Benchmarking Mobile-Agent Systems

Marios Dikaiakos Melinos Kyriakou George Samaras
Dept. of Computer Science
University of Cyprus
P.O. Box 20537, 1678 Nicosia
Cyprus

November 28, 2001

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

With the emergence of Internet as a world-wide infrastructure for communication and information exchange, Internet-based distributed applications have gained remarkable popularity. One of the most promising approaches for developing such applications is the Java-based mobile-agent paradigm [13, 28]. Mobile agents (MA) are being used already in a variety of Internet-based distributed computing applications: Web databases [17], cooperative environments [6], information-gathering systems [8], electronic commerce systems

[31], and so on. In that context, a distributed application can be considered as a dynamic group of agents working in coordination to accomplish some goal. The employment of MA technologies for the development of next-generation Internet systems opens numerous research problems related to programming APIs and tools, security, fault-tolerance, design paradigms and programming techniques, communication, intelligence, scalability and performance. 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 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 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 [7], Mitsubishi's Concordia [11] and ObjectSpace's Voyager [9]) 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 hierarchical performance analysis framework. Section 3 introduces the suite of "micro-benchmarks" that we propose to implement the lower level of our performance evaluation approach; we present and discuss several experimental results from the implementation of these benchmarks. Section 4 presents our implementation of the second layer of our hierarchical framework, that is, a set of "micro-kernel" benchmarks; we provide and discuss experimental results from the implementation of these benchmakrs as well. Conclusions and future work are given in Section 5, and related work is presented in Section 6.

2 A Framework for Investigating Mobile-agent Performance

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 [30]. Model development is performed in a "top-down" manner, starting from high-level structure and moving towards 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 systems 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: this makes it hard to define and determine unequivocally the condition of a particular MA-system at a particular moment.
- The complex architecture of MA platforms: simple metrics used for the performance characterization of typical parallel and distributed systems are not adequate for isolating performance problems of MA systems. Further investigation and definition of more complex metrics is necessary.
- The variety of distributed computing (software) models that are applicable to mobile-agent applications dictates the design of different experiments, tailored to the alternative software models of interest.
- Construction of simple and portable benchmarks for experimentation is difficult due to the diversity of operations found in MA platforms.
- The presence of mobility, which makes it hard to establish a concise and stable representation of system resources that affect MA performance, due to the dynamic nature and agile configuration of MA systems.
- The additional complexity introduced by issues that affect the performance of Java, such as run-time compilation, memory management, garbage collection, etc.

To cope with the complexities of MA performance analysis we propose the adoption of a hierarchical approach inspired by the structure of MA-based applications. This structure

is determined by:

1. The MA platform adopted to program a particular application. Mobile-agent platforms (such as [3, 4, 9, 11, 28]) are middleware systems with a programming interface, which exposes to the programmer a set of core functionalities. Typically, these functionalities include support for object mobility (transportation and location services), communication between objects, security, fault-tolerance etc. Various MA platforms differ in terms of their functionality, programming interface and performance characteristics, all of which are influenced by underlying implementation details.

2. Higher level abstractions representing the design choices made by software developers for a particular application. These abstractions are implemented with the programming interface of the MA platform at hand, and usually correspond to generic MA 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*.

2.1 Basic Elements

We define as *Basic Elements* of mobile-agent platforms, a 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 [7, 4, 9, 11, 28]:

- *Agents*, 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:

  1. Creating an agent at a local or remote place.
  2. Dispatching an agent from one place to another.
  3. Receiving an agent that arrives at some place.
  4. Communicating information between agents via messages or messenger agents.
  5. Synchronizing the processing of two agents.
  6. Locating an agent on the move, etc.

2.2 Application Frameworks

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" [5]. 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 MAs 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 (see Figure 1);

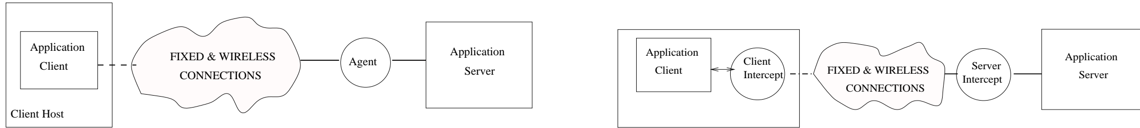


Figure 1: The Client-Agent-Server and Client-Intercept-Server Models.

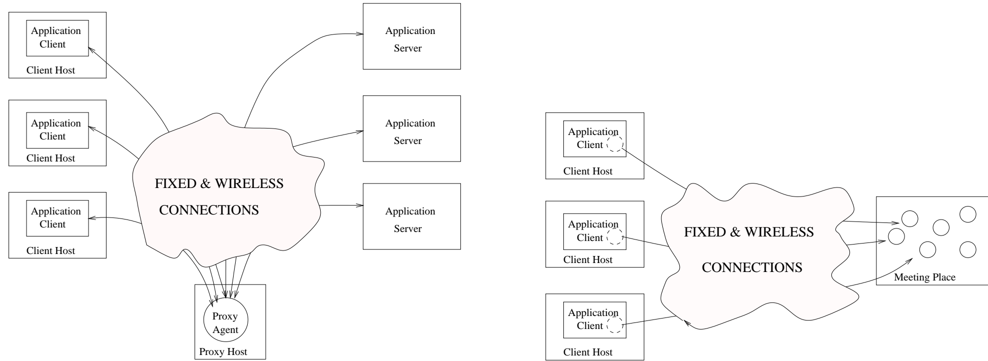


Figure 2: The Proxy-Server Model and the Meeting Pattern.

more details on these models are given in [21, 22]. Additional application frameworks correspond to the *Roaming Mobile-Agent* Model, and the *Forwarding* and *Meeting* agent-design patterns (see Figure 2). The Roaming MA model 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, 7]. 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.

2.3 The 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 3). 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.

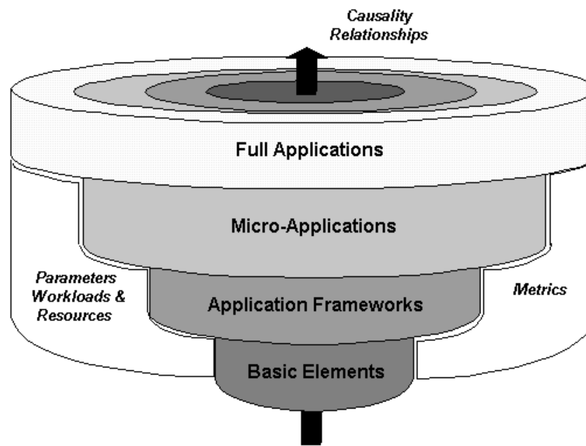


Figure 3: The Hierarchical Performance Analysis Framework.

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 at 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.

2.4 Benchmarking Mobile-Agent Systems

To apply our hierarchical performance analysis 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 3. 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) implemented 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 introduce places with full application functionality and employ synthetic workloads complying to the *TPC-W* specification [27].

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 [21]; 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 on a LAN, unless mentioned otherwise. In future experiments, we plan to incorporate setups including wireless Ethernet and connectivity over WANs. It should be noted that the current experimental setup is expected to provide us with optimistic estimates of mobile-agent performance capacity.

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 long period of time. Additional, metrics are measured in certain micro-kernels and will be described later.

3 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.

These micro-benchmarks involve places located at different computing nodes, agents with the minimum functionality that is required for carrying out the behaviors studied, and

Name	Description
CL	Captures the overhead of agent-creation locally within a place.
CR	Captures the overhead of agent-creation at a remote place.
AL	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.

Table 1: Definition of Micro-benchmarks.

messages carrying minimal information between agents. Table 2 presents the parameters and metrics for our benchmarks.

3.1 CL

With this micro-benchmark we study the overhead of agent-creation. To this end, we create 1 to 1000 agents and measure the total elapsed time. An excerpt from the implementation of this benchmark on Concordia, is given in Figure 4. Measurements are presented in Figure 5. The left diagram of Figure 5 presents the total times measured; the right diagram reports the average time it takes to create an agent in each experiment. From both diagrams 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 for a single agent creation, Aglets outperform Voyager.

As we increase the number of agents created from 2 to 1000, the average creation-time per agent drops faster in Voyager than Aglets. The improved "scalability" of Concordia and Voyager are 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 [14, 15]. Another remark is that the time it takes to create an agent, drops with the increase of loop size. 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. Finally, in Table 3, we present the agent-creation capacity of the three platforms (peak and sustained).

Name	Parameters						Metrics			
	Loop Size	Operating System	Place Config.	Channel Config.	Agent Size	Msg. Size	Total Time	Average Time	Peak Rate	Sustained Rate
CL	√	√	√	-	√	-	√	√	√	√
CR	√	√	√	√	√	-	√	√	√	√
AL	√	√	√	√	√	-	√	√	√	√
MSG-1W	√	√	√	√	-	√	√	√	√	√
MSG-2W	√	√	√	√	-	√	√	√	√	√
SYNCH	√	√	√	√	-	√	√	√	√	√
MSG-MA	√	√	√	√	√	-	√	√	√	√

Table 2: Micro-benchmark Parameters and Metrics.

```
public void createLocal()
{
  int j;
  LocalAgent agents[] =new LocalAgent[10000];
  System.out.println("Starting Creation Locally....");
  starttime = System.currentTimeMillis();

   for (j=0;j<noofag;j++){

 try {
        System.out.println("Creating Agent:" + j);
       //Creating Agent
       agents[j] = new LocalAgent (j);

        //Sets its Itinerary
        Itinerary itinerary = new Itinerary();
               itinerary.addDestination(new
               Destination(getItinerary().
               getCurrentLocation().getDestinationHost(),"method"));
        agents[j].setItinerary(itinerary);

              //Sets its Codebase
                agents[j].setHomeCodebaseURL(getHomeCodebaseURL());

             } catch (Exception e)
         {
     System.out.println(e.getMessage());
      e.printStackTrace();
         }
           switch (j) {
          case 0 :
          case 1 :
                 .

  }
  //System.out.println("Finish.Agents Created...." + j);
  calcTime();
```

Figure 4: CL. Code Example in Concordia.

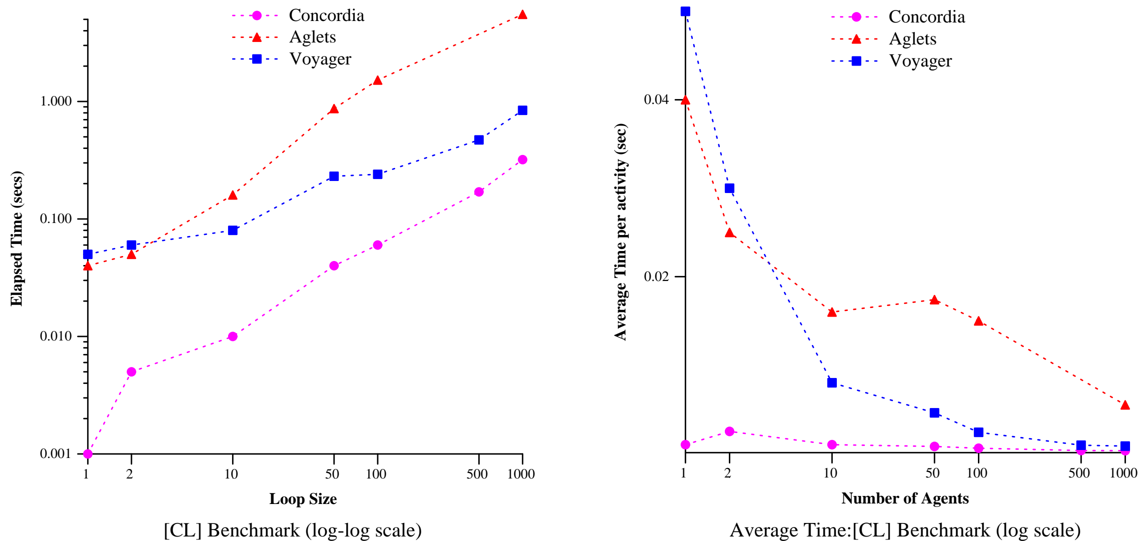


Figure 5: CL. Total and average timings for local agent creation.

Platform	CL		CR		AD	
	Peak	Sustained	Peak	Sustained	Peak	Sustained
	(agents/sec)		(agents/sec)		(agents/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

Table 3: CL, CR and AD. Peak and sustained rates.

3.2 CR

This 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 at a remote place that resides in the same LAN. We time the overall overhead of agent creation at the origin place. To achieve remote creation of agents, the remote place needs to have the necessary classes available at its site 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:

- Under Concordia, a messenger agent migrates from the origin place to another place in the same LAN. 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.
- A Voyager agent at the remote place can load classes from other locations on demand. To this

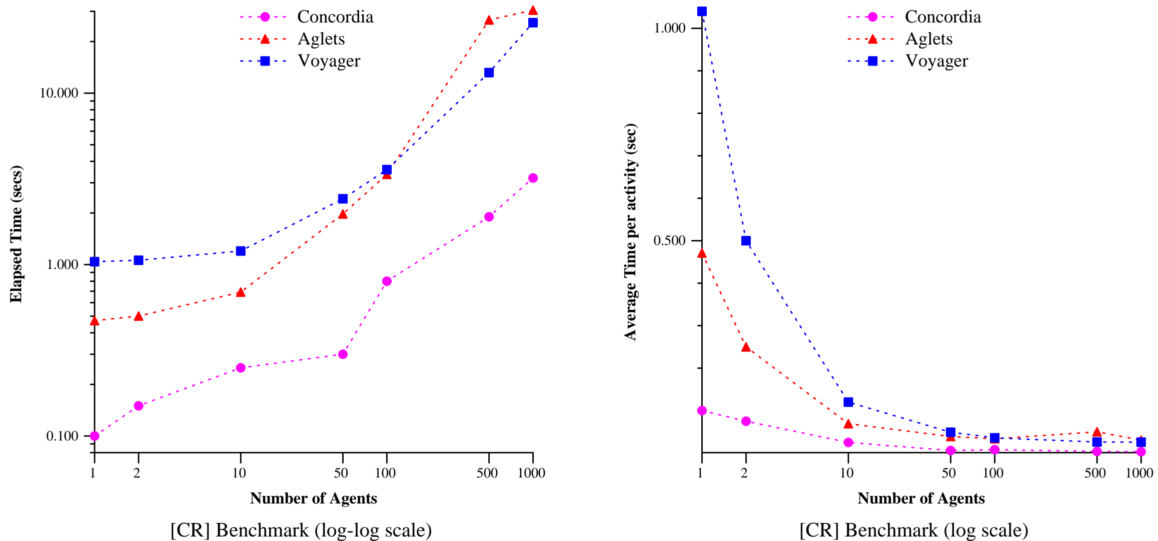


Figure 6: CR. Total and average timings for remote agent creation.

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. We employ this mechanism to implement remote creation of agents in Voyager. An excerpt of the corresponding code is given in Figure 7.

- An Aglet can load a remote class on demand from a remote Tahiti server, which is the agent execution environment of Aglets (place). 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.

Diagrams in Figure 6 show our measurements for the *CR* benchmark. As we can see, Concordia and Aglets have better performance than Voyager 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 3. 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

```
Static public void main(String args[])
{
         //URL where Agents will be created
         String RemoteServerAdress = "cs283.ucy.ac.cy:8000";
         String codebase = "bench.creater.Agent";
         IAgent agent[]=new Iagent[10000];

        try{

         Voyager.startup("7000");

          //Enable this server so that other servers to load classes from this one
         ClassManager.enableResourceServer();

           //Start The Timer
StartTime = System. CurrentTimeMillis();

          for(int i=0;i<agno;i++)
          {
          //Creating Agents Remotely
          agent[i]= (Iagent)Factory.create(codebase,RemoteServerAddress);

          switch (i) {
           case 0 :
  case 1 :
   .
   .
  case 4999 : { calcTime(); }  }
  }
          }calcTime();

   } catch (Exception e) { System.out.println(e);}
   //Shutdown the Server
   Voyager.shutdown();
```

Figure 7: CR. Code example in Voyager.

their improved robustness. In contrast, Aglets performance drops for very large numbers of created agents.

3.3 AD

This benchmark measures the overhead of dispatching mobile agents to a remote place in a LAN. For our experiments, we create 1 to 1000 mobile agents. Then, we dispatch these agents to the remote place. We measure *only* the time of the dispatch operation. An example of this benchmark's implementation on Aglets is given in Figure 8. Diagrams in Figure 9 report timing measurements for the *AD* benchmark. As shown in Figure 9 (left), 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 3 we can see that a Concordia place can dispatch 25.6 agents per second, whereas Aglets and Voyager can dispatch only 5.36 and 8.3 agents per second, respectively.

In Voyager, a dispatched agent and all of its non-transient parts are copied to the destination place using Java serialization, instead of pass-by-reference features like `java.RMI.Remote`. The new

```
public void create_launchAgents()
    {
        String ClassName = new String("microbenchmarks.dAglet");
        proxies=new AgletProxy[agentsno];

      //We first create the Aglet Agents locally
      for(int i=0;i<agentsno;i++)
        {
           try
            {
                proxies[i]= getAgletContext().createAglet(getCodeBase(),ClassName,null);
            }
            catch (Exception e)
            {
                System.out.println ("Unable to create "+I+" Agent");
            }
        }

      //Start timer just before begin launching Agents
      long StartTime = System.currentTimeMillis();

       for(int i=0;i<agentsno;i++)
        {
           try
            {   //We finally launch Agents one by one
                proxies[i].dispatch(new URL("atp:Destination Server Url"));

            }
            catch (Exception e)
            {
                System.out.println("Unable to Launch "+i+" Agent");
            }
          switch (i) {
      case 0 :
          case 1 :
                 .
      case 4999 :{ calcTime(); }   }
```

Figure 8: AD. Code example in Aglets.

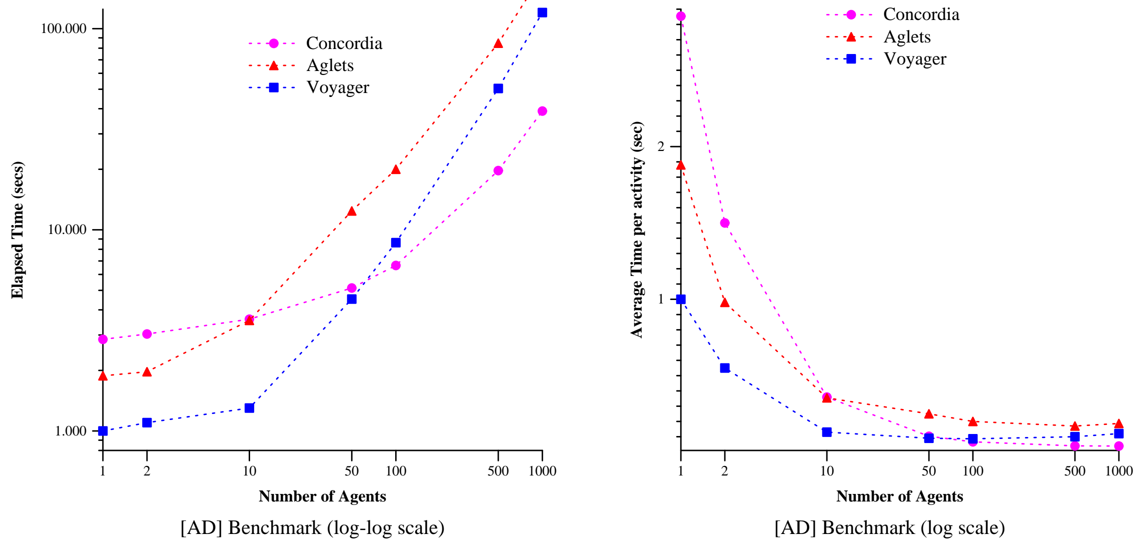


Figure 9: AD. Total and average timings for agent dispatch.

address of the agent and all of its non-transient parts are cached in the original place, whereas the agent-object is released and can be reclaimed by the garbage collector.

Transportation of Concordia agents is based on RMI. Concordia transfers an image of the agent only; all other objects are transferred and cached on a need-to-use basis. Once the necessary classes are transferred to the destination, no other transfer is required since subsequent agent transportations use agent classes from the cache.

Finally, Aglets use a special protocol in agent transportation called ATP. In every transportation of an agent, Aglets transfer all the objects reachable from the agent object, even if they are not needed in the destination place.

3.4 MSG-1W

For our messaging benchmarks we measure the elapsed time of message-exchange between agents located at different places. To this end, we create two agents in one place and dispatch one of these to a remote place in the same LAN. Then, the two agents start communication by exchanging messages. We measure only the delay involved in message-exchanges. It should be noted that the goal of the messaging-benchmarks is not to investigate the overhead of locating a remote agent before engaging in communication with it.

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. In all experiments we use messages of equal, minimal size.

Table 4 summarizes the various interfaces provided by the three MA platforms for message-passing. 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 (an example of message-passing implementation in Voyager is given in Figure 13). 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, which serves as a message gateway for the Aglet. Here, the message is an object. In contrast, 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. Figure 10 shows an excerpt from the implementation of the *MSG-1W* benchmark

```
public void sendEvents()

    {
         int i=0;

   String fullRMIURLofRemoteEventManager =
   New String(EventManagerConnection.EventManagerURL("EventManager"));

        StartTime = System.currentTimeMillis();

        Try {
            //Connects to an Event Manager
            makeEventHandler(false);
            makeEventManagerConnection(
                               fullRMIURLofRemoteEventManager,false);

  for (i=1; i<=noofmsg; i++)
 {
   try { //Posting an event
                                 postEvent(new SendMsg(i));
                              switch (i) {
                   case 1 :
                          .
                case 5000 :
           System.out.println("\tPostedMessage"+i);calcTime();}  }
        } catch (Exception e) {
            System.out.println("Error: " + i + "\n" + e);
        }
```

Figure 10: MSG-1W. Code example in Concordia using Events.

on Concordia.

Figure 11 presents the diagrams of the total elapsed time for each experiment, and the corresponding average time per message. From both diagrams 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 right diagram of Figure 11 we 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 5 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.

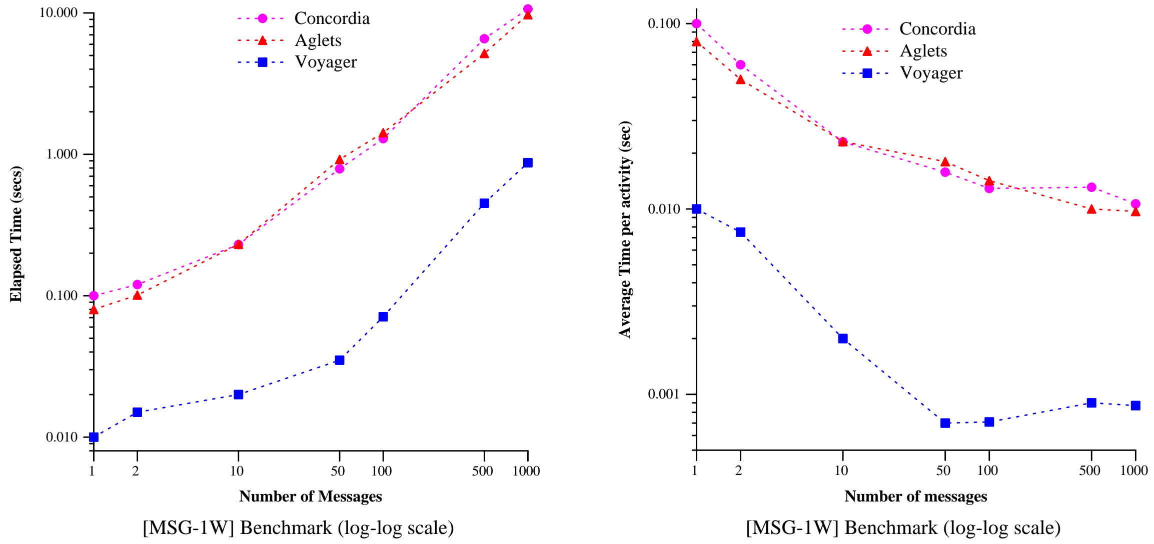


Figure 11: MSG-1W. Total and average times for non-blocking messaging.

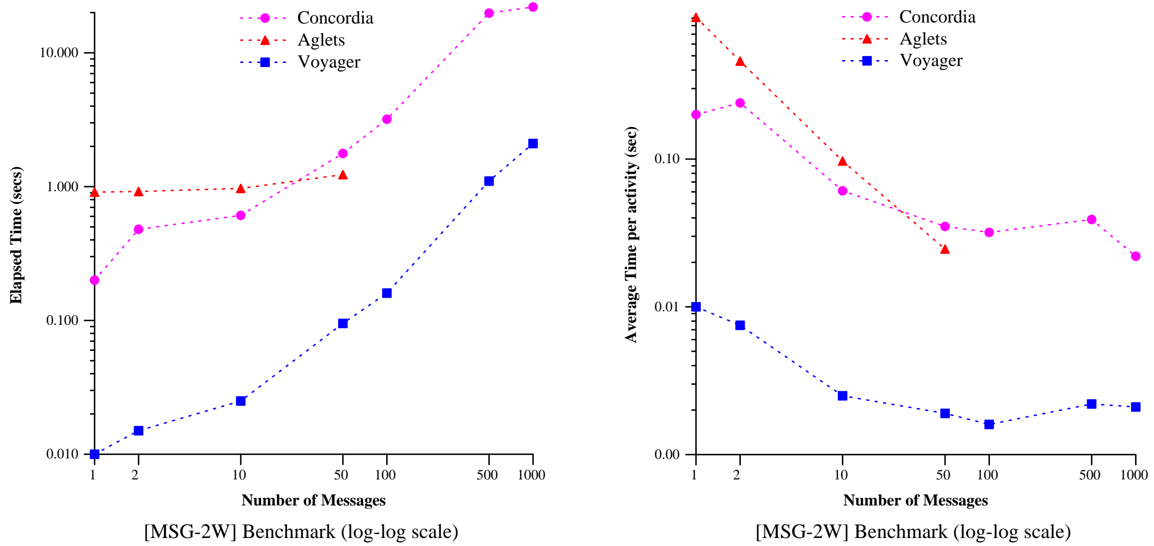


Figure 12: MSG-2W. Total and average times for non-blocking messaging with ack.

Platform	Method	Description
Aglets	sendMessage()	Synchronous message (blocking for reply value)
	sendAsyncMessage()	Asynchronous message (blocking for ack)
	sendOneWayMessage()	Asynchronous message (non-blocking)
	sendFutureMessage()	Non-blocking; sender may ask for ack later
Concordia	PostEvent(new Event())	Non-blocking; sending event to Event Manager
Voyager	Sync()	Synchronous (blocking for ack)
	OneWay()	Asynchronous (no reply from destination).
	Future()	Non-blocking; sender may ask for ack later

Table 4: Methods for sending messages.

3.5 MSG-2W

This benchmark measures the time it takes to send non-blocking messages from one agent with asynchronous acknowledgments of receipt. To this end, we use two agents located at two different hosts in a 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. The sender receives and handles the acknowledgement. We measure the time it takes to send 1 to 1000 messages and receive the respective acknowledgements. In all experiments we use messages of equal, minimal size. Experimental results are displayed in Figure 12.

As expected, Voyager exhibits the best performance and has an average time per message that is constant with respect to the number of messages sent. 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 5.

We implemented the *MSG-2W* benchmark in Voyager using the "`OneWay()`" method, which allows an agent to send messages without blocking. Upon arrival of an acknowledgment message, the recipent-agent's proxy in the sender's place invokes a method to handle the acknowledgment without blocking the ongoing send operations of the sender. In Aglets, we employed the "`sendAsyncMessage()`" method. According to specifications, an Aglet implements an event-listener

Platform	MSG-1W (msg/sec) Peak	MSG-1W Sustained	MSG-2W (2wmsg/sec) Peak	MSG-2W Sustained	SYNCH (synchs/sec) Peak	SYNCH Sustained	MSG-MA (agnt-round trips/sec) Peak	MSG-MA 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

Table 5: MSG-1W, MSG-2W, SYNCH and MSG-MA. Peak and sustained rates.

which invokes a "`handle()`" method to process incoming acknowledgments. In practice, however, this feature did not work; at successive invocations of the `sendAsyncMessage`, the sender Aglet kept outgoing messages, dispatched them all together, and received all acknowledgements together, resulting to crashes when the total number of messages exceeded 63.

3.6 SYNCH

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. An excerpt of this benchmark's implementation in Voyager is given in Figure 13. 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 14 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 5, 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

```
package bench.syncmsg;

import com.objectspace.voyager.*;
import com.objectspace.voyager.message.*;
import com.objectspace.voyager.mobility.*;
import java.util.*;

public class Main {

    static Date start,stop;
    static int msgno = 10000;
    static String codebase = "bench.syncmsg.Agent";

    static public void main(String args[]){
        try{

            Voyager.startup("7000");
            ClassManager.enableResourceServer();

          IAgent agent = (IAgent)Factory.create(codebase,"//cs283.cs.ucy.ac.cy:8000");


          start = new Date();

          for(int i=0;i<msgno;i++)
          {

          Result reply = Sync.invoke(agent,"replymethod",new Object[]{"Sync Message"});
          String l = (String)reply.readObject();

          }
          calcTime();


        } catch (Exception e) { System.out.println(e);}

        Voyager.shutdown();
    }

     static void calcTime() {
     stop = new Date();
     long started = start.getTime();
     long finish = stop.getTime();
     int total = (int) (finish - started);
     System.out.println("Time needed to send msg(no reply): " + total);
 }


}
```

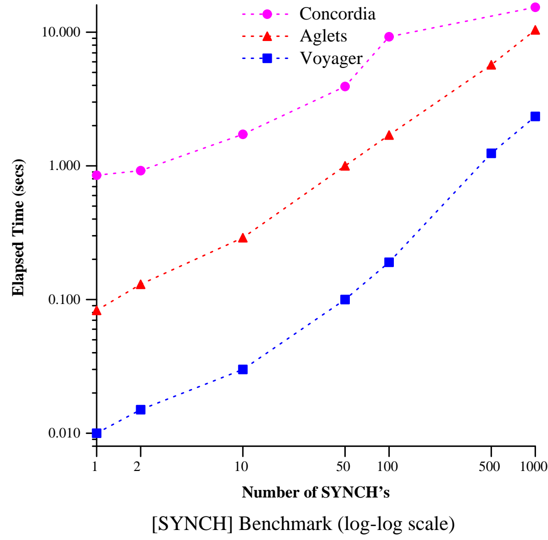
Figure 13: SYNCH. Code Example in Voyager.

one-way messaging with no acknowledgment.

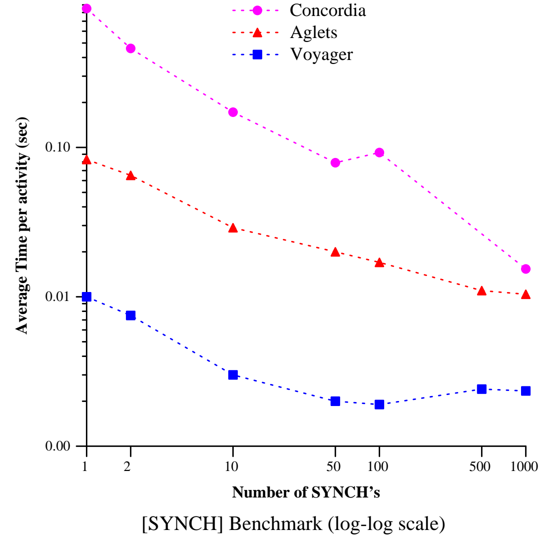
3.7 MSG-MA

The *MSG-MA* benchmark measures the overhead of using messenger agents to communicate information between two places (hosts) located in 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, we re-launch the agent. Our experimental parameter is the number of round trips performed by the messenger agent. We repeat this experiment for 1 to 1000 round trips, and measure the total elapsed time. Diagrams in Figure 15 present our measurements. Table 5 summarizes the peak and sustained rates as shown in Figures 15 for the average time of round-trips.

As we can see from Figure 15, 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 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

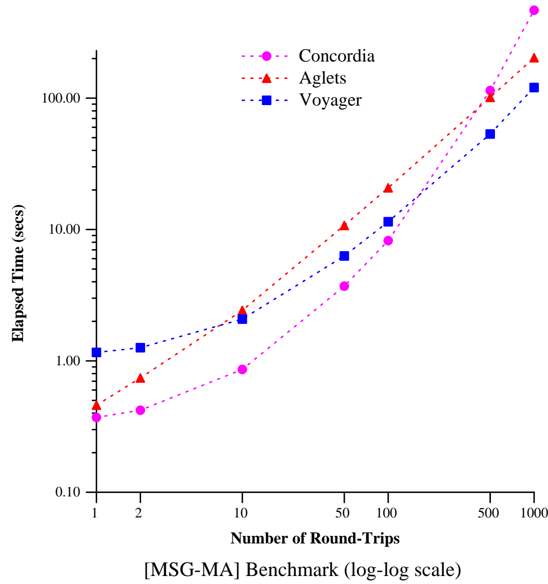


[SYNCH] Benchmark (log-log scale)

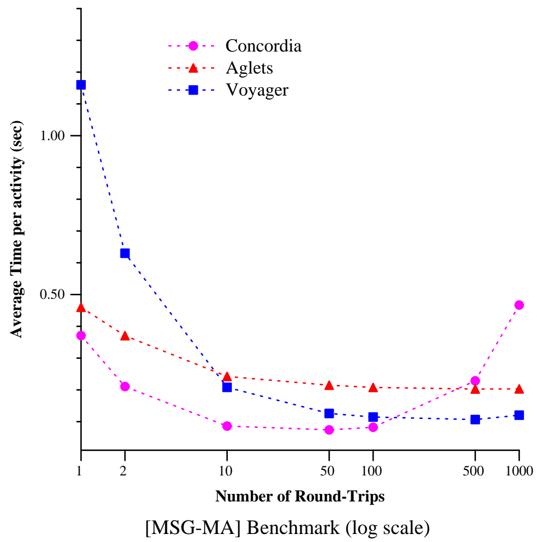


[SYNCH] Benchmark (log-log scale)

Figure 14: SYNCH: Total and average timings for message exchange.



[MSG-MA] Benchmark (log-log scale)



[MSG-MA] Benchmark (log scale)

Figure 15: MSG-MA: Total and average times vs. number of round trips.

Name	Description
ROAM	Captures the overhead of a roaming agent.
PROXY	Captures the performance of a proxy-agent serving a population of client-agents.
FORW-MSG	Captures the overhead of a forwarding agent that receives messages carrying requests, and re-directs them to servers.
FORW-MA	Captures the overhead of a forwarding agent that resides at a place, receiving and re-directing incoming agents.

Table 6: Micro-kernels.

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, and is maintained at a different location than the agent [29]; the Itinerary is composed of a list of *Destination* objects [29]. 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 (`public final 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`.

4 Micro-kernels

Due to space limitations, in this section we focus on three application frameworks: Proxy-Server, Roaming MA, and Forwarding pattern. Early experimentation with other application frameworks (C/S, C/A/S, and C/I/S) has been presented in [21], and will be examined further in future work. Accordingly, we define the micro-kernels presented in Table 6. In the following sections we present our experiments with the *ROAM*, *PROXY* and *FORW* benchmarks.

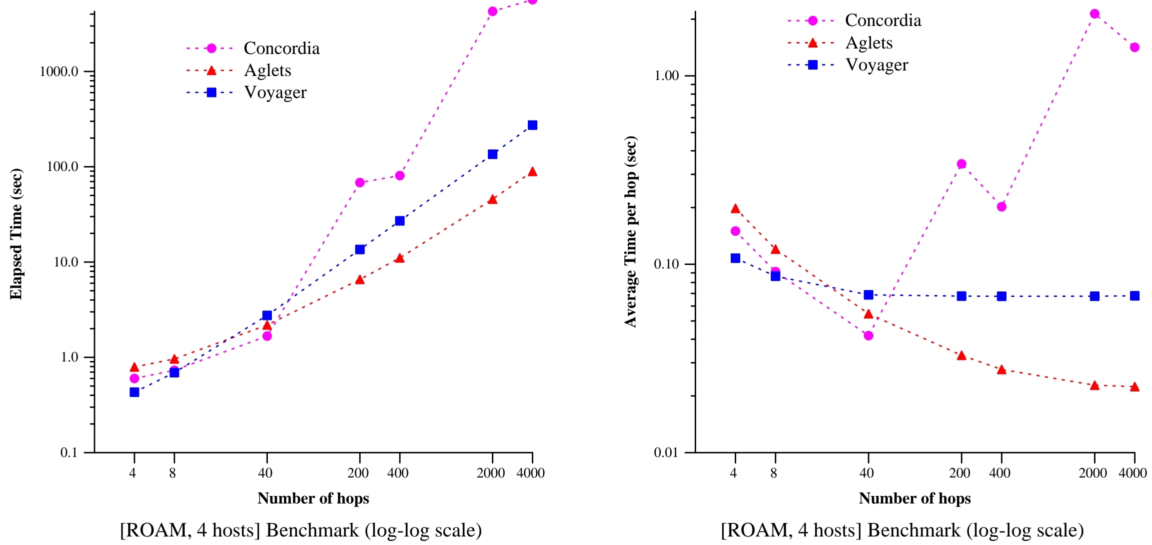


Figure 16: ROAM: Total and average times vs. number of hops.

4.1 ROAM

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 create an agent at a place and set its itinerary so that it visits a number of places, in a fixed trail, 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. For Aglets, the implementation of agent mobility in *ROAM* is different than that in *MSG-MA*: upon successful arrival of an Aglet to a new place, the `onArrival()` method is invoked automatically. We overwrote `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 16, we report measurements taken when an agent roams four different places (including its starting point), making in total 4 to 4000 hops.

As we can see from the right diagram of Figure 16, the average time per hop in Voyager is practically constant with respect to the total number of hops, resulting to a sustained speed of 14.7 hops per second. Aglets average performance improves as we increase the number of hops; obviously a side-effect of the initial high overhead incurred when an agent visits a place for the first time, which is amortized by the minimal cost of subsequent re-visits. This results to a sustained speed of 44.64 hops per second. 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

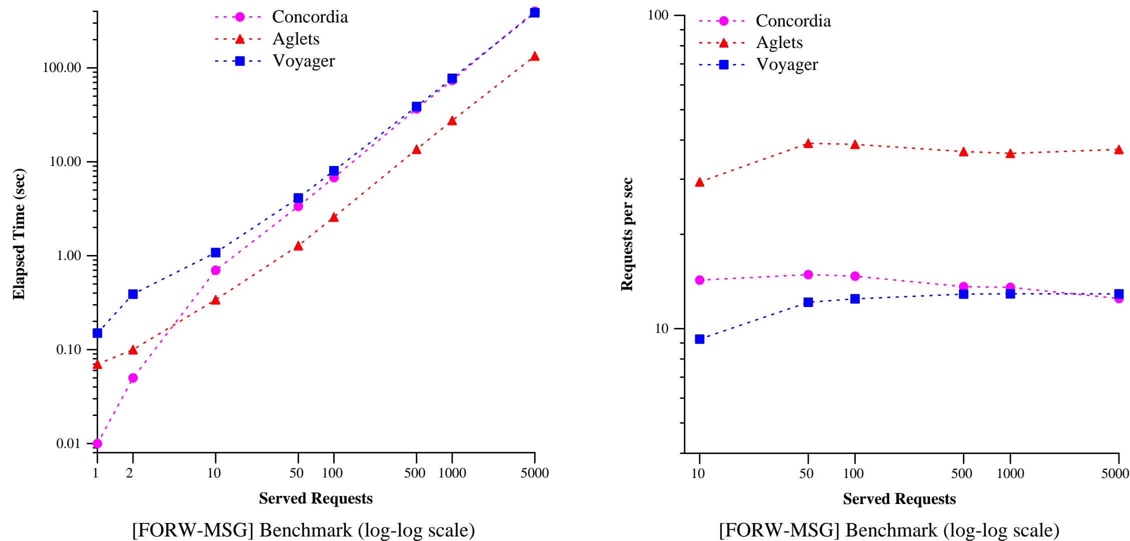


Figure 17: FORW-MSG: Total elapsed times and Service rates for 3 clients, 3 servers.

itineraries in Concordia. Under Concordia, we observe a sustained speed of 0.7 hops per second, for 4000 hops. The peak roaming speed of 23.92 hops per second is observed for a journey of 40 hops, total.

4.2 PROXY

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. The proxy listens for incoming messages carrying client-requests. Whenever it receives a message carrying a request, it inspects the request message and forwards it to the appropriate server. Whenever a server receives a request, it replies to the client that sent it through the proxy-agent. Upon receipt of the server's reply, the client issues a new request, following the same procedure.

In addition to the experimental parameters defined in Section 2.3, the *PROXY* benchmark is parameterized with respect to the number of clients and servers involved in our experiments, and

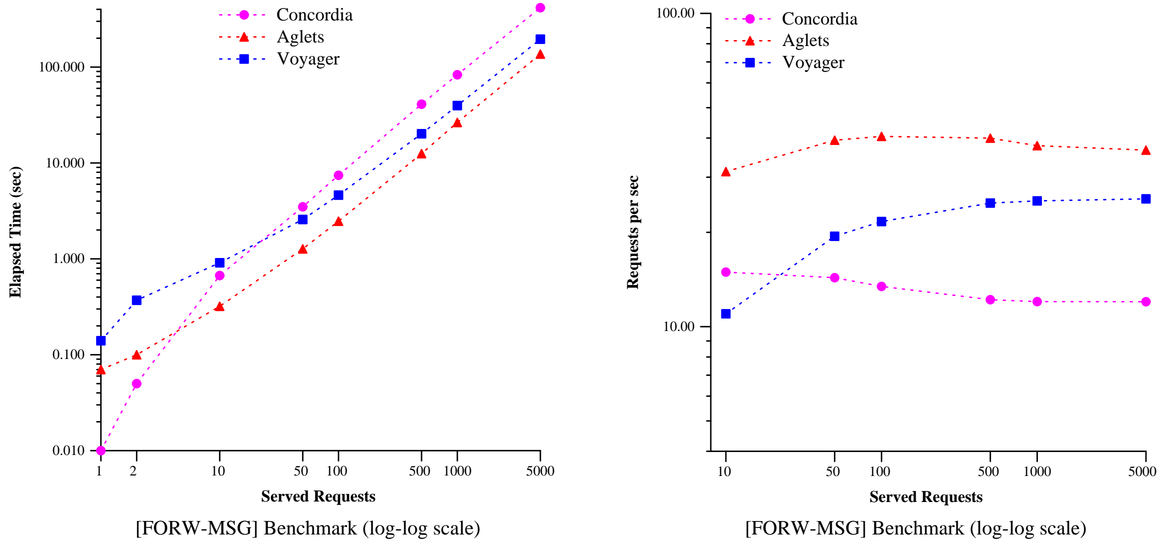


Figure 18: FORW-MSG: Total elapsed times and Service rates for 6 clients, 3 servers.

the total number of requests handled by the proxy-agent.

4.3 FORW-MSG

The *FORW-MSG* micro-kernel represents an implementation of the Forwarding pattern mentioned earlier [1]. This micro-kernel seeks to capture the overhead that arises when a mobile agent receives incoming client requests and forwards them to appropriate servers, taking into account issues such as the capabilities and load of end-servers, etc. The implementation of the *FORW-MSG* micro-kernel investigates the performance of the Forwarding pattern when implemented on top of a MA middleware platform. To this end, we use a forwarding agent that mediates between several clients and servers. The forwarding agent waits for request messages from agent-clients located at different hosts. The forwarding listens for incoming messages carrying client-requests. Whenever it receives a message carrying a request, it inspects the request message and forwards it to the appropriate server. Whenever a server receives a request, it 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.

In addition to the experimental parameters defined in Section 2.3, the *FORW-MSG* 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 forwarding-agent. Here, we report measurements from four experiments involving three server-agents, and three, six, and twelve client-agents respectively, and 1 to 5000 requests served. All agents reside in the same LAN.

We measure the time it takes the forwarding-agent to receive and forward incoming requests to

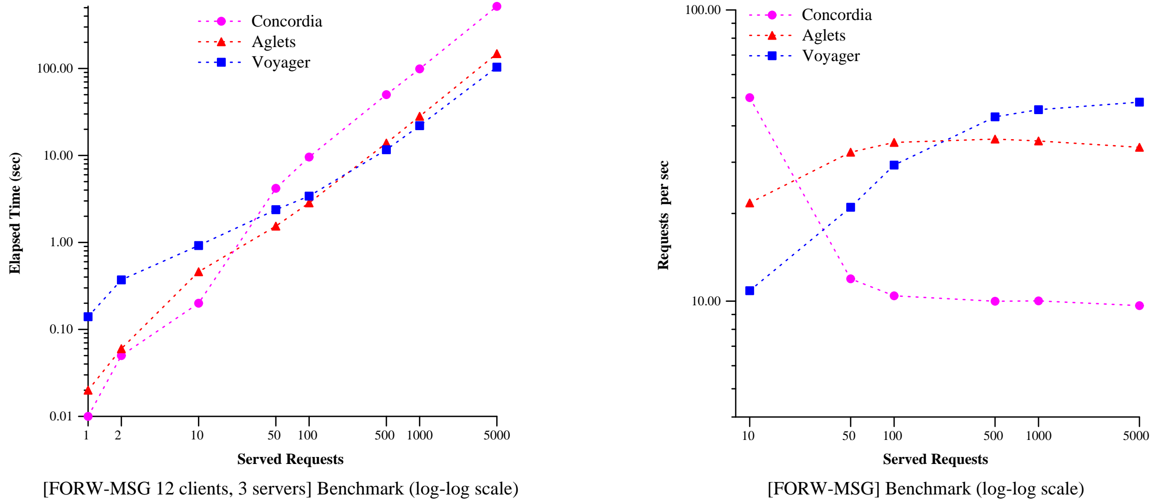


Figure 19: FORW-MSG: Total elapsed times and Service rates for 12 clients, 3 servers.

the appropriate servers. Moreover, we report the rate of request-handling achieved by the forwarding-agent. Figures 17, 18 and 19 present diagrams with our measurements for the three configurations described earlier. We time the performance of the forwarding-agent for handling 1 to 5000 requests sent to three servers from 3, 6, and 12 clients respectively. Increasing the number of clients results to an increase in the number of concurrent requests reaching the forwarding-agent.

As we can see from the diagrams of the Figures 17, 18 and 19, the performance of each MA platform converges to a certain sustained rate of requests served per second in all three experiments. In the three-client case, the Concordia forwarding-agent can handle 12.48 requests per second, Voyager can handle 12.91 requests per second, and Aglets can serve 37.33 requests per second.

As we increase the number of clients to twelve, Concordia's capacity drops by almost 23%, for an equal total number of requests dispatched in our network. In contrast, Aglets capacity drops by 9.5%, whereas Voyager achieves a service rate almost four times higher (see Figure 20). This shows that twelve concurrent clients do not exceed the capacity that a Voyager forwarding-agent has in handling *concurrent* incoming requests, which is something expected given the observed very good performance of messaging under Voyager.

4.4 FORW-MA

The *FORW-MA* micro-kernel represents an implementation of the Forwarding pattern [1]. 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 create a forwarding mobile agent at a place A and dispatch it to a "forwarding" place, B. Subsequently, the forwarding agent "listens"

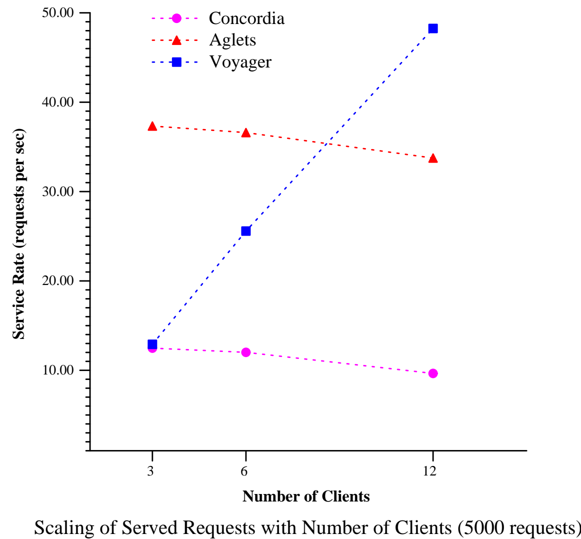


Figure 20: FORW-MSG: Service rates wrt the number of clients.

for incoming agents that we create in A and dispatch to B. Upon arrival of a new agent, the forwarding agent redirects it to a third place, C. In Concordia, the forwarding agent implements the `AgentListener` interface. Through that interface, the forwarding agent registers its presence at the forwarding place, monitors agent-arrival events, and accesses incoming-agents' classes to handle their redirection towards C. In Voyager and Aglets, agents arriving at B, notify the forwarding agent of their arrival and receive redirection instructions through the forwarding-agent's proxy registered at B.

In addition to the experimental parameters defined in Section 2.3, the *FORW-MA* benchmark is parameterized with respect to the total number of mobile agents handled and re-routed by the forwarding agent. For our experiments we use one dispatching and one destination place only. We measure the total elapsed time from the moment the forwarding agent receives notification about the arrival of the first incoming agent until it dispatches the last incoming agent to the destination place. We report measurements from experiments involving 1 to 1000 messenger agents. All agents reside in the same LAN.

As we can see from Figure 21, the forwarding capacity of each MA platform converges to a certain sustained rate of requests served per second. For 1000 agents, Concordia and Aglets can forward 19.84 and 9.54 agents per second respectively, whereas the corresponding number for Voyager is 5.76.

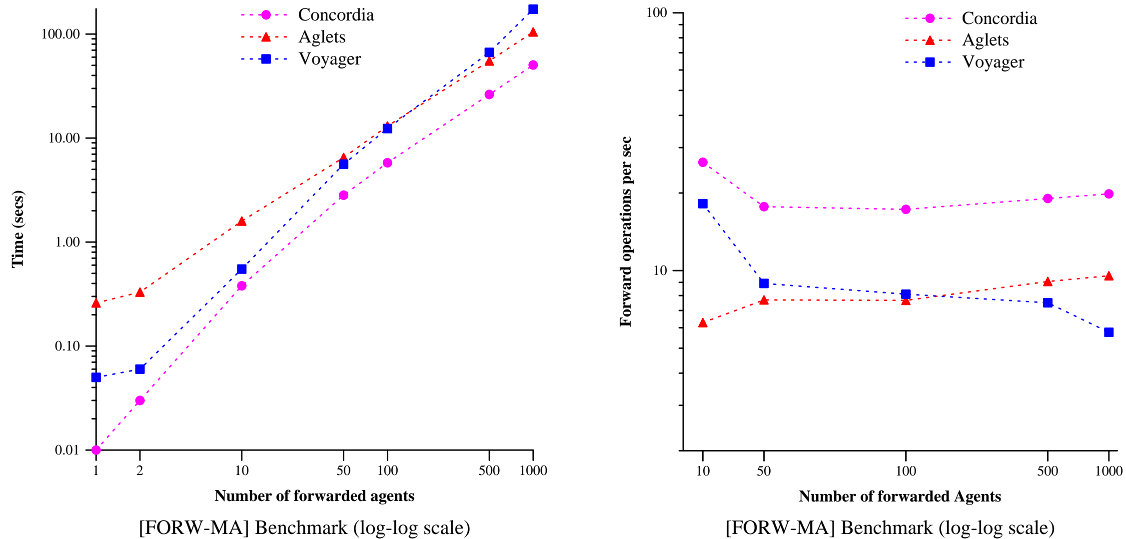


Figure 21: FORW-MA. Total elapsed times and Forwarding rates.

Platform	ROAM (hops/sec)		FORW-MSG (msg/sec)		FORW-MA (agents/sec)	
	Peak	Sustained	Peak	Sustained	Peak	Sustained
Concordia	23.92	0.68	50	9.65	26.31	19.84
Aglets	44.64	44.64	40.49	33.75	9.54	9.54
Voyager	14.7	14.7	48.25	48.25	18.18	5.76

Table 7: ROAM, FORW-MSG and FORW-MA. Peak and sustained rates.

5 Related Work

A number of recent projects have addressed the issues mentioned above. One thread of work examines the relative advantages of the mobile-agent paradigm versus other distributed-computing models from a performance perspective:

For instance, in [26], Strasser and Schwehm introduce a mathematical model to compare analytically the performance of *agent migration* and *remote execution* with the more traditional approach of *remote procedure calls* (RPC), in the case where mobile agents are used for filtering information in information retrieval applications. Their performance model takes into account issues, such as network throughput, communication latency, and network load. Furthermore, the size and execution time for RPCs, the size of agent code, data, and state, the "selectivity" (filtering ratio) of an agent, etc. The authors use their model to identify situations where agent migration has performance advantages over remote procedure calls. Analytical results are corroborated with experimentation

Platform	**Concordia**		**Aglets**		**Voyager**	
Benchmark	**Peak**	**Sustained**	**Peak**	**Sustained**	**Peak**	**Sustained**
CL (agnt/sec)	3125	3000	65.78	11	1189.06	1100
CR (agnt/sec)	312.5	310	29.76	11.05	38.8	38.8
AD (agnt/sec)	25.68	25.6	5.9	5.36	11.58	8.31
MSG-1W (msg/sec)	77.39	73.2	102.94	102.94	1428.57	1146.78
MSG-2W (2wmsg/sec)	31.35	20.2	10.3	8.13	625	476.19
SYNCH (synch/sec)	16.03	14	96.15	92	526.32	413
MSG-MA (rt/sec)	12.15	2	4.93	4.9	9.38	8.3
ROAM (hops/sec)	23.92	0.68	44.64	44.64	14.7	14.7
FORW-MSG (msg/sec)	50	9.65	40.49	33.75	48.25	48.25
FORW-MA (agnt/sec)	26.31	19.84	9.54	9.54	18.18	5.76

Table 8: Synopsis of experimental measurements.

using the Mole platform [25].

A similar problem is studied by Puliafito, Riccobene, and Scarpa in [18]. The authors compare the mobile-agent, remote-evaluation and client-server models of distributed computing. To this end, they use non-Markovian Petri nets modeling, applying probability distributions to model parameters, such as request size, time for searching data in a server, processing time, size of replies and queries, code size for migrating codes, and throughput of the communication network. Petri-net analysis shows that for the scenarios examined, which are pertinent to information filtering applications, the performance advantage of mobile agents arises under certain "external" to this model factors, such as network connectivity and speed.

Mathematical modeling is used by Kotz, Jiang, Gray, Cybenko and Peterson to study a more extended mobile-agent application scenario in [12]. According to this scenario, mobile agents are used to support a data-filtering application involving many wireless clients that filter information from a large data stream arriving across a wired network from a server. The mathematical model is used to compare analytically two alternative approaches: a) The server combines and broadcasts all the data streams over the wireless channel and filtering takes place at each client site. b) Each client dispatches an agent to the server; the agent monitors and filters the data stream before sending relevant data to its corresponding client. The authors use two performance metrics for their comparison study: computation and bandwidth requirements. They conclude that the mobile agent approach trades server computation and cost for savings in network bandwidth and client computation, which is an

important remark in the context of "thin" clients used in mobile-computing applications.

In addition to mathematical analysis and Petri-net modeling, there has been a range of simulation and experimental studies on mobile-agent systems. In [24], Spalink, Hartman and Gibson study the performance advantages of employing a mobile agent for conducting search within a file at the file-server's location instead of searching remotely over the network. Trace-driven simulation shows that the MA approach is advantageous when the server's CPU is not a bottleneck.

Another thread of work employs Petri-net modeling and experimentation to investigate performance properties of mobile-agent systems: For instance, General Stochastic Petri nets are used by Rana in [19] to investigate the performance properties of two agent design patterns [1], "Task" and "Interaction," and a combination thereof. The study is performed in the context of an agent-based, e-commerce application. The Petri-net models introduced are executed with a Petri-net simulator to study the performance and scalability of the underlying application. Furthermore, Petri nets are used by Rana and Stout in [20] to model the performance of multi-agent systems in a way that captures properties arising both from agent-collaboration requirements pertinent to multi-agent applications and the performance characteristics of MA systems.

Samaras, Dikaiakos, Spyrou and Liverdos in [21] propose, employ, and validate an approach to evaluate and analyze MA performance in the context of basic mobile-computing models applied in the provision of distributed database access over the Web. The proposed experimental setup is used to compare quantitatively two commercial MA platforms and to study the performance of different approaches for database access over the Web, using mobile agents.

Silva, Soares, Martins, Batista and Santos define and run benchmarks in [23], to evaluate the performance of some of the existing mobile agent platforms.

Bandyopadhyay and Krishna in [2] study the effectiveness of mobile agents when used as a communication media to support routing in ad hoc, wireless networks. The authors employ a routing scheme based on the location of the various mobile nodes. Their results prove that mobile agents are very effective in the support of autonomous and asynchronous operations, especially for large, highly dynamic ad hoc networks.

In [16], Papastavrou, Chrysanthis, Samaras and Pitoura evaluate all currently available Java-based approaches that support persistent connections between Web clients and database servers. These approaches include Java applets, Java Sockets, Servlets, Remote Method Invocation, CORBA, and mobile agents technology. The comparison was along the dimensions of performance of query processing and programmability. Their findings point out that best performance is not always achievable with high programmability and low resource requirements, moreover, the mobile agent technology needs to improve its programmability while giving particular emphasis in its infrastructure. The study provided an insight to potential scalability problems with the currently available

mobile agent implementations and in particular the Aglets Workbench. It was shown that the mobile agents approach, as represented by Aglets, cannot support interactions that require movement or exchange of large amounts of data such as a large number of consecutive queries with increased size of query result. Hence, it is necessary to develop more efficient mobile agent infrastructures, if the full potential of mobile agents is to be explored. They also suggest that the goal should not be to make mobile agents yet another communication paradigm but instead a more effective distributed computing one. Their result crowned CORBA as the overall winner and Applets as the clear looser.

Ismail and Hagimont in [10] investigate "low-level" aspects that affect mobile-agent performance. To this end, they implement a stripped-down mobile-agent platform on Java and conduct timings on the overhead of agent serialization, agent transfer over a local- or wide-area network, and agent installation on a receiving machine. Furthermore, they employ their experimental platform to implement and evaluate the performance of two typical distributed applications (Quality of Service management in a distributed multimedia application and Data Mining on the Web). Performance results are compared with results from respective implementations with a full-fledged MA platform (Aglets by IBM) and a client-server approach implemented on top of Java's RMI. The experiments conducted show that, in spite of the costly mechanisms employed by Java to support object migration, mobile agents can lead to significant performance improvements, especially when agent migration allows a significant reduction in communication between a client and a server.

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 analyze 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 approach for analyzing the performance of MA middleware quantitatively, by focusing on the different layers of a MA-based system's architecture. 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, should be adopted as a practice for both the MA middleware design, and the programming of applications. Raw performance data show that agents cannot sustain the typical loads expected to arise in Internet middleware. 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. Therefore, support for high-performance Java programming, resource management, and performance debugging must be added to MA platforms of the near future, along with current efforts to expand MA application interfaces. We are currently working towards the implementation of and experimentation with micro-applications and the development of our framework upon other mobile-agent middleware platforms. We are also investigating how the low-level implementation aspects of different platforms affect micro-benchmark and micro-kernel results.

7 Acknowledgements

This work was supported partly by the grant PENEK-No 23/2000 from the Research Promotion Foundation of Cyprus.

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] S. Bandyopadhyay and K. Paul. Evaluating the Performance of Mobile Agent-Based Message Communication among Mobile Hosts in Large Ad Hoc Wireless Network. In *Proceedings of the 2nd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 69–73. ACM, August 1999.

[3] W. Brenner, R. Zarnekow, and H. Wittig. *Intelligent Software Agents. Foundations and Applications.* Springer, 1998.

[4] 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.

[5] T. Budd. *Understanding Object-Oriented Programming with JAVA.* Addison-Wesley, 2000.

[6] A. Castillo, M. Kawaguchi, N. Paciorek, and D. Wong. Concordia as Enabling Technology for Cooperative Information Gathering. In *Japanese Society for Artificial Intelligence Conference*, June 1998. http://www.meitca.com/HSL/Projects/Concordia/.

[7] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.

[8] M. Dikaiakos and D. Gunopulos. FIGI: The Architecture of an Internet-based Financial Information Gathering Infrastructure. In *Proceedings of the International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, pages 91–94. IEEE-Computer Society, April 1999.

[9] G. Glass. Overview of Voyager: ObjectSpace's Product Family for State-of-the-Art Distributed Computing. Technical report, ObjectSpace, 1999.

[10] L. Ismail and D. Hagimont. A Performance Evaluation of the Mobile Agent Paradigm. In *OOPSLA '99, Proceedings of the Conference on Object-oriented Programming, Systems, Languages and Applications*, pages 306–313. ACM, November 1999.

[11] R. Koblick. Concordia. *Communications of the ACM*, 42(3):96–99, March 1999.

[12] David Kotz, Guofei Jiang, Robert Gray, George Cybenko, and Ronald A. Peterson. Performance Analysis of Mobile Agents for Filtering Data Streams on Wireless Networks. In *Proceedings of the Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2000)*, pages 85–94. ACM Press, August 2000.

[13] D. B. Lange and M. Oshima. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–91, March 1999.

[14] Mitsubishi Electric ITA. *Concordia Developer's Guide*, October 1998. http://www.meitca.com/HSL/Projects/Concordia.

[15] ObjectSpace. *ObjectSpace Voyager, General Magic Odyssey, IBM Aglets. A Comparison.* ObjectSpace, June 1997.

[16] S. Papastavrou, P.K. Chrysanthis, G. Samaras, and E. Pitoura. An Evaluation of the Java-based Approaches for Web Database Access. In *Proceedings of the Fifth IFCIS International Conference on Cooperative Information Systems (CoopIS2000)*, September 2000.

[17] S. Papastavrou, G. Samaras, and E. Pitoura. Mobile Agents for WWW Distributed Database Access. In *Proceedings of the Fifteenth International Conference on Data Engineering*, pages 228–237. IEEE, March 1999.

[18] A. Puliafito, S. Riccobene, and M. Scarpa. An Analytical Comparison of the Client-Server, Remote Evaluation and Mobile Agents Paradigms. 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 278–292. IEEE-Computer Society, October 1999.

[19] O.F. Rana. Performance Management of Mobile Agent Systems. In *Proceedings of Autonomous Agents 2000*, pages 148–155. ACM, June 2000.

[20] O.F. Rana and K. Stout. What is Scalability in Multi-Agent Systems? In *Proceedings of Autonomous Agents 2000*, pages 56–63. ACM, June 2000.

[21] 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.

[22] G. Samaras, E. Pitoura, and P. Evripidou. Software Models for Wireless and Mobile Computing: Survey and Case Study. Technical Report TR-99-5, Department of Computer Science, University of Cyprus, March 1999.

[23] L.M. Silva, G. Soares, P. Martins, V. Batista, and L. Santos. Comparing the Performance of Mobile Agent Systems: a Study of Benchmarking. *Computer Communications*, 23(8):769–778, 2000.

[24] T. Spalink, J. Hartman, and G. Gibson. The Effects of a Mobile agent on File Service. 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 42–49. IEEE-Computer Society, October 1999.

[25] M. Strasser, J. Baumann, and F. Hohl. Mole - A Java Based Mobile Agent System. In J. Baumann, editor, *2nd ECOOP Workshop on Mobile Object Systems*, 1996.

[26] M. Strasser and M. Schwehm. A Performance Model for Mobile Agent Systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 97)*, pages 1132–1140, June 1997.

[27] Transaction Processing Performance Council (TPC). *TPC Benchmark W (Web Commerce) - Draft Specification*, December 1999.

[28] D. Wong, N. Paciorek, and D. Moore. Java-based Mobile Agents. *Communications of the ACM*, 42(3):92–95, March 1999.

[29] 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/.

[30] M. Woodside. Software Performance Evaluation by Models. In C. Lindemann G. Haring and M. Reiser, editors, *Performance Evaluation: Origins and Directions*, pages 283–304. Springer, 1999.

[31] G. Yamamoto and Y. Nakamura. Architecture and Performance Evaluation of a Massive Multi-Agent System. In *Proceedings of the 3rd Annual Conference on Autonomous Agents*, pages 319–325, May 1999.