactions, performance optimizations) and software engineering, with the ultimate aim of teaching them how to develop reliable distributed systems.

The students formed 7 project teams. Each team had between 4 and 6 members and focused on a different application of their own choice and design. The projects, summarized in Table 3, varied in size (6-12 KLOC), in scope and in the application domain (online game, e-commerce) that each respectively targeted.

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 fail-over in warm-passive replication or in active replication – the final result will be incorrect. To address this issue, all requests were uniquely numbered and, before processing an invocation, each server replica verified 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. The students were free to implement either an active or a passive replication mechanism for their applications (most teams chose the latter). The replication mechanisms are not transparent, as in the case of MEAD, and the clients are involved in the recovery process.

The 7 teams 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. Two applications used CORBA and five used EJB. All the FTDS applications are implemented in Java. Each project has a three-tier architecture, with a client issuing all the requests, a

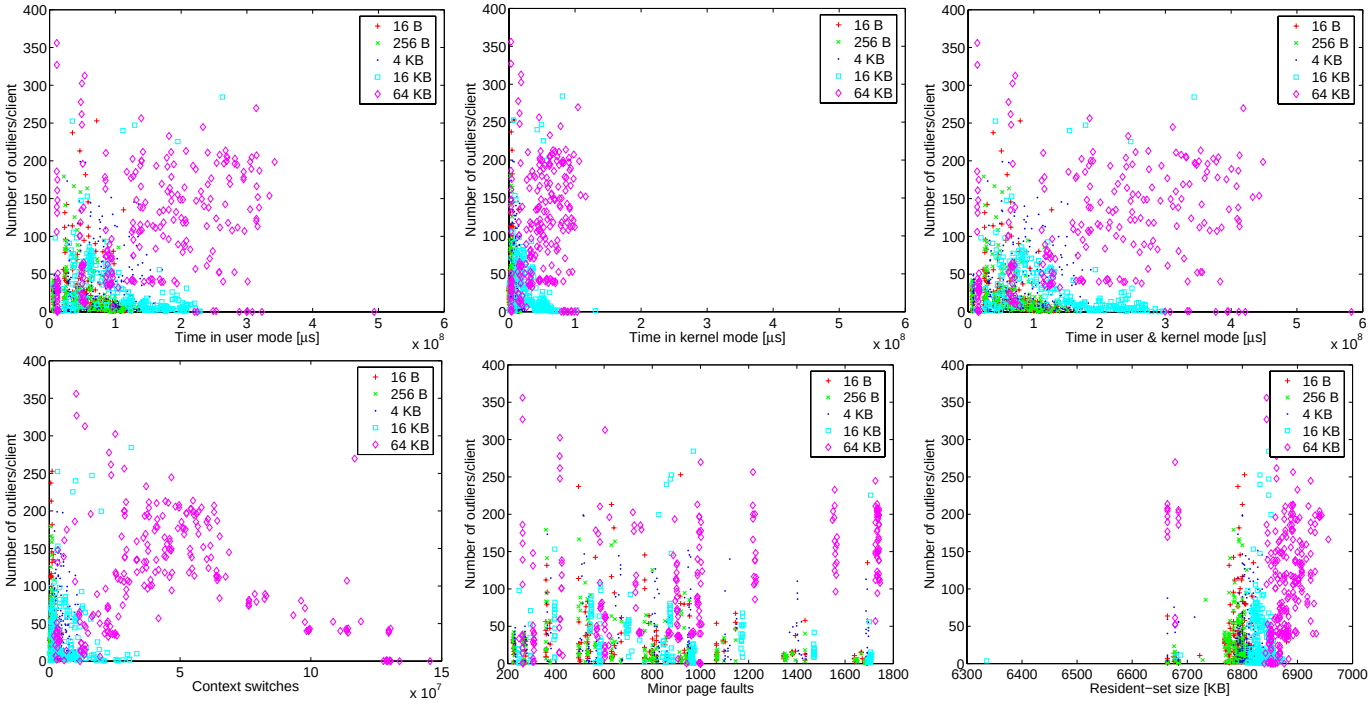


Figure 14. Correlation of system resources with the number of outliers in the MEAD trace.

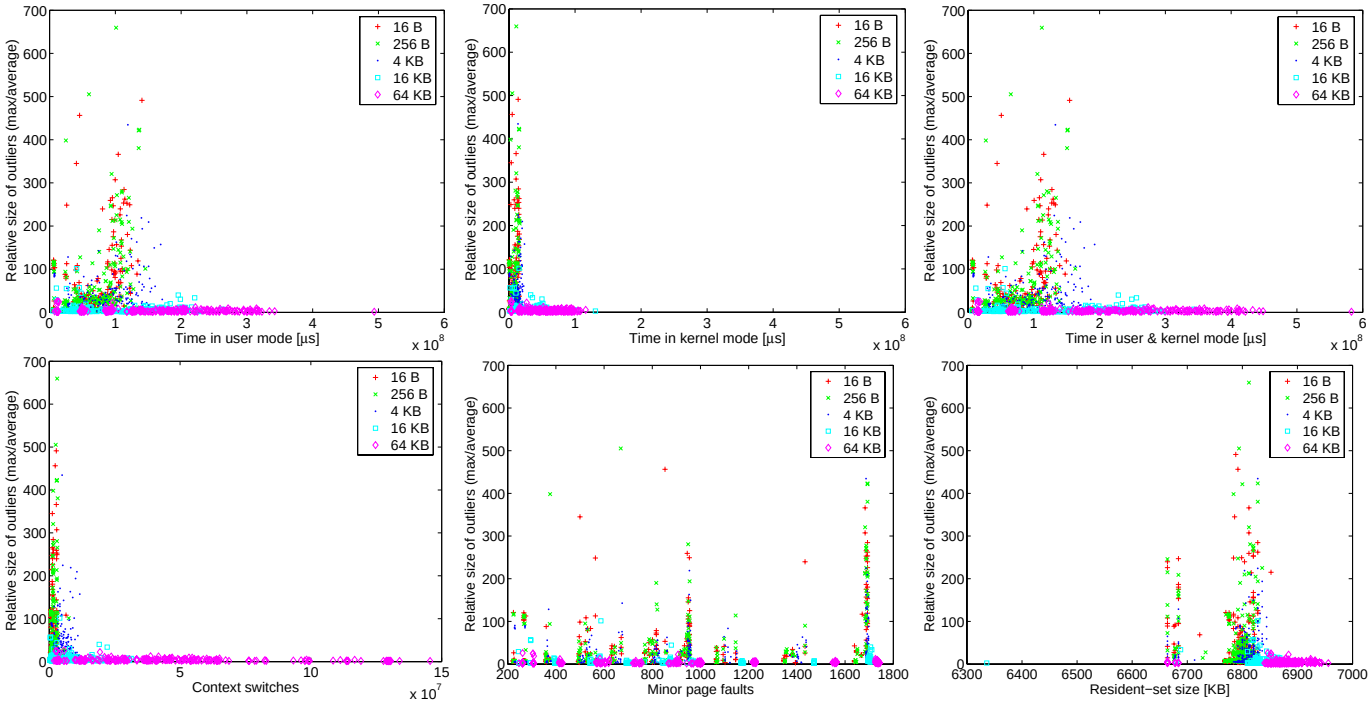
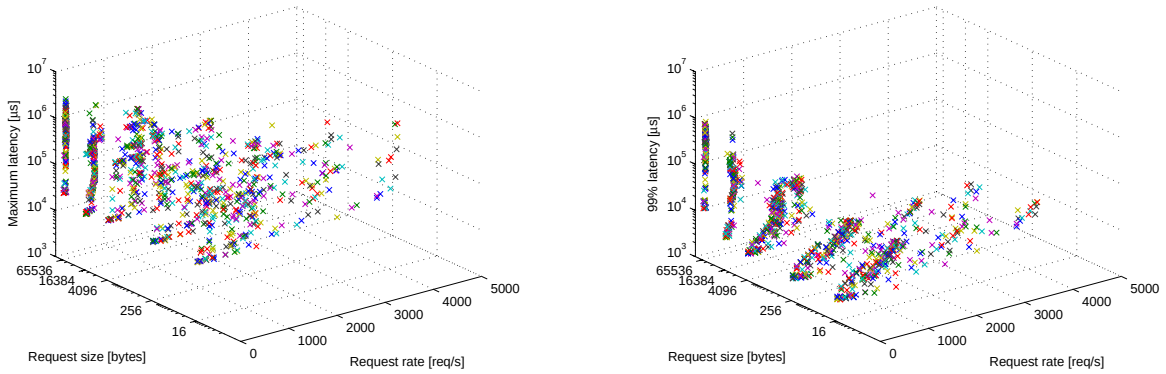


Figure 15. Correlation of system resources with the relative size of the outliers in the MEAD trace.



(a) Maximum latency.

(b) 99th percentile latency.

Figure 16. Trends for the maximum and the 99th percentile latency in the MEAD trace.

stateless middle tier implementing the business 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. These design choices are summarized in Figure 17.

Several factors influence the production of outliers in the FTDS trace:

- 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 JVM's heap-size, the number of large outliers could 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. However, even after increasing the JVM's heap size, project 4 has produced an outlier 3556 times larger than the mean latency from the corresponding experiment.

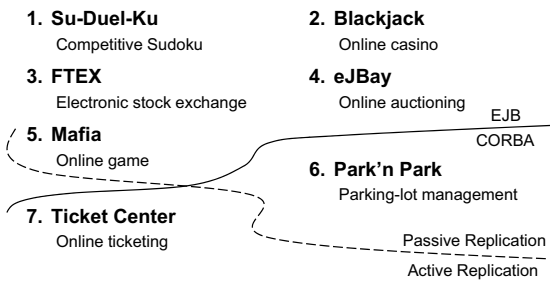


Figure 17. Infrastructure design choices.

- Different applications may produce either many small outliers or a few large ones. The relative size of the outliers seems to be negatively correlated with the number of outliers in each experiment, confirming Finding II. This is consistent for the aggregated configurations from the FTDS trace and across all applications, taken separately.

After removing the magical 1%, we observe four trends in the FTDS trace:

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

The message payload tends to affect the predictability of applications with fast response times, 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. 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 this additional latency. Due to this property and to the fact that this is a CORBA application, project 6 is closest to our experimental setup from the MEAD trace.

As for the MEAD trace, our results suggest that we cannot obtain bounded maximum latencies by changing the request rates, the maximum numbers of clients or the message sizes.