Performance Evaluation of Mobile-Agent Middleware: A Hierarchical Approach*

Marios Dikaiakos, Melinos Kyriakou, and George Samaras

Department of Computer Science
University of Cyprus
CY-1678 Nicosia, CYPRUS

Abstract. In this paper, we introduce a hierarchical framework for the quantitative performance evaluation of mobile-agent middleware platforms. This framework is established upon an abstraction of the typical structure of mobile-agent systems and is implemented through a set of benchmarks, metrics, and experimental parameters. We implement these benchmarks on three mobile agent platforms (Aglets, Concordia and Voyager) and run numerous experiments to validate our framework and compare the mobile-agent middleware environments quantitatively. We present results collected from our experiments, which help us understand MA performance and identify existing bottlenecks. Our results can be used to guide the improvement of existing platforms, the performance analysis of other systems, and the performance prediction of MA applications.

1 Introduction

The Mobile Agent (MA) paradigm is one of the most promising approaches for developing distributed applications on Internet [9]. The employment of Java-based MA technologies for the development of next-generation Internet systems opens numerous research problems. In our work, we focus on *quantitative performance evaluation* of mobile agents and propose a framework for investigating the performance characteristics of MA-based platforms and applications.

In this context, we introduce a performance evaluation approach that can be used to gauge the performance characteristics of different mobile-agent platforms. This approach extends and refines previous work of ours [12,6], by defining 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 capture performance properties of interest. We implement these benchmarks upon three Java-based, mobile agent middleware platforms (IBM's Aglets [4], Mitsubishi's Concordia [13] and ObjectSpace's agent-enhanced object request broker, Voyager [7]), and run various experiments.

* This work was supported in part by the Research Promotion Foundation of Cyprus, grant PENEK-No 23/2000.

G.P. Picco (Ed.): MA 2001, LNCS 2240, pp. 244–259, 2001.

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: Sections 2 and 3 introduce our performance analysis framework. Sections 4 and 5 present the implementation of the first two levels of our framework with a suite of micro-benchmarks and micro-kernels, and report our experimentation results. We conclude in Section 6.

2 Basic Elements and Application Frameworks

Typically, the performance assessment of software systems is conducted through experimentation and monitoring, simulation, modeling and combinations thereof. The more complex a system is the harder its performance evaluation becomes, dictating the employment of these techniques at various levels of abstraction. To this end, software systems are modeled as hierarchical structures of interacting modules, i.e., subsystems and objects; each module is assigned a performance model that incorporates performance and load parameters of relevance, and a description of the underlying architecture and workload [14]. Model development is performed in a "top-down" manner, starting from high-level structure and moving toward code implementation. Experimentation and/or simulation can be used at various layers of abstraction to specify the values of modeling parameters.

The development and assembly of performance models for MA middleware is more complicated than for more "traditional" parallel, distributed or object-oriented software; when analyzing the performance of MA-based systems, we must take into account issues such as: the absence of global time, control and state information; the complex architecture of MA middleware and the agility of MA systems; the variety of distributed computing (software) models that are applicable to mobile-agent applications; the diversity of operations found in MA middleware, and the additional complexity introduced by issues that affect the performance of Java (run-time compilation, memory management, garbage collection, etc.).

To cope with the complexity of MA-performance evaluation, we propose the adoption of a hierarchical approach that takes into consideration the structure of typical MA-based applications. This structure is influenced, first, by the mobile-agent platform adopted to develop an application. MA platforms are middleware systems with a programming interface that exposes to the programmer a set of core functionalities providing support for object mobility (transportation and location services), communication between objects, security, fault-tolerance etc. [2, 7,8]. MA platforms are differentiated by their functionality, programming interface and performance characteristics, all of which are influenced by underlying implementation details. The structure of a MA application is further determined by the design choices that the application developper makes on how to use the API provided by the middleware platform, when developing the particular appli-

cation. Typically, these design choices can be abstracted as mobile-agent *design patterns* [1].

Therefore, to investigate the performance of mobile-agent applications, we have first to develop an approach for capturing basic performance properties of MA middleware. These properties must be defined independently of how particular mobile-agent API's are used to program and deploy applications and systems on Internet. Then, we have to analyze the performance characteristics of design patterns commonly used in MA applications. To facilitate this approach, we introduce two abstractions: *Basic Elements* and *Application Frameworks*.

We define as **Basic Elements** the set of basic abstractions that incorporate the fundamental functionalities commonly found and used in MA platforms. For the objectives of our work, the basic elements of MA platforms are identified from existing, "popular" implementations as follows [2,4,7,8]: a) *Agents*, defined by their state, implementation (byte-code), capability of interaction with other agents/programs (interface), and a unique identifier. b) *Places*, representing the environment in which agents are created and executed. A place is characterized by the virtual machine executing the agent's byte-code (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). c) *Behaviors* of agents within and between places, which correspond to the basic functionalities of a MA platform, such as: 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 or messenger agents; synchronizing the processing of two agents; locating an agent on the move, etc.

Basic elements of MA systems are combined into scenarios of MA-use, which we call **Application Frameworks**. In Object-orientation, *software frameworks* represent a way of "structuring generic solutions to a common problem by providing the structure of a program but no application-specific details" [3]. The overall control and the flow of execution is provided by the framework and therefore does not need to be rewritten for each new problem. Accordingly, application frameworks of MA's define solutions common to various problems of agent design and are defined in terms of places participating in a scenario, agents placed at or moving between these places, and interactions of agents and places (agent movements, communication, synchronization, resource use). Application frameworks correspond to widely applicable models of distributed computation on particular application domains, and represent widely accepted and portable approaches for addressing typical agent-design problems [1]. Typically, application frameworks are the building blocks of larger MA applications.

We focus on application frameworks that correspond to the Client-Server model of distributed computing and its extensions for mobile computing: 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 [12]. Additional application frameworks correspond to the *Roaming Mobile-Agent* Model, and the *Forwarding* and *Meeting* agent-design patterns. The Roaming MA model

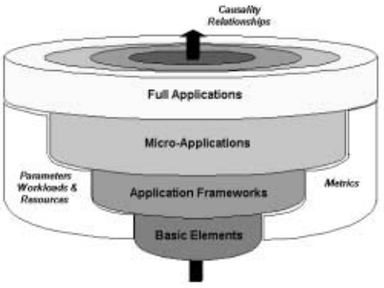

Fig. 1. The Hierarchical Performance Evaluation Framework.

corresponds to the case of an agent that roams from one place to the other, engaging in some interaction with the places visited. The *Forwarding* pattern "allows a given place to mechanically forward all or specific agents to another place" [1]. The *Meeting* pattern provides a way for two or more agents to initiate local interaction at a given place [1,4]. The *Forwarding* and *Meeting* patterns represent the performance traits of agents and places in terms of their capability to re-route agents and to host inter-agent interactions.

3 A Hierarchical Performance Evaluation Framework

In view of the remarks above we propose a framework for the Hierarchical Evaluation of MA-performance, which consists of four layers of abstraction (see Figure 1). At a first layer, our framework explores the performance traits of *basic elements* of MA platforms, seeking to expose their performance behavior: how fast they are, what is their overhead, if they become a performance bottleneck when used extensively, etc.

Having isolated the performance characteristics of basic MA elements, we explore the characteristics of application frameworks in order to explain the performance behavior of full-blown applications that use these frameworks as building blocks. Consequently, at the second layer of our framework, we investigate implementations of popular *application frameworks* upon simple workloads. In particular, we measure metrics capturing the performance capacity of an application framework, the overhead incurred by the interaction of its constituent elements, the bottlenecks affecting its performance, etc. For example, an application framework 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 framework 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 this framework affects overall performance, we have to find out what is the overhead of transporting an agent from a remote place to a database-enabled place, connecting to a database agent, performing a simple query, and returning the results over a wireless connection. Interaction with the database agent should be

kept minimal because we are trying to capture the overhead of this framework and not to investigate database behavior. We also need to quantify how many requests can be served by the database agent per second, etc.

It is interesting to explore the performance behavior of instances of these frameworks under conditions expected to occur in a real execution of a full-blown application. To this end, we can enrich the scenarios implemented in the application frameworks by extending the functionality of mobile agents and by simulating realistic workload conditions. This is the focus of the third layer of our hierarchy, where we study *micro-applications*, i.e., implementations of application frameworks that realize particular functionalities of interest (e.g., database connectivity) and run on synthetic workloads. Finally, at the fourth layer of our framework, we study *full-blown applications* running under real conditions and workloads.

Our 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 stressed that the design of our performance evaluation 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 of our performance analysis hierarchy.

To apply our hierarchical performance evaluation framework in the study and comparison of performance characteristics of different MA platforms and MA-based applications, we propose three layers of benchmarks that correspond to the first three layers of the hierarchy of Figure 1. These benchmarks are defined as follows:

- **Micro-benchmarks:** short loops designed to isolate and measure performance properties of basic elements of MA systems, for typical system configurations. Micro-benchmarks test the performance of simple activities (behaviors) provided by the basic elements of a MA system.
- **Micro-kernels:** short, synthetic codes designed to measure and investigate the properties of application frameworks, for typical system configurations.
- **Micro-applications:** instantiations of micro-kernels for real applications. Here, we propose to use places with full application functionality and employ synthetic workloads complying to specifications like the *TPC-W*.

In the following sections, we introduce a suite of micro-benchmarks and micro-kernels that we use to evaluate the performance of mobile-agent middleware quantitatively. In earlier work we have examined micro-applications that involved the use of mobile agents to provide database access over the Web [12]; a study of micro-applications will be conducted in future work.

Our benchmarks are accompanied by *parameters* that define the context of our experimentation, and the *metrics* measured. Parameters determine the *workload* that drives a particular experiment, expressed as the number of invocations of some basic element or application framework, and the *resources* attached to participating places and agents. Metrics represent a concise description of the performance characteristics isolated by our benchmarks.

Our benchmarks can be parameterized according to the following parameters: *Operating System* and *Place Configuration* represent the resources of each place involved in our experimentation; *Channel Configuration* represents the network upon which we conduct our experiments, which can be a LAN, a WAN, a wireless network, or combinations thereof. *Agent Size* and *Message Size* represent the size of an agent and a message exchanged between two agents, respectively. *Loop size* defines the number of times a particular benchmark is executed to gather time measurements. Additional benchmark-specific parameters are employed in micro-kernels and will be described later.

The number of parameters involved in our benchmarks lead to a huge space of experiments, many of which may not be useful or applicable. Therefore, we have conducted preliminary experiments with three commercial platforms, IBM's Aglets, Mitsubishi's Concordia, and ObjectSpace's Voyager, and tried various parameter settings before settling to a small set of experimental parameters and benchmark configurations that provide useful insights. Our experiments involve places located at different computing nodes within the same LAN, agents with the minimum functionality that is required for carrying out the behaviors studied, and messages carrying minimal information between agents. We have used a 100 Mbps Ethernet with 18 PCs, equipped with Pentium III processors running at 500MHz and 64MB main memory. The PCs ran the Microsoft's Windows NT 4.0 Operating System and Sun's JRE 1.1.7. On this platform we experimented with Aglets version 1.0.3, the professional edition of Voyager ORB, version 3.1, and an evaluation copy of Concordia, version 1.1.4. The experiments were conducted at night, when the utilization of the LAN was minimal. We also ran some experiments under heavier network load (when the lab was used by students to run applications from a central file-server, to browse the Web, etc.). All data reported in the following sections correspond to the low-network-traffic case, unless mentioned otherwise. In future experiments, we plan to incorporate setups including wireless Ethernet and connectivity over WANs.

For most of our benchmarks we report four metrics: *Total time* is the total elapsed time it takes to run a particular benchmark. This metric represents the performance of the basic activity examined by the benchmark. A study of the total-time for different benchmark parameters can identify bottlenecks that arise under high loads (large loop size) and test the robustness of each platform. *Average time* provides an estimate of the time it takes for a particular basic activity of a MA system to complete; for instance, the time of sending a short message, dispatching a light agent, etc. *Peak rate* is the maximum measured rate of a basic activity, defined as the number of these activities carried out per second. *Sustained rate* is the number of basic activities carried out per second, when we conduct stress-tests, i.e., run an experiment continuously over a long period of time. For instance, a sustained rate of 40 for the agent-creation benchmark means that we can generate approximately 40 agents per second on the particular machine running the experiment, if the experiment is executed continuously over a period of time. Additional, metrics are measured in certain micro-kernels and will be described later.

Table 1. Definition of Micro-benchmarks.

Name	Description
CL	Captures the overhead of agent-creation locally within a place.
CR	Captures the overhead of agent-creation at a remote place.
AD	Captures the overhead of dispatching agents toward a remote place; Agents have been created locally.
MSG-1W	Captures the overhead of non-blocking messaging with no acknowledgment from the message recipient.
MSG-2W	Captures the overhead of non-blocking messaging with asynchronous acknowledgment from the message recipient.
SYNCH	Captures the overhead of blocking messaging, which synchronizes two agents using message-exchange.
MSG-MA	Captures the overhead of agent-communication with messenger agents.

4 Micro-Benchmarks

In this section, we present the suite of proposed micro-benchmarks and experimental results derived by these benchmarks. 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 micro-benchmarks presented in Table 1 and present the metrics measured in a number of experiments with these benchmarks. Excerpts of the code implementing these benchmarks are presented in [5].

4.1 Agent Launching: CL, CR, and AD

With the **CL** micro-benchmark we study the overhead of agent-creation. To this end, we generate 1 to 1000 agents and measure the total elapsed time. The left diagram of Figure 2 reports the average time for generating one agent with respect to the total number of created agents. From this diagram, we can easily see that the overhead of creating a single agent in Concordia is negligible with respect to the overhead in Aglets and Voyager; furthermore, that Voyager outperforms Aglets.

It is interesting to note that the time it takes to create an agent drops with the increase of loop size, for all platforms. This happens because, after the first time an agent is created, its byte-codes are already cached in the agent-host's memory. Therefore, subsequent agent creations take minimal time. On the other hand, the better "scalability" of Concordia and Voyager over Aglets that we observe in the left diagram of Figure 2, is attributed to memory management mechanisms implemented in both platforms: when heap space is consumed, the two platforms transfer inactive agents to disk, thus maintaining a minimum of free space [10,11]. Table 2, presents the agent-creation capacity of the three platforms (peak and sustained).

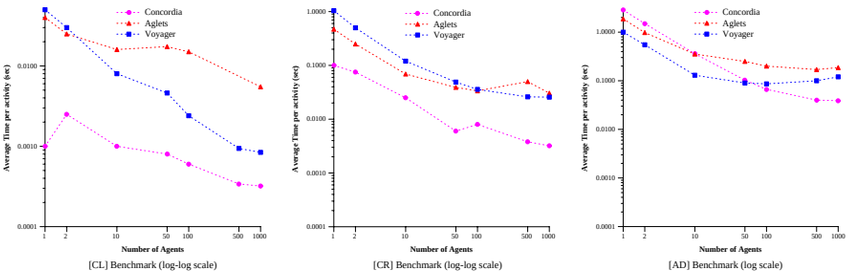

Fig. 2. CL, CR & AD: Average timings for agent creation and dispatch.

Table 2. CL, CR and AD: Peak and sustained rates.

Platform	CL		CR		AD	
	Peak (agents/sec)	Sustained (agents/sec)	Peak (agents/sec)	Sustained (agents/sec)	Peak (agnts/sec)	Sustained (agnts/sec)
Concordia	3125	3000	312.5	310	25.68	25.6
Aglets	65.78	11	29.76	11.05	5.9	5.36
Voyager	1189.06	1100	38.8	38.8	11.58	8.31

The **CR** benchmark measures the total time it takes to create agents at a remote host. To this end, we use a stand-alone JAVA program running on an "origin" host and issuing instructions to generate 1 to 1000 agents on a remote place. We time the overall overhead of agent creation at the origin place. To remotely create agents, the remote place needs to have the necessary classes locally or to be able to download these classes from another place on demand, during agent-creation. This is accomplished in a number of different ways: (a) Under Concordia, a messenger agent migrates from the origin place to another place. Upon arrival, the messenger creates a new agent at the remote place. The messenger transports with it the classes required by the agent under creation. (b) A Voyager agent at the remote place can load classes from other locations on demand. To this end, it employs a Resource Loader object which resides in its Voyager server. The Resource Loader maintains a registry of remote Voyager servers, which may store useful classes and serve them over the network. Whenever an agent seeks a class that is not available in its local classpath, it invokes the Resource Loader which returns an interface (proxy). Through that interface, the agent can access the remote class. (c) An Aglet can load a remote class on demand from a remote Tahiti server, which is the agent execution environment (place) of Aglets. To this end, the Aglet must establish an additional network connection with the remote place. In order to make the remote classes available through the network, they should be placed in the secondary storage of the remote host and be included in the classpath of the remote place at its initialization.

The middle diagram in Figure 2 shows our measurements for the *CR* benchmark. As we can see, Concordia and Aglets have better performance than Voy-

ager for a small number of created agents. Again, Concordia is the clear "winner,"even for large numbers of created agents. As we increase the number of created agents, however, the average time to create an agent in Voyager drops faster than the respective time in Aglets, and the values of the two platforms converge. The performance of the three platforms in terms of their capacity to create agents remotely is summarized in Table 2. It is interesting to note that remote creation of agents under Concordia and Voyager is approximately an order of magnitude slower than local agent-creation. Furthermore, we note that, for Concordia and Voyager, the peak and sustained rates of agent creation are almost equal, which is a result of their improved robustness. In contrast, Aglets performance drops for very large numbers of created agents.

The **AD** benchmark measures the overhead of dispatching mobile agents to a remote place in a LAN. We create and dispatch 1 to 1000 agents to the remote place. We measure *only* the time of the dispatch operation and plot our results in the right diagram of Figure 2. As we can see from this diagram, Voyager has the best performance in dispatching agents for short loop sizes. As we increase the number of agents launched, Concordia's performance improves considerably, due to its caching mechanisms. Furthermore, Concordia is very robust, even in cases of heavy network load. In contrast, we noticed that Voyager and Aglets crashed occasionally when we dispatched more than 600 agents in an experiment, and the network was heavily loaded. From Table 2 we can see that a Concordia place can dispatch 25.6 agents per second, whereas Aglets and Voyager can send only 5.36 and 8.3 agents per second, respectively.

4.2 Inter-agent Communication: MSG-1W, MSG-2W, SYNCH, and MSG-MA

The *MSG-1W* benchmark measures the elapsed time for sending non-blocking messages from one agent to another. For this benchmark we employ two mobile agents located at two different hosts in the same LAN. The first agent sends a number of messages to the second; there is no explicit acknowledgment of receipt from the second agent. We measure the time it takes to send 1 to 1000 messages of equal, minimal size.

To implement *MSG-1W* we employ the `OneWay` method of Voyager. In particular, a Voyager agent sends a message to a destination agent via the destination-agent's local "proxy." The message consists of the remote agent's name, the name of the method that will be invoked upon receipt of this message by the destination agent, and the arguments that will be passed to this method. The `OneWay` method does not return a reply and is non-blocking. Voyager employs standard Java serialization to transport messages across the network. In Aglets we implement *MSG-1W* with the `sendAsyncMessage()` method, which is invoked on the remote-agent's proxy that serves as a message gateway for the Aglet. Here, the message is an object. On the other hand, Concordia uses *events* to implement message-passing: events are sent by the dispatching agent to an Event Manager through the `postEvent()` method. The receiving agent must register with that Event Manager as well, to listen for and receive particular events. Examples of message-passing implementation are given in [5].

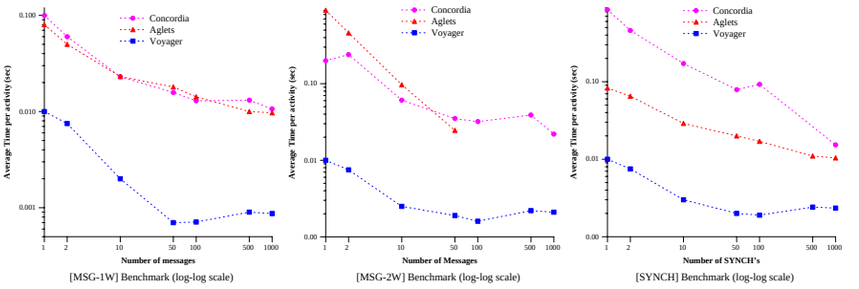

Fig. 3. MSG-1W, MSG-2W & SYNCH: Average time measurements

Table 3. MSG-1W, MSG-2W, SYNCH and MSG-MA: Peak and sustained rates.

Platform	MSG-1W		MSG-2W		SYNCH		MSG-MA	
	Peak (msg/sec)	Sustained	Peak (2wmsg/sec)	Sustained	Peak (synchs/sec)	Sustained	Peak (agnt-round trips/sec)	Sustained
Concordia	77.39	73.2	31.35	20.2	16.03	14	12.147	2
Aglets	102.94	102.94	10.3	8.13	96.15	92	4.93	4.9
Voyager	1428.57	1146.78	625	476.19	526.32	413	9.38	8.3

Figure 3 (left) presents the diagram of the average time per message for each experiment. From this diagram we can see that Voyager has the fastest messaging. Furthermore, its messaging is very robust, even under heavy network load. One-way messaging performance of Aglets and Concordia is similar; nevertheless, Aglets crashed occasionally when sending too many messages. From the left diagram of Figure 3 we also note that the average time to send a message decreases with respect to the number of messages dispatched during each experiment. This figure is stabilized for larger loop sizes. In Voyager and Aglets this happens because, after the first message is sent to the remote agent, all involved classes are installed in the caches of both places participating in the message-exchange. Consequently, the "initiation" overhead incurred by subsequent messages is minimal. In Concordia, the dispatch of repeated messages from one agent toward another, via an Event Manager, requires only one connection to the Event Manager. As we send more messages, the connection overhead is amortized across all messages.

Table 3 presents the peak and sustained rates for message-dispatching. A Voyager agent can send 1146.78 messages per second, whereas the capacity of Concordia and Aglets are 73.2 and 102.94 agents per second, respectively.

The **MSG-2W** benchmark measures the time it takes to send non-blocking messages from one agent to another, with asynchronous acknowledgments of receipt. To this end, we use two agents located at two different hosts in our LAN. The first agent sends non-blocking messages to the second; upon arrival of a message, the recipient-agent immediately replies back to the sender, acknowledging the receipt. To this end, we invoke the `sendFutureMessage()` method in Voyager and the `future()` in Aglets. We measure the time it takes to send 1 to

1000 messages and receive the respective acknowledgments. In all experiments we use messages of equal, minimal size. As expected, Voyager exhibits the best performance, with minimal fluctuation with respect to the number of dispatched messages (see Figure 3, middle). Concordia and Aglets have comparable performance when dispatching continuously up to 50-60 messages. For larger message numbers, Aglets crash. This explains the very small rates reported for Aglets in Table 3.

The **SYNCH** benchmark measures the time it takes to perform a synchronization between two agents; the synchronization operation is implemented with the exchange of two messages. To this end, we place the agents at two different places (hosts) in the same LAN. One agent sends a message to the other and gets blocked until it receives a reply. The second agent waits for incoming messages; upon receiving a message, it replies back. We use the `Synch()` method in Voyager and the `sendMessage()` method in Aglets. We conducted this "ping-pong" experiment from 1 to 1000 times. For each experiment, we measured the total elapsed time it takes to complete all synchronization activities. Figure 3 (right) presents our measurements. In agreement with the *MSG-1W* and *MSG-2W* benchmarks, Voyager exhibits a synchronization capacity significantly higher than Concordia and Aglets. Furthermore, it achieves a synchronization rate (number of SYNCH's per second) which is practically constant with respect to the number of the ping-pong operations performed.

As we can see from Table 3, Voyager agents are capable of conducting 413 synchronizations per second on the same LAN. Aglets come second in the synchronization capacity (92 SYNCH's per second, sustained) and Concordia achieves only 14 SYNCH's per second, sustained. We believe that Voyager outperforms Concordia and Aglets due to its low overhead of message initiation. This is also the reason why in Voyager the peak rate of SYNCH's is reached for small loop-sizes, and does not drop significantly for larger loop-sizes. It is interesting to note that the implementation of a blocking-message exchange in Aglets is much more efficient than the implementation of messaging with asynchronous acknowledgments, and that its performance is comparable to the performance of one-way messaging with no acknowledgment.

The **MSG-MA** benchmark measures the overhead that arises when two places (hosts) interact via a messenger agent; both hosts reside in the the same LAN. To implement this benchmark, we create an agent in the first place and set its itinerary so that the agent moves to the second place and then returns back. Upon return, the same agent is re-dispatched and retracted for a number of times. Our experimental parameter is the total number of round-trips performed by the messenger agent. We conduct experiments for 1 to 1000 round-trips, and measure the total elapsed time. We present our measurements in the left diagram of Figure 4. Table 3 summarizes the peak and sustained rates as shown in Figure 4 for the average time of messenger round-trips.

As we can see from Figure 4 (left), Concordia and Aglets exhibit better performance for one and two round-trips. Nevertheless, the average time per round-trip in Voyager drops much faster as we increase the number of round-trips. The same figure for Aglets is stabilized after 10 round-trips. Consequently, Voyager exhibits the best performance for larger numbers of round-trips (over

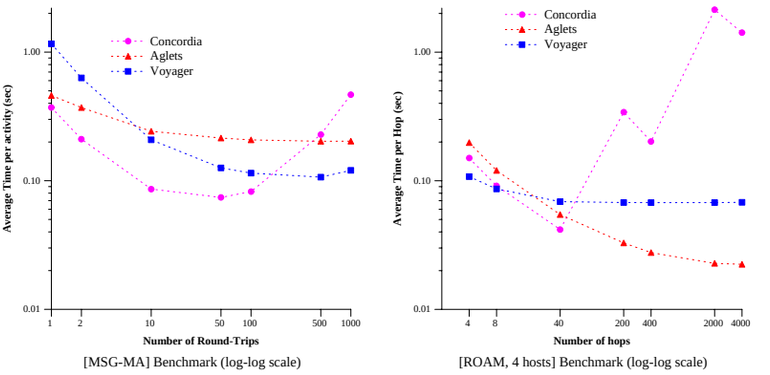

Fig. 4. MSG-MA and ROAM: Average times.

500). It is interesting to note that the average delay of a messenger-agent's round-trip in Concordia increases with the number of round-trips. We believe this is a side-effect of the agent-roaming implementation in Concordia: every time an agent has to move to another host, a Destination object must be added to the agent's *Itinerary*, in order to determine its next move. The Itinerary is a data structure separate than the agent, maintained at a different location than the agent itself. The Itinerary is composed of a list of *Destination* objects [13]. Each Destination indicates the place (host) to which the agent is expected to travel, and the name of the method that the agent will execute upon arrival to that place. In our experiments for *MSG-MA* we employ a messenger agent that travels numerous times back and forth between two places.

In contrast to Concordia, an agent in Voyager or Aglets can be re-launched to a new destination, upon arrival to some place. To this end, a method can be called by the agent to determine its next destination. In particular, in Aglets we use the `dispatch` method to send an Aglet to a remote location. This location is passed as argument to the dispatch method (`Aglet.dispatch (URL destination)`). Upon arrival to its destination, the Aglet is pulled back to its original place with the `retractAglet()` method. In Voyager, we use the `Mobility.of()` method to obtain the mobility facet of an agent and invoke the `moveTo()` method of its `IMobility` interface. To pull the agent back, we call again `moveTo()`.

5 Micro-Kernels: ROAM, PROXY, and FORW

Due to space limitations, in this section we focus on three application frameworks only: Roaming MA, Proxy-Server, and the Forwarding pattern; early experimentation with other application frameworks (C/S, C/A/S, and C/I/S) has been presented in [12]. Accordingly, we define the micro-kernels presented in Table 4.

The **ROAM** micro-kernel investigates the overhead incurred by an agent that roams from place to place in a network. To implement this benchmark we

Table 4. Micro-kernels.

Name	Description
ROAM	Captures the overhead of a roaming agent.
PROXY	Captures the performance of a proxy-agent serving requests from a number of client-agents.
FORW	Captures the overhead of a forwarding agent, residing at a place, receiving and re-directing incoming agents

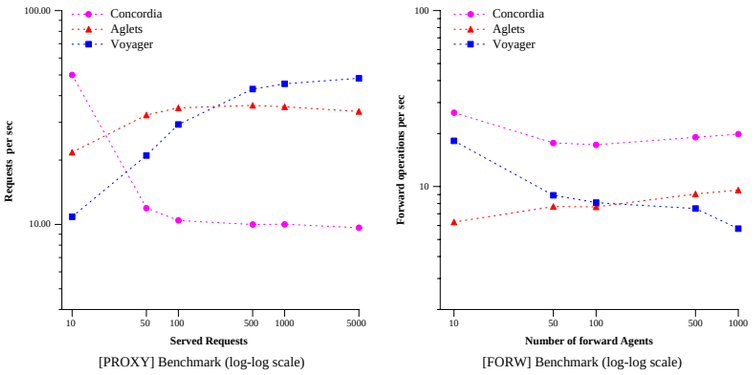

Fig. 5. PROXY & FORW: Service rates.

create an agent at a place, and set its itinerary so that it visits a number of places and then returns back to its place of origin. We dispatch this agent and measure the total time it takes to complete its trip. The itinerary is fixed before the agent starts its voyage. It should be noted that the implementation of agent mobility in *ROAM* is different than that in *MSG-MA*, for the Aglets platform: upon successful arrival of an Aglet to a new place, the `onArrival()` method is invoked automatically. We have overwritten `onArrival` so that it dispatches the Aglet to its next destination. Experimental parameters of this benchmark are the number of hops taken by the roaming agent before coming back to its origin place, and the different places it visits (in its journey, an agent can visit one place multiple times).

In Figure 4 (right), we report measurements taken when an agent roams four different places (including its starting point), making 4 to 4000 hops totally. As we can see from this diagram, the average time per hop in Voyager is practically constant with respect to the total number of hops. Aglets average performance improves as we increase the number of hops; obviously a side-effect of an initial high overhead incurred when an agent visits a place for the first time, which is amortized by the reduced cost of subsequent re-visits. The performance behavior of Concordia worsens for longer agent voyages, in concordance with the *MSG-MA* micro-kernel. We believe this is a side-effect of the handling of itineraries in Concordia.

The **Proxy-Server** model is an extension of the Client-Agent-Server model with the "Agent" accepting connections from many clients and forwarding requests to more than one Servers. This scenario arises in cases where an agent is dispatched to the "edge" of the network to act as proxy. This agent receives incoming client requests and forwards them to appropriate servers, optimizing the communication of clients and servers, caching server replies, etc. The *PROXY* micro-kernel investigates the performance of the Proxy-Server model when implemented on top of a MA middleware platform. To this end, we use a mobile agent as proxy that mediates between several clients and servers. The proxy agent waits for request messages from agent-clients located at different hosts. Whenever it receives a message, it inspects the request message and forwards it to the appropriate server. Upon receipt of a request, a server replies directly to the client that sent it. Upon receipt of the server's reply, the client issues a new request, following the same procedure.

The *PROXY* benchmark is parameterized with respect to the number of clients and servers involved in our experiments, and the total number of requests handled by the proxy-agent. Here, we report measurements from one experiment involving three server-agents and twelve client-agents. We measure the time it takes the proxy-agent to receive and forward incoming 1 to 5000 requests to the appropriate servers. Moreover, we report the rate of request-handling achieved by the proxy-agent. Figure 5 (left) presents a diagram with our measurements. Further experiments are reported in [5]. As we can see from this diagram, the performance of each MA platform converges to a certain sustained rate of requests served per second. In the twelve-client case, the Concordia, Aglets and Voyager proxy-agents can handle 9.65, 33.7 and 48.25 requests per second, respectively.

The **FORW** micro-kernel represents an implementation of the Forwarding pattern. This micro-kernel seeks to capture the overhead that arises when a mobile agent receives incoming mobile agents and re-routes them to other places. To this end, we use a "forwarding" mobile agent parked at a particular place. The forwarding agent "listens" for incoming agents; upon arrival of a new agent, the forwarding agent directs it to another place. The *FORW* benchmark is parameterized with respect to the total number of mobile agents handled and re-routed by the forwarding agent. We use one dispatching and one destination place only and measure the total elapsed time from the receipt of the first agent to the dispatch of the last one from the forwarding agent. The forwarding capacity of each MA platform converges to a certain sustained rate of requests served per second (see Figure 5, right). Voyager and Aglets can forward 19.84 and 9.54 agents per second respectively, whereas the corresponding number for Concordia is 5.76.

6 Conclusions

In this paper, we introduced a hierarchical framework for the quantitative performance evaluation of mobile-agent middleware platforms. We specified this framework as a hierarchy of benchmarks designed to enable the performance characterization of key components of MA middleware, and analyzed the performance of important classes of MA applications. This hierarchy is defined along a

number of dimensions pertinent to MA systems: the basic elements of MA platforms, distributed computing models of relevance, expected application frameworks, the context of MA execution, and expected workload characteristics. We proposed a set of micro-benchmarks and micro-kernels to implement the lower two levels of our benchmark hierarchy. We implemented these benchmarks in three of Java-based, mobile-agent middleware environments (Mitsubishi's Concordia, IBM Aglets, and Objectspace's Voyager). We presented results from experiments conducted to validate our framework and compare the mobile-agent middleware environments quantitatively.

To our knowledge, our framework provides the first structured and layered approach for analyzing the performance of MA middleware quantitatively (extensive coverage of related work is given in [5]). Experiments with our micro-benchmark and micro-kernel 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 of basic performance properties of MA systems. Furthermore, they provide a solid base for the assessment of the design choices made by middleware developers, from a performance perspective. For instance, our experimental results show that caching of classes and object re-use can lead to significant performance improvements and, therefore, call for a more in depth study of techniques for their easy integration and optimization in MA middleware design and application development. Raw performance data show that agents cannot sustain the loads expected to arise in Internet middleware where places and agents could face workloads on the order of hundreds or thousands of requests per second, in the form of incoming messages, agents, etc. Furthermore, all examined platforms exhibit problems of robustness and performance scalability under high-loads, which are issues of critical importance for Internet services and applications. In such cases, places and agents should incorporate support for memory and resource management, request scheduling, recovery, high-performance execution of bytecodes, etc. Last, but not least, our approach can provide a basis for the development of performance prediction models and tools for mobile-agent systems.

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. T. Budd. *Understanding Object-Oriented Programming with JAVA*. Addison-Wesley, 2000.
4. D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.
5. M. Dikaiakos, M. Kyriacou, and G. Samaras. Benchmarking Mobile-agent Systems. Technical Report TR-01-2, Department of Computer Science, University of Cyprus, May 2001.

6. M. Dikaiakos and G. Samaras. A Performance Analysis Framework for Mobile-Agent Platforms. In Wagner and Rana, editors, *Infrastructure for Agents, Multi-Agent Systems, and Scaleable Multi-Agent Systems*, volume 1887 of *Lecture Notes in Computer Science*. Springer, 2001.
7. G. Glass. Overview of Voyager: ObjectSpace's Product Family for State-of-the-Art Distributed Computing. Technical report, ObjectSpace, 1999.
8. R. Koblick. Concordia. *Communications of the ACM*, 42(3):96–99, March 1999.
9. D. B. Lange and M. Oshima. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–91, March 1999.
10. Mitsubishi Electric ITA. *Concordia Developer's Guide*, October 1998. http://www.meitca.com/HSL/Projects/Concordia.
11. ObjectSpace. *ObjectSpace Voyager, General Magic Odyssey, IBM Aglets. A Comparison.* ObjectSpace, June 1997.
12. G. Samaras, M. 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.
13. D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet. Concordia: An Infrastructure for Collaborating Mobile Agents. *Lecture Notes in Computer Science*, 1219, 1997. http://www.meitca.com/HSL/Projects/Concordia/.
14. M. Woodside. Software Performance Evaluation by Models. In C. Lindemann G. Haring and M. Reiser, editors, *Performance Evaluation: Origins and Directions*, LNCS. State of the Art Survey, pages 283–304. Springer, 1999.