Performance Evaluation of Component-Based Middleware Systems: A Research Survey

Arundhati Kogekar
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, Tennessee, 37235
Email: akogekar@dre.vanderbilt.edu

Abstract—Middleware is of extreme importance in ensuring that a distributed, heterogeneous and large-scale system is scalable, fault-tolerant and manageable. There are many new and evolving component-based middleware technologies available today. Evaluating the performance of these technologies is essential for selecting the best suited middleware for the given set of requirements. This paper explores the various issues involved in the performance evaluation of middleware. It also describes the various evaluation techniques currently in use. With the help of case studies, it compares and contrasts the handcrafted and model-driven approaches toward performance analysis.

I. INTRODUCTION

A conceptual overview of middleware and benchmarking is presented in this section.

A. Middleware

Middleware is the software layer that lies between the operating system and applications and is present on each host in a distributed environment. It consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network. Middleware is essential for migrating mainframe applications to client/server applications and providing for communication across heterogeneous platforms. Examples of middleware frameworks are The Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM), Mono and J2EE. Figure 1[1] shows how middleware fits in between applications and the underlying operating system.

Fig. 1. Middleware

Component middleware is a class of middleware that enables reusable services to be composed, configured, and installed to create applications rapidly and robustly [2]. Component middleware technologies provide higher-level services which allow end-users and developers to specify Quality of Service (QoS) requirements in the context of distributed real-time systems. The CORBA Component Model is a specification for extending CORBA to use components. The J2EE framework is based on the Java Beans Architecture and is used extensively in enterprise Java applications and web services.

Functions of Middleware:

Middleware performs the following functions[3]:

- Hiding distribution, i.e. the fact that an application is usually made up of many interconnected parts running in distributed locations
- Hiding the heterogeneity of the various hardware components, operating systems and communication protocols
- Providing uniform, standard, high-level interfaces to the application developers and integrators, so that applications can be easily composed, reused, ported, and made to interoperable
- Supplying a set of common services to perform various general purpose functions, in order to avoid duplicating efforts and to facilitate collaboration between applications.

B. Benchmarking

Benchmarking is the task of measuring the performance of a system (or a subsystem or an application) on a given task (or workload). The task (or workload) to be measured is denoted as benchmark[4].

Uses of benchmarking:

- Prediction of the performance of a system: Performance of a known system under an unknown set of conditions or tasks or vice versa can be predicted using benchmark analysis.
- Monitoring and diagnostics: A system can be monitored or any problems can be diagnosed by comparing the its performance with the benchmarking results of a similar correctly configured system.
- Assessment of a change: Systematic benchmarking of middleware primitives can reveal performance bottlenecks and bad design decisions as well as errors in the implementation. Detailed, extensive and repetitive benchmarking can be used for finding regressions in middleware implementations. This is useful in assessing the impact of change(both positive and negative) on the performance of middleware.
• Aid in the design of middleware: During middleware design, benchmarking should be used to determine the optimal architecture for given constraints, such as memory capacity, processing power, or network throughput and latency. Important in this aspect is the task of validating models created during the design and thus predicting the real behavior of the architecture. Another important feature of the use of benchmarking to design middleware is the ability to capture and document consequences of using a specific architecture.

The organization of this paper is as follows: Section II explores the issues present in middleware benchmarking. Section III presents various case studies to explain the approaches used in the performance analysis of middleware. Section IV provides the concluding remarks.

II. Issues in Middleware Benchmarking

While evaluating the performance of a particular middleware framework, the following important issues have to be considered[5]:

• Choice of Metrics: Many benchmarking techniques tend to utilize only timestamps for collecting statistics. However, sometimes timestamping by itself will not present a complete picture of the system. Attaching the timestamps with CPU utilization or resource consumption data will improve the accuracy of the results.

• Artificial Workload: Microbenchmarking of an isolated feature of a subsystem constitutes an artificial workload. It is difficult to draw any conclusions about the real-world behavior of the system from these results. Therefore their use should be limited.

• Measurement Techniques: The interdependencies between various measurement techniques should be considered while drawing conclusions about overall system performance. A change or improvement in one set of measurements will affect other measurements as well.

• Level of Granularity: The granular level at which measurements are taken has to be considered carefully. A small suite of measurements makes comparisons simpler and more comprehensible. However coarse-grained measurements may not provide adequate insight into the behavior of the internal subsystems. Therefore there should be a proper mix of fine-grained and generalized measurements.

• Interpretation of Results: Interpreting results is tricky since results are also affected by low-level features such as cache size. The analyst typically is not aware of these low-level factors. The correct interpretation of the results also depends on the precise definition of measurements as well as the exact configuration of the system when the measurement was carried out.

• Layered Architecture: Middleware frameworks have layered architectures. Therefore the impact of the different layers on each other becomes an important issue while benchmarking middleware. Similarly, middleware itself is a layer in between applications and operating system or even another middleware. The interference of these other layers should also be considered while evaluating the middleware performance.

• Random Initial State: A fundamental principle of benchmarking is that a benchmark should be reproducible. However, there exists an inherent non-determinism in the system known as random initial state. This is caused due to non-determinism in memory allocation and code compilation. It is unavoidable, and will introduce some anomalies in the benchmarking results. Figure 2 shows the effect of random initial state on consecutive runs of a benchmark[6].

Fig. 2. Effect of Random Initial State

III. Case Studies

With the help of case studies, this section discusses the various approaches toward middleware benchmarking. The benchmarking techniques are categorized in the following way:

• Hand-crafted Benchmarking
  – Regression Benchmarking for Mono
  – Performance Evaluation of DCOM
  – Open CORBA Benchmarking

• Model Driven Benchmarking.
  – CCMPPerf

A. Handcrafted Benchmarking Techniques

In handcrafted benchmarking, the benchmarking code is written manually by a developer. This paper presents three case studies which implement the handcrafted benchmarking techniques.

1) Regression Benchmarking for Mono: Regression benchmarking[7] is the use of detailed, extensive benchmarks repeatedly to identify the performance bottlenecks or design flaws in a system. Regression benchmarking can therefore be used to quantify the impact of change on the system performance. While regression benchmarking is an important tool in system development, it does have its own associated costs. These costs can be categorized as follows:

• Design: Extensive developer resources have to be utilized to design a comprehensive regression benchmarking suite.
• Execution: A benchmark needs to be run repeatedly to get meaningful results. Extensive, repetitive benchmarks
take a large amount of time to execute. They also generate gigabytes of data, which incur additional cost of storage.

- **Evaluation:** The gigabytes of raw generated data have to be processed before they can be analyzed. This consumes a lot of processing time and developer resources.

The Mono project[8] is an open source development platform based on the .NET framework. It allows developers to build Linux and cross-platform applications with improved developer productivity.

A test system for the regression benchmarking of Mono is described in[9]. The system monitors the performance of Mono by running four benchmarks on the daily snapshots of the current state of Mono. Due to the extremely large amount of data generated, an important part of regression benchmarking is the automated analysis of the results. This analysis is required to locate changes in source code corresponding to performance changes. Figure 3[8] shows the performance changes corresponding to version changes in Mono as measured by the HTTP Ping benchmark.

2) Performance Evaluation of DCOM: DCOM (Distributed Component Object Model) is a set of extensions made to Microsoft’s Component Object Model for the distributed environment. It provides the capability to dynamically load/unload components, a shared memory management scheme between components, support for network interoperability and remote communication. Additional information about DCOM can be found in [10].

An in-depth performance analysis of DCOM has in general been overlooked by the developer community, most probably because it has a proprietary code base with which developers are not allowed to experiment. [11] is one of the few papers which analyze DCOM performance. It categorizes the performance overhead of DCOM into three types:

- **Marshalling overhead** incurred during gathering the call stack data into RPC buffers.
- **Run-time overhead** incurred during data-size independent DCOM/RPC processing.
- **Transport overhead** which is the sum of the execution time for RPC loadable transport, the TCP protocol stack, and actual wire time.

Figure 4[11] shows the results obtained during this performance analysis. It can be seen that the transport overhead dominates the round-trip latency.

<table>
<thead>
<tr>
<th>Round-trip Latency (full call)</th>
<th>413 µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round-trip Latency (64KB)</td>
<td>12.6 ms</td>
</tr>
<tr>
<td>Max. DCOM Bandwidth</td>
<td>11.4 MB/s</td>
</tr>
<tr>
<td>DCOM/RPC Runtine</td>
<td>100 µs</td>
</tr>
<tr>
<td>Marshalling Time</td>
<td>7 µs + 18 µs/KB + 34 µs/KB above 50K</td>
</tr>
<tr>
<td>Transport Time</td>
<td>153 µs + 110 µs/RB + 260 µs/584 bytes</td>
</tr>
</tbody>
</table>

Fig. 4. Performance Analysis of DCOM

Based on this analysis, Li, et al.[11] suggest the use of Virtual Interface Architecture(VIA), reduction in data copying at the marshaling layer and efficient flow control in order to minimize the various performance overheads.

3) Open CORBA Benchmarking: The CORBA specification was designed by the Object Management Group (OMG) to standardize and facilitate remote communication between objects in a distributed client-server environment. Additional information about the CORBA specification can be found in [12]. Since CORBA is a specification, it has been implemented in numerous ways by different vendors. It is often useful to analyze the performance of each of these implementations. The Open CORBA Benchmarking test suite [13] is an important effort in this regard.

The goal of the Open CORBA Benchmarking initiative is to develop a suite of tests which would be generalized enough to run on any CORBA implementation, and which would cover all commonly used broker functionality. However, this has the potential of making the benchmarks too hard to understand and measure. Therefore two sets of users are considered in [13]:

- **Broker Vendors**, who are interested in their evaluating the performance of their own broker to identify and resolve any performance bugs. They require a detailed, white-box analysis of all the function paths present in their system.
- **Broker Users**, who are concerned with what performance to expect from a particular broker implementation or how to design their application based on the underlying bro-
suites consist of the following types of benchmarks:

- **Invocation benchmarks**: For vendors, invocation benchmarks measure the distribution of delivery and round-trip latency times for a simple invocation with no arguments in all invocation modes. For users, invocation benchmarks just measure the above times for a single invocation mode, since the values are quite close to each other for all modes.
- **Marshaling benchmarks**: For vendors, the marshaling benchmarks measure the dependencies of delivery time and round-trip time for a simple invocation on the type, encapsulation, size and direction of arguments. For users, these benchmarks just measure dependencies of delivery time and round-trip time on the size of character data passed as an argument.
- **Dispatcher benchmarks**: For vendors, dispatcher benchmarks measure the dependence of invocation times on the complexity of interfaces, the hierarchy of object adapter, the servant registration policy, and the number of registered servants. For users, the dependence of invocation times on only the number of registered servants is measured.
- **Parallelism benchmarks**: For vendors, parallelism benchmarks measure the dependence of invocation times for a simple invocation on the number of threads and number of clients that issue or handle the invocations in parallel. For users, the benchmarks are simplified to just measure the dependence of invocation times for a simple invocation on the number of threads with a single client.
- **Miscellaneous benchmarks**: For vendors, miscellaneous benchmarks measure the performance of the broker for functions which are not directly related to invocation, such as dynamic any inspection. These benchmarks are not included in the user sub-suite.

The Open CORBA Benchmarking Suite follows the full disclosure policy while reporting the results. Thus the report contains not only the results of the benchmarks, but also the description of the benchmark, a description of the form in which results are reported, an explanation of what the results signify and when simplifications might not apply. The suite also tries to abstract the results from the influence of the system on which the benchmark is executed. This enables accurate comparison between different brokers running on different systems.

More information about the Open CORBA Benchmarking suite, along with sample benchmarking results, can be found in [13].

**Limitations of Handcrafted Benchmarking**: Although handcrafted benchmarking is widely used, it does have its limitations. These limitations arise due to the accidental complexity of large-scale middleware systems. Handcrafted benchmarking incurs a lot of costs in the following situations:

- **Combining performance factors**: Users often want to know the inter-relation between performance factors in their system. This is difficult to analyze using handcrafted benchmarks.
- **Incorporating background load**: It is time-consuming and difficult to adapt handcrafted benchmarking code to the variability in the background load.
- **Incorporating network conditions**: Handcrafted benchmarking code does not consider the variability in network conditions.
- **Reporting results**: There is no standard way of reporting benchmarking results. In addition, if a user wants results in a different format, changes in the benchmarking code have to be done manually, which is error-prone and tedious.
- **Portability of results**: Handcrafted benchmarking code is specific to a platform or operating system. This makes it difficult to run the benchmark on other systems or to compare results obtained on different systems.

**B. Model-driven Benchmarking**

The limitations of handcrafted benchmarking techniques are addressed by model-driven benchmarking techniques in the following ways:

- Model-driven benchmarking provides the right level of abstraction to visualize and analyze the experiment. This saves considerable time and efforts since the developer does not have to write error-prone and tedious low-level code for evaluating performance.
- This approach provides the capability to automatically adapt benchmarking code to changing configurations. The system configuration can be changed in the model and benchmarking tool would generate the corresponding benchmarking code.
- Model-driven benchmarking makes performance evaluation at design time possible, which enables the system designer to correct faulty design decisions at the right time.

The following section provides more details about the process of model-driven benchmarking with the help of a case study.

1) **CCMPerf**: The CORBA Component Model(CCM) is a component-based extension to the CORBA 2.x specification. It is also a specification and therefore many developers have come up with their own CCM implementations. CCMPerf[2] is a model-driven benchmarking suite designed to compare and contrast the performance of different CCM implementations. Figure 5 [14] illustrates the various steps involved in benchmarking CCM implementations using CCMPerf.

1) Using the component repository, an experimenter selects the appropriate components that he/she wants to experiment with.
2) Using CCMPerf, he/she models a component assembly, which is a collection of components which interact with each other.

3) CCMPerf generates the required descriptor files, benchmarking code and script files required to perform the experiment.

4) The test is then run on a target platform and the results generated are used as a feedback to help the planner make a more informed decision.

The modeling paradigm of CCMPerf focuses on three aspects [14]:
- **Configuration**, that describes the interfaces provided or required by a component
- **Metric**, that defines the metric captured in the benchmark.
- **Inter-connection**, that defines how the components will interact in the particular benchmarking experiment.

A constraint checker validates the experiment precluding the possibility of an invalid configuration, such as conflicting metrics, invalid connections or incompatible exchange format. Constraints are defined in the CCMPerf meta model and are defined using Object Constraint Language (OCL). The use of constraints ensures that the experiment is correct a priori, thus minimizing errors at run-time. This is one of the important benefits of using a model-driven approach toward benchmarking.

The benchmarking tests followed in CCMPerf focus on the following two types of metrics:
- **Black-box Metrics**: CCMPerf can be used to measure the performance of an implementation without knowledge of its internal structure. The black-box metrics include round-trip latency, throughput, jitter, collocation performance, data-copying overhead and memory footprint.
- **White-box metrics**: White-box metrics use explicit knowledge of system internals to record and analyze benchmarking data. The white-box metrics supported by CCMPerf include functional path analysis, lookup-time analysis and context-switch overhead.

The benchmarking tests in CCMPerf can be categorized into three general areas, each of which uses a range of white-box and black-box metrics for performance evaluation. These areas are briefly described below:
- **Distribution middleware benchmarks**: These calculate the distribution performance overhead. They can measure the additional overhead incurred by the CCM CORBA 3.x specification over the CORBA 2.x specification. They are helpful for users who are considering transitioning from CORBA 2.x to CORBA 3.x or for developers who need to know if the CCM implementation meets end-to-end QoS requirements.
- **Common middleware service benchmarks**: These can be used to analyze the trade-offs between using standard CORBA 2.x services and customized user-defined implementations.
- **Domain specific middleware benchmarks**: These tests can be used to test whether a particular CCM implementation can be used satisfactorily in a particular domain.

The use of CCMPerf in the performance analysis of one CCM implementation, the Component Integrated ACE ORB (CIAO), is described in [2].

### IV. Conclusion

This paper covers a range of approaches toward the performance evaluation of middleware. It shows how some of the limitations of hand-crafted techniques can be overcome by using model-driven benchmarking techniques. It may be safely predicted that due to their many benefits, model-driven benchmarking approaches will continue to become more widespread as more and more complex "'systems of systems'” are developed.

### References


