A study of unpredictability in fault-tolerant middleware
Tudor Dumitraș*, Priya Narasimhan

Department of Electrical & Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, United States

A R T I C L E   I N F O

Article history:
Received 26 December 2011
Received in revised form 13 July 2012
Accepted 19 October 2012
Available online 29 October 2012

Keywords:
Latency
Unpredictability
Remote procedure call
Middleware
Fault tolerance

A B S T R A C T

In enterprise applications relying on fault-tolerant middleware, it is a common engineering practice to establish service-level agreements (SLAs) based on the 95th or the 99th percentiles of the latency, to allow a margin for unexpected variability. However, the extent of this unpredictability has not been studied systematically. 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 the systems examined arises from at most 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 filtering out 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.

© 2012 Elsevier B.V. All rights reserved.

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 these systems exhibit and the increasing tuning and configuration burden they impose on the users. These two problems are related because an effective tuning is rendered more difficult if the outcomes of configuration actions are hard to predict. For example, fault-tolerant (FT) middleware, which is used in the most critical enterprise and embedded applications, provides a multitude of specialized configuration options that demand an in-depth understanding of FT semantics.¹ Faced with this configuration challenge, practitioners often find it difficult to identify the causes of unpredictable system behavior. We naturally expect that faults and misconfigurations will have a disruptive effect on the

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.
components (henceforth called the dictability through white-box monitoring of the system’s built, previously—where we dissect the sources of unpredictability with programming practices under our supervision, during the “Fault-Tolerant Distributed Systems” course at Carnegie Mellon University—and we correlate their fault-free and fault-tolerant behavior of such complex systems. Rather than identifying a unique source of the fault-free unpredictability, we show that the high-latency outliers have multiple causes and may originate in any component of the system. Additionally, the maximum latency cannot be regulated through parameters such as the number of clients, the replication degree, the replication style or the request rates, and is not correlated with operating-system metrics such as the amount of time spent in kernel-mode, the number of page faults or the size of the resident set. We also show that, for all the applications and configurations examined, removing 1% of the highest recorded latencies yields predictable latency profiles. Our detailed findings are summarized in Table 1.

1.1. Contributions
We find extensive evidence that middleware systems can have unbounded latencies in the absence of failure, 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. Rather than identifying a unique source of the fault-free unpredictability, we show that the high-latency outliers have multiple causes and may originate in any component of the system. Additionally, the maximum latency cannot be regulated through parameters such as the number of clients, the replication degree, the replication style or the request rates, and is not correlated with operating-system metrics such as the amount of time spent in kernel-mode, the number of page faults or the size of the resident set. We also show that, for all the applications and configurations examined, removing 1% of the highest recorded latencies yields predictable latency profiles. Our detailed findings are summarized in Table 1.

1.2. Roadmap
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 our findings 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 [6] 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 (RPCs), abstracting away the platform and protocol specific details for network programming. Additionally, fault-tolerant (FT) middleware [7] 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 [8,9].

Fig. 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
marchals 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 [10]) or using group-communication protocols (e.g., Spread [11], 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 [12,13]. 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 representative: it must be applicable to more than one system, and it should cover the configurations, workloads and environments that are typical for those systems.
- The rule must be pragmatic: it must not rely on detailed mathematical models of system latency, but, rather, it should provide a straightforward mechanism for handling and reasoning about unpredictability when designing systems or negotiating SLAs.

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 1.** 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 [14]).
- **C3** The latency is not influenced by configuration parameters or by the environment. Specifically, we perform an analysis of variance (ANOVA [14]) to determine the correlation between these parameters and the end-to-end latency. When possible, we also investigate the latency outliers to determine their cause.

**Threats to validity.** Our findings are valid for the systems and configurations tested; we caution the reader that any extrapolations should be made with great care. We have tested our hypothesis in clusters connected with a local-area network, using 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. These settings are representative for applications using fault-tolerant middleware based on CORBA and EJB. Our results may not be valid in environments with high propagation delays, such as wide-area networks, or with intermittent connectivity, such as wireless networks. Moreover, for certain applications (e.g., embedded real-time systems) bounding the 99th percentile is not enough; in such cases, nothing short of predictable worst-case behavior will be sufficient.

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, 99th percentile and maximum latency.

We combine three experimental traces, ATL, MEAD and FTDS, which have been collected independently. 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 Fig. 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 differences between the ATL, MEAD and FTDS traces are summarized in Table 2. 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.
We instrument the systems examined by inserting time probes inside the client–side workload generator and at the boundaries between the layers from Fig. 1. Each probe records timestamps using the gettimeofday() system call, invoked through the Java Native Interface (JNI) for the Java experiments. On all the operating systems used in our experiments, this POSIX system call provides a very accurate timer, based on the CPU cycle counters, allowing us to record timestamps with microsecond precision. This allows us to measure the end-to-end latencies $R_{\text{conf}}(l)$ experienced by each client, and, in the MEAD trace, to decompose them into the components due to each layer: $R_{\text{Application}}$, $R_{\text{Middleware}}$, $R_{\text{Replication}}$ and $R_{\text{Communication}}$. In the FTDS trace, we decompose the end-to-end latency into two components: $R_{\text{Middleware}}$, combining the processing time of the middleware layer and the network delay between the client and the middle tier, and $R_{\text{server}}$, representing the response time of the business logic and database accesses. Because the FT mechanisms are custom implemented in the FTDS applications, the $R_{\text{server}}$ component of the latency combines the $R_{\text{Application}}$ and $R_{\text{Replication}}$ components from the MEAD trace. The database latency does not have an analogue in the MEAD trace. Conversely, the agreement latency does not have an analogue for the FTDS applications, because they do not rely on group communication (see Fig. 1).

To minimize the interference from the experimental harness, we pre-allocate 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. 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. One system (MEAD) is examined in two different experimental traces and produces similar outcomes, which confirms the repeatability of our results.

3.1.1. Effect 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-rt3 (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).

Table 2 Qualitative comparison of the three experimental traces.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MEAD</th>
<th>FTDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro-benchmark</td>
<td>Link and node failures, with no single point of failure (the Replication Manager)</td>
<td>Java</td>
</tr>
<tr>
<td>None (most systems), single link-failures (SCTP, TAO/SCTP), node failures (MEAD)</td>
<td>C++, Java</td>
<td></td>
</tr>
<tr>
<td>Operating system</td>
<td>TimeSys 3.1, Linux 2.4.2–2.4.20, Linux 2.6.8–2.6.23 (including Linux-rt)</td>
<td>TimeSys 3.1, SUSE Linux 2.6</td>
</tr>
<tr>
<td>Data collection</td>
<td>Third party</td>
<td>Authors Under the authors’ supervision</td>
</tr>
</tbody>
</table>

3.1.2. Effect 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.

3.1.3. Effect 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.

3.1.4. Effect 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.

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 [9], one replica is actively processing requests while several backups are waiting to take over after a fault. In active (state-machine) replication [8], all the replicas accept and process the same requests, and the system tolerates faults as long as at least one replica remains functional.
3.2. Data summary

Table 3 summarizes our experiments. The systems from the ATL, MEAD and FTDS traces cover a broad spectrum of latency profiles.

<table>
<thead>
<tr>
<th>Experimental configurations</th>
<th>System type</th>
<th>Language</th>
<th>Fault model</th>
<th>Latency range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ATL Trace (Section 4)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spread [11]</td>
<td>Message-oriented group communication</td>
<td>Java</td>
<td>M, L, N, P</td>
<td>0.3–8.3 ms</td>
</tr>
<tr>
<td>Java RMI [16]</td>
<td>Distributed-object middleware</td>
<td>Java</td>
<td>M</td>
<td>0.4–9.3 ms</td>
</tr>
<tr>
<td>JacORB [17]</td>
<td>Distributed-object middleware (CORBA)</td>
<td>Java</td>
<td>M</td>
<td>0.2–8.4 ms</td>
</tr>
<tr>
<td>TAO/TCP [18]</td>
<td>Distributed-object middleware (CORBA)</td>
<td>C++</td>
<td>M</td>
<td>87 μs–9.5 ms</td>
</tr>
<tr>
<td>TAO/SCTP [18]</td>
<td>Distributed-object middleware (CORBA)</td>
<td>C++</td>
<td>M, 1L</td>
<td>607–894 μs</td>
</tr>
<tr>
<td>MEAD [3]</td>
<td>Distributed-object middleware (CORBA)</td>
<td>C++</td>
<td>M, N, L, P</td>
<td>1–12.5 ms</td>
</tr>
<tr>
<td>JBoss [19]</td>
<td>Component middleware (EJB)</td>
<td>Java</td>
<td>M</td>
<td>0.7–2.4 ms</td>
</tr>
<tr>
<td><strong>MEAD Trace (Section 5)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FTDS Trace (Section 6)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1: Su-Duel-Ku</td>
<td>Enterprise application (EJB based)</td>
<td>Java</td>
<td>M, 1N</td>
<td>10–77 ms</td>
</tr>
<tr>
<td>2: Blackjack</td>
<td>Enterprise application (EJB based)</td>
<td>Java</td>
<td>M, 1N</td>
<td>7–54 ms</td>
</tr>
<tr>
<td>3: FTEX</td>
<td>Enterprise application (EJB based)</td>
<td>Java</td>
<td>M, 1N</td>
<td>48–697 ms</td>
</tr>
<tr>
<td>4: eJBay</td>
<td>Enterprise application (EJB based)</td>
<td>Java</td>
<td>M, 1N</td>
<td>15–125 ms</td>
</tr>
<tr>
<td>5: Mafia</td>
<td>Enterprise application (EJB based)</td>
<td>Java</td>
<td>M, 1N</td>
<td>23–157 ms</td>
</tr>
<tr>
<td>6: Park'n Park</td>
<td>Enterprise application (CORBA based)</td>
<td>Java</td>
<td>M, 1N</td>
<td>3–7 ms</td>
</tr>
<tr>
<td>7: Ticket Center</td>
<td>Enterprise application (CORBA based)</td>
<td>Java</td>
<td>M, 1N</td>
<td>61–1018 ms</td>
</tr>
</tbody>
</table>

* Faults tolerated: M: Message loss; L: Multiple link failures; IL: Single link failure; N: Multiple link failures; IN: Single node failure; P: Network partition.

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 [10]—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) [15] 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 [11] is a package of group-communication protocols, which enforce extended virtual synchrony (EVS) [20] 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 Fig. 1). Java RMI [16], JacORB [17], TAO [18] and MEAD [3] 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 [19] 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

Fig. 2a shows the high discrepancy between the mean 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 higher than the mean latency (note the logarithmic scale on the Y-axis of Fig. 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 Fig. 2b: the randomness seems to disappear, and the 99th percentiles follow the trend of the mean values.

---

* The highest mean latency from the ATL trace was recorded for TCP, in the only experiment that used message payloads up to 4 MB.
**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 $C_2$).

Fig. 3a 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 $3\sigma$ test. Even fewer requests incur the pathologically-high latencies illustrated in Fig. 2. Fig. 3b shows that, among all experiments, the relative size of the largest outliers ($C_1$) is negatively correlated with the outlier count ($C_2$).

**Finding II:** Across all the experiments in the ATL trace, the relative size of the highest response times and the number of high-latency requests are negatively correlated.

**Implications:** The fault-free unpredictability is caused by a few large-latency outliers.

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 3). 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 99th percentile of the latency is not always close to the mean. Fig. 4a 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 99th percentile occurs in the distant second mode. This observation 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 99th percentile is predictable according to Criterion $C_1$.

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 ($C_3$). 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.

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 [21]. 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).

### 4.2. Limits of unpredictability

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.

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

---

Fig. 3. Frequency of high-latency outliers.

Fig. 4. Examples of unpredictable and predictable behavior.
mean latency, with a correlation coefficient \( r = 0.82 \). Excluding JBoss,\(^7\) the 99th percentile can be described by a generic model, which does not depend on the system (significant with \( p = 0.001 \)):

\[
\bar{R}_{conf} \approx 1.04 \cdot \bar{R}_{conf} + 263 \, \mu s
\]

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 \( C_3 \). This is illustrated graphically in Fig. 4b, which compares the fitted least-squares lines\(^8\) with the observed values of the 99th percentile for several systems from the ATL trace. By eliminating 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 is limited to 1% of the remote invocations.

### 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 mature open-source components 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.

The MEAD system\(^3\), illustrated in Fig. 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\(^2\) 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)\(^18\), which provides excellent mean latency, compared with other middleware systems (see Table 3). MEAD implements both active and passive replication, relying on the Spread (v. 3.17.3) group communication toolkit\(^11\) 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. We conduct these experiments on the Emulab testbed\(^40\).

### 5.1. Configuration predictability

Fig. 6 shows the effect of the configuration parameters on the latency distributions. For an increasing number of clients (Fig. 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\(^6\). By varying the replication parameters and the reply size (Fig. 6c), we 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\(^2\), 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 16 KB, 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.

---

\(^7\) The experiments that deviate the most from this model are the ones with bimodal latency distributions, such as the one illustrated in Fig. 4a.

\(^8\) Note that, for the purpose of simplifying the display of the data, these simple regression models do not take into account the testbed, which is not a numeric factor.
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 the network daemons specific to the Emulab testbed and the daemon of the Spread group communication system. 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.

---

9 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.
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. 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 (Fig. 1). Fig. 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.2.

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 [23], TPC-W [24] or RUBiS [25], 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 4. Each application relies on a middleware platform, either EJB (projects 1–5) or CORBA (projects 6 and 7). As shown in Fig. 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 re-launches the crashed replicas. A Fault Detector monitors the heartbeats of all the server replicas and notifies the Replication Manager when a fault occurs. The clients use the Replication Manager and the Naming Service to obtain references to the server objects during initialization or during crash-fault recovery.

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 allowed to make a number of simplifying assumptions in order to remain focused on the fault-tolerance of the middle-tier server. For instance, they could assume that the Replication Manager, the Naming Service and the database never fail. 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 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 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, FTTEX, 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 than 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 effect on outliers; the maximum sizes of outliers are comparable for all the values of these parameters.

**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.

**Table 4** Characteristics of the 7 applications from the FTDS Trace.

<table>
<thead>
<tr>
<th>Project (# developers)</th>
<th>Replication style</th>
<th>Description</th>
<th>Message payloads requests/replies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Su-Duel-Ku (5)</td>
<td>Passive</td>
<td>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 [26]</td>
<td>4 b/$\approx$200 b</td>
</tr>
<tr>
<td>2: Blackjack (5)</td>
<td>Passive</td>
<td>Gaming application, where users play Blackjack online. Users can create online profiles (stored in the database), place bets and play against the house.</td>
<td>$\approx$30 b/$\approx$6 b</td>
</tr>
<tr>
<td>3: FTTEX (5)</td>
<td>Passive</td>
<td>Electronic stock exchange (e.g. NASDAQ). Users create online profiles, list the current orders for a stock and place buy and sell orders (either market-price or limit); the application matches buy and sell orders automatically. The user profiles and the details of all the transactions are stored in the database</td>
<td>$\approx$30 b/$\approx$50 b</td>
</tr>
<tr>
<td>4: eJBay (6)</td>
<td>Passive</td>
<td>Distributed auctioning system, similar to eBay. Allows posting items for sale and bidding for them; the user profiles and the information related to auctions are stored in the database</td>
<td>116 b/98 b</td>
</tr>
<tr>
<td>5: Mafia (4)</td>
<td>Passive</td>
<td>Online version of the popular “Mafia” game, where users create character profiles and communicate through instant messaging. The application stores the persistent state of the game in the database</td>
<td>$\approx$41 b/4 b</td>
</tr>
<tr>
<td>6: Park’n Park (5)</td>
<td>Passive</td>
<td>Application for managing parking lots. Keeps track of how many spaces are available in the lots and recommends alternative locations when a lot is full</td>
<td>3 b/4 b</td>
</tr>
<tr>
<td>7: Ticket Center (5)</td>
<td>Active</td>
<td>Online ticketing application for express buses, allowing users to search schedules and available seats, buy and cancel tickets and check reservation status</td>
<td>$\approx$16 b/4 b</td>
</tr>
</tbody>
</table>

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 within 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 FTTEX 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.3.

### 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.

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...

---

**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.
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 Fig. 10a, we plot the recov-
ery 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 corre-
sponding to fault-detection, fail-over and normal request
processing.\textsuperscript{10}

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 pro-
ject 4 (eBay), 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 (Fig. 10b).
Every 10 s, 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 in-
crease 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 corre-
sponding configurations are comparable with the recov-
ery times.

\textbf{Finding VII:} For enterprise applications with state-
less middle tiers, the maximum fault-free latency
is often comparable with the time needed to
recover from crash faults.

\textbf{Implications:} It is worth reexamining the cost/ben-
efit 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. Be-
cause volatile state can be recreated after a fault and does
not need to be synchronized, the low recovery times re-
ported here are realistic.

\textsuperscript{10} 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.
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. Fig. 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: \( p < 0.01 \), the 99th percentile of the latency is at most 25\% larger than the mean latency. This shows that only 1\% of the requests issued in each configuration are responsible for the unpredictable behavior. By removing these requests, we eliminate the pathologically large end-to-end response times that render the maximum latency unpredictable.

8. Discussion

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). SLAs based on percentiles, rather than on the maximum latency, are common in the industry; for example, Amazon.com provides guarantees on the 99.9th latency percentile [2]. In this paper we analyze the fault-free unpredictability systematically, in 16 distributed systems, and we show that latency percentiles can usually be predicted with high confidence, enabling service-class guarantees based on such statistical measures. Our broad empirical study provides a scientific foundation for the engineering choices made in practice and allows us to answer a number of open questions about unpredictability.

8.1. Can we reconcile fault-tolerance and real-time requirements?

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) [27]. 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.

8.2. Is unpredictability hidden in complex applications?

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, eBay). 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.

8.3. 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. We therefore try to pinpoint the sources of unpredictability by designing an empirical study that encompasses all the layers of the middleware stack—examined both in isolation and as parts of a complex system. In both MEAD and FTDS traces we observe that the typical source of outliers is system-dependent and that outliers might originate in any 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 predominant 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 that: (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 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 on top of the unpredictability related to the potential occurrence of faults.

8.4. How can we design systems that cope with the inherent unpredictability?

Existing approaches for the autonomic management of computer systems [28–33] focus on predicting the average system behavior. When slow requests can be resubmitted to a different server in the replica group without introducing state inconsistencies—e.g. because the requests are read-only, idempotent or because the system employs a duplicate detection technique like the one described in Section 6—this strategy can alleviate the unpredictability of the maximum latency. Proactively retrying up to 1% of requests will reduce the number of outliers because only a fraction of the retries are likely to keep suffering from high latency.

8.5. Does fault-free unpredictability affect data-intensive systems?

Fault-free unpredictability is not characteristic of CORBA and EJB middleware; a similar problem occurs in modern data-intensive systems. Applications based on the MapReduce style of distributed computation [34], which employs fault tolerance techniques similar to the ones we study, also exhibit small numbers of high latency outliers that render the maximum latency unpredictable [35,36]. Ananthanarayanan et al. showed that these outliers have multiple causes, including data skew, cross-rack traffic and disk failures [36]. While these problems did not occur in our experiments, the method presented in this paper, for systematically studying the sources and mechanisms of unpredictability, provides a blueprint for future empirical studies that will expand our understanding of unpredictable behavior in other systems.

9. Related work

Anecdotal evidence and recent studies [12,13] 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 [20]. 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. [37] 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. [38] 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 [39] presents a performance evaluation of FT middleware and suggests techniques for improving this performance.

In [4,5], we presented preliminary analyses of the MEAD and FTDS traces, respectively. In this article, we report additional results for these traces, we include a new trace (the ATL trace), which is likely to emphasize different system behaviors, and we present a comparative analysis of the three traces. This analysis encompasses all the layers of the middleware stack (evaluated both in isolation and when combined into a coherent system). To the best of our knowledge, this represents the broadest empirical study of unpredictability in fault-tolerant middleware.

Because of the inherent unpredictability, existing approaches for the autonomic management of computer systems [28–33], 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 [28]. 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 [29]. 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 [30]. Because of the heterogeneity of COTS-based distributed systems, Mansour et al. [31] 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 on mean response-time [32]. 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 [33]. The emphasis that these systems place on average behavior corroborates our observation that the worst-case latency is difficult to bound and control in many different settings.

10. Conclusions

In this paper, we examine the predictability of 16 middleware systems and communication protocols. By exhaustively exploring 2402 configurations of these 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 up to three 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.

Based on empirical evidence, we introduce a rule of thumb that suggests that the fault-free unpredictability is usually limited to only 1% of remote invocations. We have confirmed this observation for all the systems we examined; moreover, we could find no other rule that effectively isolates the inherent unpredictability. In our experiments, the 99th percentile of the end-to-end latency closely follows the trend of the mean and 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 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 Roșu, Jay Wylie, Arun Iyengar, Jean-Charles Fabre, and the 35 students from the Spring 2006 installment of the Fault-Tolerant Distributed Systems course.

Appendix A. Supplementary material

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.comnet.2012.10.015.

References


Tudor Dumitras is a senior research engineer at Symantec Research Labs (SRL), currently building the Worldwide Intelligence Network Environment (WINE). Tudor’s prior research focused on improving the dependability of large-scale distributed systems (addressing operator errors during software upgrades), of enterprise systems (addressing the predictability of fault-tolerant middleware), and of embedded systems (addressing soft errors in networks-on-chip). He received the 2011 A.G. Jordan Award, from the ECE Department at Carnegie Mellon University, for an outstanding Ph.D. thesis and for service to the community, the 2009 John Vlissides Award, from ACM SIGPLAN, for showing significant promise in applied software research, and the Best Paper Award at ASP-DAC’03. Tudor holds a Ph.D. degree from Carnegie Mellon University.

Priya Narasimhan is the CEO and Founder of YinzCam, Inc., a Carnegie Mellon spin-off company focused on mobile live streaming and technologies to provide sports fans with the ultimate experience in NHL/NFL stadiums. Prof. Priya Narasimhan is also an Associate Professor with the Electrical & Computer Engineering Department. Her research interests lie in the fields of dependable distributed systems, fault-tolerance, embedded systems, mobile systems and software technology. Her work has earned her a 2011 ad:tech Innovation Award, an Alfred Sloan Fellowship, the 2009 Carnegie Science Center’s Emerging Female Scientist Award, an NSF CAREER Award, a Best Paper Award, the 2001 UCSB Lancaster Outstanding Doctoral Dissertation Award and two IBM Faculty Partnership Awards. Her students selected her as the recipient of the 2008 Eta Kappa Nu (Carnegie Mellon Sigma Chapter) Excellence in Teaching Award, and Carnegie Mellon’s College of Engineering selected her as the recipient of its 2009 Benjamin Richard Teare Teaching Award. She was the CTO and, later, the VP of Engineering of Eternal Systems, Inc., a startup company that commercialized the results of her Ph.D. research to develop and sell 24 x 7 highly available platforms and solutions for data centers and large online systems. Her research greatly influenced the development of the Fault Tolerant CORBA industrial standard. She has also served as the Director of Intel Labs Pittsburgh in 2010, and the Director of the CyLab Mobility Research Center, at Carnegie Mellon University in 2009. Priya holds a Ph.D. degree from the University of California at Santa Barbara.