A Performance Analysis Framework for Mobile Agent Systems

Marios D. Dikaiakos George Samaras

Department of Computer Science
University of Cyprus, CY-1678 Nicosia, Cyprus
{mdd,cssamara}@ucy.ac.cy

Abstract. In this paper we propose a novel performance analysis approach that can be used to gauge quantitatively the performance characteristics of different mobile-agent platforms. We materialize this approach as a hierarchical framework of benchmarks designed to isolate performance properties of interest, at different levels of detail. We identify the structure and parameters of benchmarks and propose metrics that can be used to capture their properties. We present a set of micro-benchmarks, comprising the lower level of our hierarchy, and examine their behavior when implemented with commercial, Java-based, mobile agent platforms.

1 Introduction

Quantitative performance evaluation is crucial for performance "debugging," that is the thorough understanding of performance behavior of systems. Results from quantitative performance analyses enhance the discovery of performance and scalability bottlenecks, the quantitative comparison of different platforms and systems, the optimization of application designs, and the extrapolation of properties of future systems. The quantitative performance evaluation of mobile-agent systems is much harder than the analysis of more traditional parallel and distributed systems. To study MA-system performance, one should take into account issues such as [4]: the absence of global time, control and state information; the complicated architecture of MA platforms; the variety of distributed computing (software) models applicable to mobile-agent applications; the diversity of operations implemented and used in MA-based applications; the constinuously changing resource configuration of Internet-based systems, and the impact of issues affecting the performance of Java, such as interpretation versus compilation, garbage collection, etc.

In this context, we focus on the *quantitative performance evaluation* of mobile agents. In particular, we introduce a performance analysis approach that can be used to gauge the performance characteristics of different mobile-agent platforms used for the development of systems and applications on Internet. This approach defines a "hierarchical framework" of benchmarks designed to isolate performance properties of interest, at different levels of detail. We identify the

T. Wagner and O.F. Rana (Eds.): Infrastructure for Agents, LNAI 1887, pp. 180–187, 2001.

structure and parameters of benchmarks and propose metrics that can be used to capture their properties. We implement these benchmarks with a number of Java-based, mobile agent platforms (IBM's Aglets [3], Mitsubishi's Concordia [6], ObjectSpace's Voyager [5], and IKV's Grasshopper [2]) and run various experiments. Experimental results provide us with initial conclusions that lead to further refinement and extension of benchmarks and help us investigate the performance characteristics of the platforms examined. The remaining of this paper is organized as follows: Section 2 describes our performance analysis framework. Section 3 introduces the hierarchy of benchmarks we defined to implement this framework, and presents our experimental results from the lower layer of benchmarks, the *micro-benchmarks*. We conclude in Section 4.

2 A Performance Analysis Framework

Basic Elements

To analyze the performance of mobile-agent platforms, we need to develop an approach for capturing basic performance properties of these platforms. These properties should be defined independently of the various ways each particular mobile-agent API can be used to program and deploy applications and systems on Internet. To this end, our approach focuses on *basic elements* of MA platforms that implement the functionalities commonly found and used in most MA environments. Also, it seeks to expose the performance behavior of these functionalities: how fast they are, what is their overhead, if they become a performance bottleneck when used extensively, etc. For the objectives of our work, the basic elements of MA platforms are identified from existing, "popular" implementations [2, 5, 6] as follows:

- *Agents*, which are defined by their state, implementation (bytecode), capability of interaction with other agents/programs (interface), and a unique identifier.
- *Places*, representing the environment in which agents are created and executed. A place is characterized by the virtual machine executing the agent's bytecode (the *engine*), its network address (location), its computing resources, and any services it may host (e.g., a database gateway or a Web-search program).
- *Behaviors* of agents within and between places, which correspond to the basic functionalities of a MA platform: creating an agent at a local or remote place, dispatching an agent from one place to another, receiving an agent that arrives at some place, communicating information between agents via messages, multicasts, or messenger agents, synchronizing the processing of two agents, etc.

Application Kernels

Basic elements of MA environments are typically combined into *application kernels*. Application kernels define scenarios of MA-usage in terms of a set of places participating in the scenario, a number of agents placed at or moving

between these places, a set of interactions of agents and places (agent movements, communication, synchronization, resource use). Essentially, application kernels describe solutions common to various problems of agent design. These solutions implement known models of distributed computation on particular application domains [8]; they represent widely accepted and portable approaches for addressing typical agent-design problems [1]. Typically, application kernels are the building blocks of larger applications; their performance properties affect the performance behavior of applications.

The performance traits of an application kernel depend on the characteristics of its constituent elements, and on how these elements are combined together and influence each other. For example, an application kernel could involve an agent residing at a place on a fixed network and providing database-connectivity services to agents arriving from remote places over wireless connections. This kernel may exist within a large digital library or e-commerce application. It may, as well, belong to the "critical path" that determines end-to-end performance of that application. To identify how the kernel affects overall performance, we need to isolate its performance characteristics: what is the overhead of transporting an agent from a remote place to a database-enabled place, connecting to the database, performing a simple query, and returning the results over a wireless connection. Interaction with the database is kept minimal, is we are trying to capture the overhead of this kernel and not to investigate the behavior of the database.

Investigating the performance of "popular" application kernels can help us explain the behavior of full-blown applications built on top of these kernels. Consequently, a study of application kernels has to be included in our performance analysis framework and should focus on simple metrics capturing basic performance measurements, overheads, bottlenecks, etc. For performance analysis purposes, we define application kernels corresponding to the Client-Server model of distributed computing and its extensions: the Client-Agent-Server model, the Client-Intercept-Server model, the Proxy-Server model, and variations thereof that use mobile agents for communication between the client and the server. More details on these models are given in [7, 8]. Besides the Client-Server family of models, we define application kernels that correspond to the *Forwarding* and the *Meeting* agent design patterns, defined in [1, 3]. We choose the *Forwarding* and *Meeting* patterns, because they can help us quantify the performance traits of agents and places in terms of their capability to re-route agents and to host inter-agent interactions.

Parameterization

To proceed with performance experiments, measurements and analyses, after the identification of basic elements and application kernels, we need to specify the *parameters* that define the context of our experimentation, and the *metrics* measured. Parameters determine: a) The *workload* that drives a particular experiment, expressed as the number of invocations of some basic element or application kernel. Large numbers of invocations correspond to intensive use of the element or kernel during periods of high load. b) The *resources* attached to

participating places and agents: the channels connecting places, the operating system and hardware resources of each place, and the functionality of agents and places.

The exact definition of parameters and parameter-values depend on the particular aspects under investigation. For example, to capture the intrinsic performance properties of basic elements, we consider agents with limited functionality and interface, which carry the minimum amount of code and data needed for their basic behaviors. These agents run within places, which are free of additional processing load from other applications. Places may correspond either to agent servers with full agent-handling functionality or to agent-enabled applets. The latter option addresses situations where agents interact with client-applications, which can be downloaded and executed in a Web browser. Participating places may belong to the same local-area network, to different local-area networks within a wide-area network, or to partly-wireless networks. Different operating systems can be considered.

Parameters become more complicated when studying application kernels. For instance, when exploring the Client-Server model, we have to define the resources to be incorporated at the place which corresponds to the server-side of the model. Resources could range from a minimalistic program acknowledging the receipt of an incoming request, to a server with full database capabilities.

Application Frameworks

Following an investigation of intrinsic performance properties of application kernels, it is interesting to examine how these kernels behave when employed in a real application. To this end, we need to enhance application kernels with the full functionality required by application domains of interest, such as database access, electronic auctions, etc. We call these adapted kernels, *application frameworks*. To experiment with application frameworks, we need to use realistic rather than simple workloads. Such workloads can be derived either from traces of real applications or from models of real workloads.

A Hierarchical Performance Analysis Approach

In view of the above remarks, we propose the analysis of MA-performance at four layers of abstraction as follows: At a first layer, exploring and characterizing performance traits of Basic Elements of MA platforms. At a second layer, investigating implementations for popular Application Kernels upon simple workloads. At a third layer, studying Application Frameworks, that is, implementations of application kernels which realize particular functionalities of interest and run on realistic workloads. Last but not least, at a fourth layer, studying full-blown Applications running under real conditions and workloads.

This hierarchical approach has to be accompanied by proper metrics, which may differ from layer to layer, and parameters representing the particular context of each study, i.e., the processing and communication resources available and the workload applied. It should be noted that the design of our performance analyses in each layer of our conceptual hierarchy should provide measurements and observations that can help us establish causality relationships between the

conclusions from one layer of abstraction to the observations at the next layer in our performance analysis hierarchy.

We propose three layers of benchmarks for the implemention of the hierarchical Performance Analysis Framework introduced in the previous sections. These benchmarks correspond to the first three levels of the hierarchy described earlier:

- **Micro-benchmarks:** short loops designed to isolate and measure performance properties of basic behaviors of MA systems, for typical system configurations.
- **Micro-kernels:** short, synthetic codes designed to measure and investigate the properties of Application Kernels, for typical applications and system configurations.
- **Application Kernels:** instantiations of micro-kernels for real applications. Here, we involve places with full application functionality and employ realistic workloads complying to the *TPC-W* specification (see http://www.tpc.org

3 Micro-benchmarks and Experimentation

We present in more details the suite of proposed micro-benchmarks and a summary of experimental results derived by these benchmarks. Further information about micro-kernels, application kernels and experimental results can be found in [7, 4]. The basic components we are focusing on are: a) mobile agents, used to materialize modules of the various distributed computing models and agent patterns; b) messenger agents used for flexible communication, and c) messages used for efficient communication and synchronization. Accordingly, we define the following micro-benchmarks:

- AC-L: Captures the overhead of agent-creation locally within a place.
- AC-R: Captures the overhead of agent-creation at a remote place.
- AL: Captures the overhead of launching agents towards a remote place.
- AR: Captures the overhead of receiving agents that arrive at a place.
- MSG: Captures the overhead of point-to-point messaging.
- MULT: Captures the overhead of message multicasting.
- SYNCH: Captures the overhead of synchronizing two agents with message-exchange.
- ROAM: Captures the agent-travelling overhead.

These micro-benchmarks involve two places located at different computing nodes, agents with the minimum functionality that is required for carrying out the behaviors studied, and messages carrying very little information between agents. Table 1 presents the parameters and metrics for our benchmarks: "Loop size" defines the number of iterations included in each benchmark. "Operating System" and "Place Configuration" represent the resources of each place involved in our experimentation. We have conducted experiments on PCs running Windows 95 and Windows NT. In most experiments, places were agent servers. We also conducted experiments where one of the places was an agent-enabled Java

Table 1. Micro-benchmark Parameters and Metrics

Name	Parameters						Metrics		
	Loop Size	Operating System	Place Config.	Channel Config.	Agent Size	Msg. Size	Total Time	Average Time	Peak Rate
AC-L	√	√	√	-	√	-	√	√	√
AC-R, AL, AR	√	√	√	√	√	-	√	√	√
MSG, SYNCH, MULT	√	√	√	√	-	√	√	√	√
ROAM	√	√	√	√	√	-	√	√	√

applet. Channel configuration specifies whether the two places involved reside at the same LAN, at two different LANs, or if one of the places gets connected to the other via a wireless link.

As shown in Table 1, we measure three different metrics. Total time to completion is a raw performance metric, which can provide us with some insight about the performance characteristics of the basic element studied by each benchmark. It can also help us identify bottlenecks as the load (loop size) is increased, and test the robustness of each particular platform. Average timings provide estimates of the overhead involved in a particular behavior of a MA system, i.e., the cost of sending a short message, of dispatching a light agent, etc. Finally, peak rates provide a representation of the performance capacity of MA platforms, based on the sustained performance of their basic elements.

The number of parameters involved in our micro-benchmarks lead to a very large space of experiments, many of which may not be useful or applicable. We have experimented with four commercial platforms: IBM's Aglets, Mitsubishi's Concordia, ObjectSpace's Voyager and the Grasshopper by IKV. We tried various parameter settings before settling to a small set of micro-benchmark configurations that provide useful insights. Fig. 1 displays the average times for micro-benchmarks **AC** and **AL** (in msec). AC was executed on a PC running Windows 95. AL was executed on two PCs running Windows 95 and residing at the same LAN, with a small traffic-load. From these figures we can easily see that, in terms of performance, Concordia and Voyager are more optimized than Aglets and Grasshopper. For Concordia, Voyager and Grasshopper, the average time it takes to create and dispatch an agent varies greatly with loop size. This can be attributed to the fact that MA platforms cache the bytecodes of classes loaded during agent creation and dispatch. Therefore, repeated invocations of the same primitive cost less than the initial ones. As the loop size increases beyond a certain point, however, the agent servers hosting the benchmark start facing overloading problems (shortage of memory, higher book-keeping overheads, etc.), leading to a degradation in performance. Behavior changes from one platform to the other, since some systems employ different techniques to cope with overloading.

Similar remarks can be driven from Figure 2, which presents the average times extracted for micro-benchmarks **MSG** and **SYNCH**. These benchmarks were executed on PCs running Windows 95 and residing at the same LAN. From

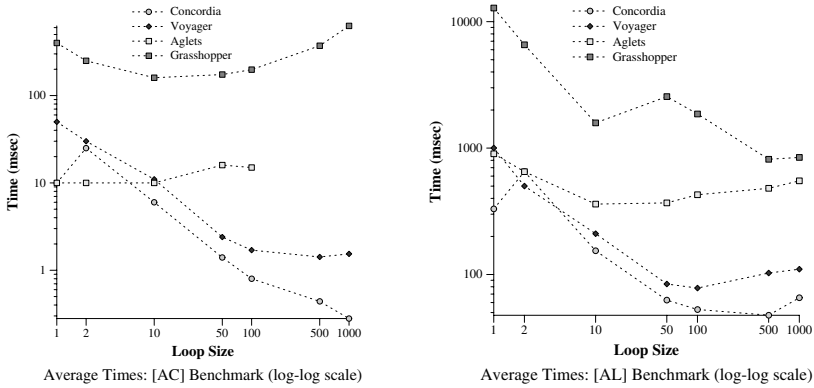

Fig. 1. Average timings for agent creation and launching.

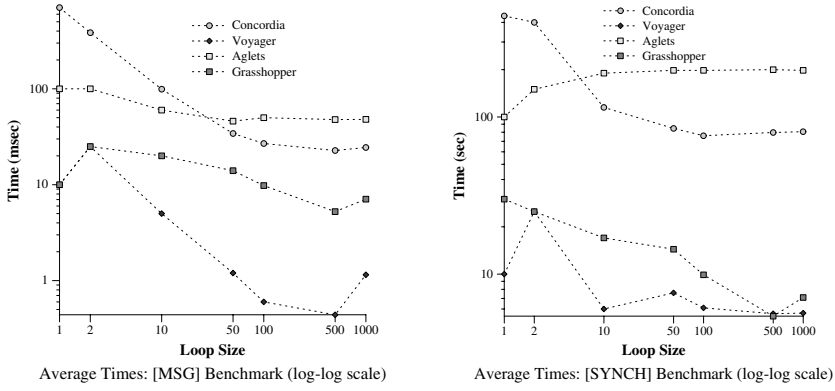

Fig. 2. Average timings for MSG and SYNCH micro-benchmarks.

these diagrams we can easily see that the time to send and exchange messages in Voyager is much shorter than in other platforms. Average message dispatch and exchange times are leveling off with loop size, due to caching of the classes that comprise the message data structures. If loop size exceeds a certain level, the messaging subsystems in all platforms start facing overloading problems (Grasshopper and Voyager). Both Concordia and Voyager, however, prove to be quite robust even under heavy load of agent or message transmissions. It should be noted that, although not shown in the diagrams of Figures 2, messaging is more robust and efficient under Windows NT, for all platforms tested.

Table 2 presents the peak rate of agent creation, agent dispatch, and message dispatch for the MA platforms studied. From these numbers, we can easily see that Concordia is a clear winner when it comes to the number of agents that can be created at and dispatched from a particular agent server. Voyager, on

Table 2. Peak Rates (Windows95)

Benchmark	Concordia	Voyager	Aglets	Grasshopper
Agent Creation (agents/sec)	3571.43	649.3	10.67	1.59
Agent Dispatch (agents/sec)	21	9.07	10.67	1.187
Message Dispatch (msg/sec)	40.9	869.56	20.7	141.44

the other hand, provides more than an order of magnitude higher capacity in message dispatch than other platforms.

4 Conclusions

To our knowledge, our Performance Analysis Framework provides the first structured approach for analyzing the performance of mobile-agent systems quantitatively, by focusing at the different layers of a MA-based system's architecture. Experiments with our micro-benchmark suite provide a corroboration of this approach. Experimental results help us isolate the performance characteristics of MA platforms examined, and lead us to the discovery and explanation of basic performance properties of MA systems. Furthermore, they provides a solid base for the assessment of the relative merits and drawbacks of the platforms examined from a performance perspective.

Acknowledgements: The authors wish to thank C. Spyrou and M. Kyriacou for helping out with experiments.

References

1. Y. Aridov and D. Lange. Agent Design Patterns: Elements of Agent Application Design. In *Proceedings of Autonomous Agents 1998*, pages 108–115. ACM, 1998.
2. M. Breugst, I. Busse, S. Covaci, and T. Magedanz. Grasshopper – A Mobile Agent Platform for IN Based Service Environments. In *Proceedings of IEEE IN Workshop 1998*, pages 279–290, Bordeaux, France, May 1998.
3. D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.
4. M. Dikaiakos and G. Samaras. Quantitative Performance Analysis of Mobile-Agent Systems: A Hierarchical Approach. Technical Report TR-00-2, Department of Computer Science, University of Cyprus, June 2000.
5. G. Glass. Overview of Voyager: ObjectSpace's Product Family for State-of-the-Art Distributed Computing. Technical report, ObjectSpace, 1999.
6. R. Koblick. Concordia. *Communications of the ACM*, 42(3):96–99, March 1999.
7. G. Samaras, M. D. Dikaiakos, C. Spyrou, and A. Liverdos. Mobile Agent Platforms for Web-Databases: A Qualitative and Quantitative Assessment. In *Proceedings of the Joint Symposium ASA/MA '99. First International Symposium on Agent Systems and Applications (ASA '99). Third International Symposium on Mobile Agents (MA '99)*, pages 50–64. IEEE-Computer Society, October 1999.
8. C. Spyrou, G. Samaras, E. Pitoura, and P. Evripidou. Wireless Computational Models: Mobile Agents to the Rescue. In *2nd International Workshop on Mobility in Databases & Distributed Systems. DEXA '99*, September 1999.