Mobile-Agent versus Client/Server Performance: Scalability in an Information-Retrieval Task

Robert S. Gray, David Kotz, and Ronald A. Peterson, Jr.
Dartmouth College

Peter Gerken, Martin Hofmann, and Daria Chacón
Lockheed-Martin Advanced Technology Laboratory

Greg Hill and Niranjan Suri
University of West Florida

Dartmouth College Computer Science
Technical Report TR2001-386

January 30, 2001

Abstract

Mobile agents are programs that can jump from host to host in the network, at times and to places of their own choosing. Many groups have developed mobile-agent software platforms, and several mobile-agent applications. Experiments show that mobile agents can, among other things, lead to faster applications, reduced bandwidth demands, or less dependence on a reliable network connection. There are few if any studies of the scalability of mobile-agent servers, particularly as the number of clients grows. We present some recent performance and scalability experiments that compare three mobile-agent platforms with each other and with a traditional client/server approach. The experiments show that mobile agents often outperform client/server solutions, but also demonstrate the deep interaction between environmental and application parameters. The three mobile-agent platforms have similar behavior but their absolute performance varies with underlying implementation choices.

Corresponding author: Bob Gray (`rgray@dartmouth.edu`).
This research was supported by the DARPA CoABS Program (DARPA contracts F30602-98-2-0107 and F30602-98-C-0162 for Dartmouth and Lockheed Martin respectively) and by the DoD MURI program (AFoSR contract F49620-97-1-03821 for both Dartmouth and Lockheed Martin).

1 Introduction

One of the most attractive applications for mobile agents is distributed information processing, particularly in mobile-computing scenarios where users have portable computing devices with only intermittent, low-bandwidth connections to the main network. A mobile agent can leave the portable device, move to the network location of a needed information resource, and perform a *custom* retrieval task local to the resource. Only the final results are transmitted back to the portable device. Source data and intermediate results are not transmitted, even though the information resource had no prior knowledge of the client or its specific task. Moreover, the mobile agent can continue the retrieval task even if the network link to the portable device goes down. Once the link comes back up, the agent sends back its results.

By moving the code to the data, a mobile agent can reduce the latency of individual steps, avoid network transmission of intermediate data, continue work even in the presence of network disconnections, and thus complete the overall task much faster than a traditional client/server solution. Of course, a mobile agent will not always perform better than a client/server solution. For example, if the agent code is larger than the total intermediate data, the mobile agent must perform worse, since it will transfer more bytes across the network than the client/server solution. Similarly, if the network is fast enough, the agent might do worse even if the code is smaller, since mobile agents are typically written in an interpreted language for portability and security reasons. With a fast and reliable network, interpreting the agent on the server might be slower than transmitting the intermediate data to the client. As network speed and reliability drops, however, or data sizes increase, the picture changes considerably.

Another common concern about the performance of mobile-agent systems is that they shift a lot of the processing load from the clients to the server. In many environments, this is a significant advantage. For example, the clients may be hand-held computers with limited memory and computational power, and the "server" may be a large multiprocessor computer. Still, this shift of computation to the server does lead to questions about the scalability of the mobile-agent platform. As the number of clients increases, how well can the mobile-agent services scale? Where is the trade-off between the savings in network transmission time and the possible extra time waiting for a clogged server CPU?

We set out to examine this question through a series of experiments. We were interested in comparing a traditional client/server (RPC) approach with a mobile-agent approach, and in comparing several mobile-agent platforms. Our goal is to understand the performance effects that are fundamental to the mobile-agent idea, and separately the performance effects due to implementation choices made by the different mobile-agent platforms.

So, in this paper, we introduce three mobile-agent platforms: D'Agents, EMAA, and NOMADS. Then we describe the scenario chosen for our tests, and the details of the tests themselves. Finally, we present the experimental results and discuss our interpretation.

2 Mobile-agent systems

In our experiments we chose to evaluate three mobile-agent platforms: D'Agents from Dartmouth College, EMAA from Lockheed-Martin Advanced Technology Laboratory, and NOMADS from the University of West Florida Institute for Human & Machine Cognition.

2.1 D'Agents

D'Agents[1] is a mobile-agent system that was developed at Dartmouth College to support information-retrieval applications and first released in 1994 [Gra97, GKCR98, GCKR00, GCK+00]. The architecture of D'Agents is shown in Figure 1.

[1]D'Agents was once known as Agent Tcl.

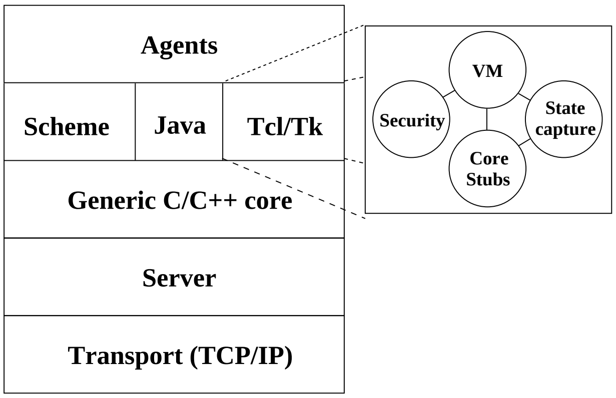

Figure 1: Architecture of the D'Agents mobile-agent system.

Like all mobile-agent systems, D'Agents is based on a server that runs on each machine. The server receives incoming agents, authenticates the identity of the agent's owner, assigns access permissions to the agent based on this identity, and executes the agent inside the appropriate execution environment. Unlike the other mobile-agent systems described in this paper, however, D'Agents supports multiple languages (Tcl, Java and Scheme) allowing the agent programmer to select an appropriate language for her particular application. In addition, for both Tcl and Java, D'Agents supports strong mobility, where the agent's complete *control state*, rather than just its code and date state, is moved from one machine to another. If an agent decides to move to a new machine inside a *for* loop, for example, the agent will still be inside the *for* loop when it resumes execution on the new machine.

Experience with undergraduate programmers suggests that strong mobility is easier than weak mobility for the *agent* programmer to understand and use [Whi96]. Strong mobility does require significant effort from the *system* programmer, however, since the system must capture enough state information to restore the agent in exactly the same control state on its new machine. In fact, off-the-shelf Tcl and Java interpreters do not provide the state-capture routines that are needed for strong mobility, and D'Agents uses modified Tcl and Java interpreters to execute its agents. Since modified interpreters make it time consuming to upgrade to new interpreter versions, D'Agents still uses Java 1.0. In contrast, many other mobile-agent systems (such as EMAA) run on top of standard Java interpreters, can be upgraded easily as new versions of Java are released, and currently use Java 2.0. Java 2.0 supports just-in-time compilation, allowing Java agents in those other systems to execute more closer to the speed of native code than in D'Agents. At the same time, the modified Java interpreter in D'Agents does have an optimized set of string routines, which improves performance significantly for string-intensive, information-retrieval tasks. For such tasks, which include the task analyzed later in this paper, a Java agent in D'Agents can equal or better the performance of a Java agent in the other systems, despite the lack of just-in-time compilation.

When an agent migrates in D'Agents, the entire code base and state image is sent from the source to target machine. D'Agents does not support code caching or code servers, nor does it fetch code from the agent's previous machine on demand. In addition, D'Agents creates a new TCP/IP connection for each message or migrating agent, rather than holding open existing connections between pairs of servers. D'Agents does send user-level acknowledgments, however, to defeat the *delayed acknowledgment* feature of TCP/IP [Ste94].

In early versions of D'Agents, the server was a multi-process server that did not spawn an interpreter process for an agent until the agent had actually arrived. For example, once a Tcl agent arrived, the server would spawn a Tcl interpreter. The overhead of this interpreter startup was large, since it involved calls to the Unix *fork* and *exec* functions, as well as initialization of the interpreter's internal data structures. To remove this startup overhead from the critical migration path, the most recent version of D'Agents has a multi-threaded server that maintains a pool of "hot" interpreters. This multi-threaded server starts up a set of interpreter processes at boot time, and then hands off incoming agents to the first free interpreter in that set. An interpreter process does not die when an agent finishes, but instead stays alive to execute the next incoming agent. Although this approach still runs each agent in a separate process,[2] it eliminates nearly all of the startup overhead.[3] In contrast to D'Agents, many other mobile-agent systems (such as EMAA, but not NOMADS) run each agent as a thread inside the server process itself. This approach reduces the migration overhead even further. Unfortunately, this approach is difficult in D'Agents, since D'Agents supports multiple languages. A multi-threaded server with pools of separate interpreter processes strikes a balance between efficiency and complexity.

D'Agents has no centralized services involved for basic agent communication or migration. By default, for example, an agent is assigned a name that is unique only in the context of its current machine. An agent receives a globally unique name only if it *chooses* to register with a directory service. In all of the experiments in this paper, when a D'Agents agent migrates to a new machine, there is only a single communication step, i.e., the old machine sending the agent to the new machine. There is no communication with any services on other machines. Finally, although D'Agents includes extensive security mechanisms, such as digitally signing and encrypting mobile agents, none of these security mechanisms were turned on during the experiments in this paper. Security mechanisms also were turned off in the other systems.

2.2 EMAA

The Extendable Mobile Agent Architecture (EMAA) is a Java-based, object-oriented mobile-agent architecture [MCW00, CMMS00]. It provides a simple, structured way for an agent to migrate from one computing node to another and to use the resources at that node. Figure 2 depicts the basic EMAA architectural components. At EMAA's core lies an agent Dock that resides on each execution host. The Dock provides an execution environment for agents, handles incoming and outgoing agent migration, and allows agents to obtain references to services. EMAA allows users to define agents, services, and events. Agents are composed of small, easily reused tasks performed to meet a goal for a user. An agent's tasks are encapsulated in an itinerary; itineraries are structured as process-oriented state graphs. Agents may be mobile, and they typically make use of stationary services. Services may implement connections to external systems (such as databases and other applications), may themselves provide some complex functionality, or may carry out any number of other functions, so long as they are not themselves primary actors. Goal-orientation and directed activity is generally reserved to be the function of agents. Both agents and services may send and receive events.

EMAA agents employ weak mobility; that is, the agent's full execution state (stack and program counter) is not transferred to the receiving machine.

[2] More precisely, each Tcl and Scheme agent runs inside its own process, but multiple Java agents run inside the same Java process, since Java's thread support makes it straightforward to have multiple agents inside a single Java virtual machine. In fact, due to the large memory footprint of the Java virtual machine, multiple Java agents per virtual machine is essential for system scalability.

[3] It does not eliminate all of the overhead since each interpreter process is allowed to handle only a fixed number of agents before it terminates and is replaced with a new process. In addition, even though an interpreter process remains active from one agent to another, the process still must do some initialization and cleanup for each agent.

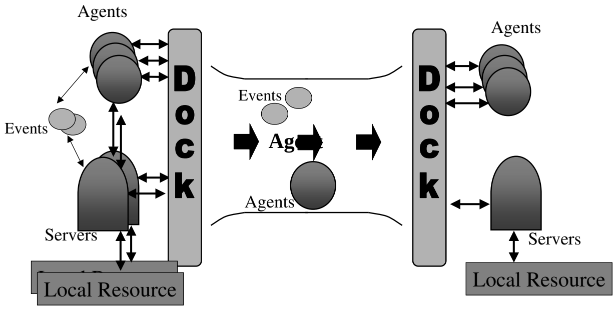

Figure 2: Architecture of the EMAA mobile-agent system.

Instead, the agent's data state is maintained, and the agent starts execution at a predefined entry point upon arrival. An EMAA agent may migrate in between execution of states in its itinerary. To migrate, an agent uses the CommunicationServer, a core Server that is a part of the Dock, to serialize and send itself to another machine. The CommunicationServer provides SSL-encrypted agent transfer over TCP/IP sockets. The receiving CommunicationServer verifies that the object received is an agent, that the class files needed to run the agent are present (if not present, they are obtained using a ClassLoader), then delivers the agent to the local AgentManager. The AgentManager gives the agent an execution thread and invokes its entry method, and may provide additional registration or authentication.

EMAA agents were designed to function robustly under harsh conditions such as those found in many tactical military networks; these frequently feature unreliable, low-bandwidth wireless connectivity. To this end, EMAA agents are typically small; for example, the agent used in the scalability tests described by this report measured approximately 750 bytes when not configured with an itinerary and tasks, and fewer than 1400 bytes when fully configured. Agents can be remotely controlled and monitored using EMAA's distributed event services, and agents may checkpoint their data state to support reinstantiation in the face of machine failure.

2.3 NOMADS

NOMADS is a system that provides strong mobility and strong security for Java-based mobile agents [SBB+00]. Strong mobility implies that the NOMADS system captures and moves the complete execution state of the agent from one platform to another (similar to D'Agents). NOMADS also supports the notion of forced mobility, where the system or administrator may asynchronously move agents between platforms potentially even without the agents realizing the movement.

One of the major limitations of the Java environment is the lack of resource-control capabilities. While the Java 2 security architecture and JAAS provide good access-control capabilities, they do not provide any resource control. For example, while it is possible to limit an agent from writing to any directory except the `/tmp` directory, once access is granted to write to the `/tmp` directory, no quantitative limits may be placed on the agent. The agent could create a dummy file and fill up all the disk space available with garbage. Similarly, if an agent is granted write access to some IP address, the agent could easily flood the network by writing continuously to that IP address. The lack of resource-control capabilities in the standard Java environment (and in all agent systems that rely on the standard Java environment) leave the system wide open to denial-of-service attacks.

NOMADS provides strong security by controlling the resources consumed by agents running within the NOMADS environment. Two types of resource controls are available: rate controls and quantity controls. Rate controls apply to I/O rates (such as disk read rate, network write rate) and to the CPU (bytecodes executed per millisecond or percentage of CPU used). Quantity controls apply to I/O operations (such as the total number of bytes written, or disk space used).

NOMADS relies on the Aroma VM, a Java-compatible VM that provides the necessary state-capture and resource control facilities. The Aroma VM is a "clean-room" implementation of the Java VM specification– it does not use any source code from the standard Java implementations, which allows the Aroma VM and consequently NOMADS to be distributed freely. While Aroma is mostly JDK 1.2.2 compatible, it does not provide support for AWT and swing. Also, Aroma does not support a Just-In-Time (JIT) compiler yet, which results in poor performance when compared to the standard Java environments. This lack is the primary reason for the slow task times of the NOMADS system in the scalability experiments described in this paper.

The Aroma VM is embedded inside Oasis, the agent execution environment. The structure of Oasis is shown in Figure 3. The `oasis` process is a daemon that usually runs in the background as a service. Users and administrators may interact with `oasis` through the Administration and Console program, which can be used to configure Oasis, specify accounts and security policies, and perform I/O with agents. The `oasis` service executes each agent in a separate instance of the Aroma VM. This archi-

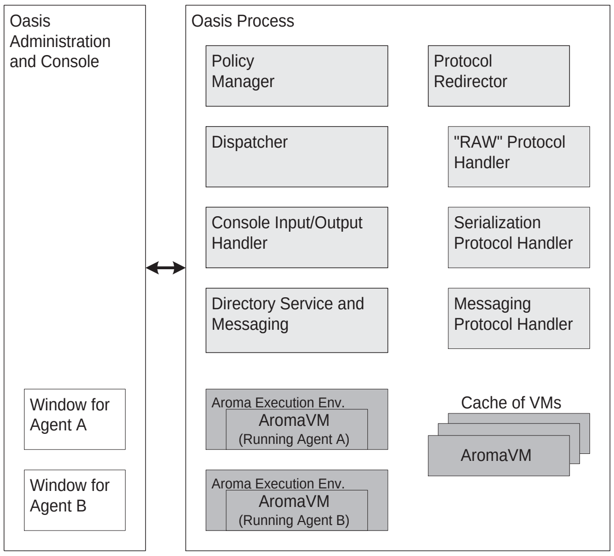

Figure 3: Architecture of the Oasis agent-execution environment for NOMADS.

tecture helps enforce the resource controls on a per-agent basis. Note that all the instances are within the same process and hence share all of the VM code and loaded classes. Essentially, the only part that is duplicated in each VM is the heap. The `oasis` service also maintains a cache of VMs to use for agents. New VMs are created on the fly if needed.

Currently, Oasis supports only the Aroma execution environment. The design of Oasis allows other execution environments to be included if desired. We plan to support the standard Java VM as an alternate execution environment. Other languages could also be incorporated via additional execution environments.

Oasis supports multiple protocol handlers that implement the agent transfer and agent messaging. Two migration mechanisms are available: full VM state (known as Raw) or individual thread serialization. In addition, the agent state may be compressed and decompressed on the fly. Oasis also includes a policy manager that maintains the security policies that are in effect. Other components include a dispatcher that is responsible for starting and stopping the execution of agents, a directory service and messaging component, and a console I/O redirection component that allows the I/O of an agent to be redirected over a network socket (similar to X windows).

3 Experiments

The goal of our experiment was to compare the scalability of mobile-agent performance versus client/server performance in an information-retrieval (IR) task as the number of clients increases. To explore this issue we implemented a simple IR task using both an agent and a client/server architecture. Our IR task represents a simple keyword query on a collection of documents stored at the server. The agents jump to the server, scan all documents for the desired key words, and return with the matching subset of documents. In the client/server case, the client downloads all of the documents and scans them on the client.

Thus the client/server application downloads all documents on every query, and does its filtering on the client machine. The agent-based application does its filtering on the server and downloads only the matching documents. The client/server application is written in C++ (for speed), while the agent-based application is written in Java (for mobility). The trade-off is thus between network bandwidth consumption and processing speed, between a fast language on distributed clients and a slower language on a shared server.

In our experiments we used one server and one to ten clients, measuring the average time for each query to complete. In the remainder of this section we provide the details of the experimental conditions.

3.1 The experiments

The information-retrieval task was a three-word keyword query on a set of 60 documents, each of which was 4096 bytes in size. The characteristics of this task are similar to those of searches on our technical-report collection here at Dartmouth, although it is not modeled on any specific real-world system. All matching documents would be returned to the user, in their entirety.

Both agent and client/server implementations scanned the entire set of 60 documents for the three keywords, so that the task's computational complexity is equivalent for both approaches. Introducing the unpredictable scan times of real queries would have required much longer testing times to get believable averages. We chose a complete scan to reduce testing time and to obtain proper experimental control. For similar reasons, we chose to declare a certain fraction of the documents to be “selected”, ignoring the actual results of the query, to increase our control over the size of the task output.

We wrote the client/server applications in C++ using TCP/IP connections with a simple protocol for handshaking between client and server. These are the steps involved in a query, from any one of the clients on any client machine:

1. Record the start time in milliseconds.
2. The client sends the keywords to the server via a TCP/IP connection.
3. The server returns 60 documents on the same connection.
4. The client executes the query (which always returns the number of documents corresponding to a selection ratio).
5. A stop time is recorded. Stop time minus start time gives total query time. Times are summed and then averaged at the end of the test by the client.

Rather than write and debug a multi-threaded server application, we ran a separate server on a different port for each client. The idea here is that having n independent server tasks on a single machine is similar to having n threads on a server spawned in response to incoming client requests, especially since Linux threads are separate tasks anyway. Thus the server machine running multiple server applications is a close equivalent of a machine running a multi-threaded server application.

We wrote the agent applications in Java. The speed of any application written in the Java language, even with a JIT compiler, is slower than that of an equivalent implementation in C++. This difference works only in the favor of the client/server approach, therefore, so any performance benefits seen with the agent approach are not due to language differences.

We used three agent systems: D'Agents AgentJava, EMAA, and NOMADS. Each system required some porting of the agent application due to differences in how each agent system works. We reviewed the ported code to ensure that all implementations would behave in a functionally identical way.

There are three different virtual machines used by the three different mobile-agent platforms, however, and it is important to understand the performance effects. D'Agents "AgentJava" uses a modified JDK 1.0.2, EMAA use the Linux Blackdown JDK 1.2.2 port with JIT compiler, and NOMADS uses its own clean-room JVM that has not yet been optimized for speed. To understand the speed differences, we ran the IR task alone in each platform.

Comparison of Average IR Task Times

	20% ratio	5% ratio
C++	3.02ms	2.92ms
D'Agents	61.6ms	55.9ms
EMAA	96.1ms	88.9ms
NOMADS	?ms	?ms

The C++ times are markably faster due to the extremely inefficient Java file I/O routines. All of the times reflect little actual disk activity because the underlying Linux file cache holds all of the documents used for the test. Due to an optimized string handling library in D'Agents it was significantly faster than JDK 1.2.2 (EMAA), even though it did not have a JIT compiler. This difference accounts for some of the performance differences between the mobile-agent systems as described below. We unfortunately do not have NOMADS numbers in time for this printing.

The agent-based experiments are controlled by a master agent that deploys the individual client agents, then listens for times to be reported by client agents.

Each client agent loops over many queries, following this procedure for each query:

1. Record "start time."
2. Jump to the server machine.
3. Record the "task start time."
4. Run the query against the 60 documents to get the matching documents based on the selection ratio.
5. Record the "task stop time."
6. Jump back to the client machine with the matching documents.
7. Record the "stop time." Stop time minus start time gives total query time. Task-stop time minus task-start time gives total task time. Total query time minus total task time gives jump time. In this way clock differences between machines become irrelevant.
8. Report these times to the master agent.

In our experiment we examined client/server and three mobile-agent implementations. We also varied

- the number of clients (1 to 10, each on a separate machine),
- the network bandwidth to the server (1, 10, 100 mbps),[4] shared by all clients, and
- the percentage of documents selected (5%, 20%).

Other parameters, fixed for these experiments, were

- the number of documents in the collection (60),
- the document size (4096 bytes),

[4]In this paper we use the prefixes m and k to refer to powers of 10, and the prefixes M and K to refer to powers of 2. Thus 10 mbps refers to 10,000,000 bits per second.

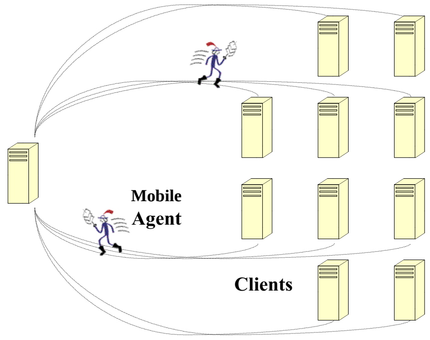

Figure 4: The cluster used for the experiments.

- the number of queries (200-1000 queries, depending on the agent system), and
- the query rate (1 query every 2 seconds, see below).

The results were averaged over 200 to 1000 queries depending on the agent system, using whatever number of queries was required to get repeatable results. The query rate was set to average one query per two seconds, but uniformly distributed over the range 0.25–0.75 queries per second. This randomness prevents agents from exhibiting synchronous behavior. This query rate is a maximum: if a query takes longer than two seconds to complete its task, the next query will not be started until the agent returns to its client machine.

We ran the experiments on a set of eleven identical Linux workstations, as in Figure 4.[5] Ten of the machines act as clients and one acts as the document server. Although we interconnected the computers with a physical 100 mbps "Fast Ethernet" network, we reduced the bandwidth available by either inserting a 10 mbps hub, or by inserting a Pentium-100 computer running the DummyNet bandwidth manager. DummyNet allowed us to set the bandwidth anywhere from 2.4 kbps to 3 mbps.[6]

We used a twelfth machine to launch the clients and monitor the experiments. Each of the eleven test machines contained all the necessary code and documents on their local disks, so that the tests would not be affected by unnecessary NFS traffic. Similarly, we isolated the test computers and network from outside traffic so that our results are not skewed by unrelated activity.

4 Results and discussion

We plot several aspects of the results in a series of figures attached to the end of this report. We first consider the total query time, and then its components "task time" and "jump time." Then we make a direct comparison between the client/server times and the agent times, by presenting the ratio between client/server and agent times.

4.1 Total query time

Each plot in Figure 5 shows the total query time for all systems. The x axis is the number of clients, which also corresponds to the number of client machines. The y axis is the averaged per-query time in milliseconds (note there is a separate scale for the NOMADS data). For the agent approach, this is the elapsed time for the agent to jump from client machine to server machine, retrieve the documents, determine which documents match the query, and jump back to the client machine. (The next two sections and two figures separate the computation and jump components of this total.)

The figure shows six plots, for three bandwidths (1, 10, and 100 mbps) and two selection ratios (5% or 20%, respectively). Generally speaking, any given implementation will slow down linearly with the number of clients, due to increasing contention for the network and the server's CPU. The slope of that line depends on the overhead of that implementation, the parameters of the query, and the speed of the network and CPU. In most cases there will be an inflection point where the slope suddenly increases, once the number clients pose sufficient load to reach the limitations of the network or CPU. That effect can be seen most readily in our 10 mbps client/server experiments, where the demands of 9 and 10 clients begin to exceed the limits of the network.[7] The effect probably occurs in the agent implementations, only with more clients than we were able to test.

In the 1 mbps network, the fact that agents bring back only 5 or 20% of the documents allows them to be less sensitive to the constraints of the slow network, while the client/server approach is bandwidth-limited. Here, as in the 10 mbps network, EMAA

[5] VA Linux VarStation 28, Model 2871E, 450 mHz Pentium II, 256 MB RAM, 5400 rpm EIDE disk, running the Linux 2.2.14 (RedHat 6.1) operating system.

[6] DummyNet is a modified FreeBSD firewall that can be used to control bandwidth allocation. See `http://info.iet.unipi.it/~luigi/ip_dummynet/` for more information.

[7] Nine clients pull 4096 bytes/document times 60 documents every 2 seconds, or 8,847,360 bps. That rate exceeds the practical limits of 10 mbps Ethernet.

and D'Agents clearly perform much better than client/server. NOMADS is much slower, due to its slower JVM (as we discuss in the next section).

In the 100 mbps network, however, client/server is the clear winner. In this environment, the network has more than enough bandwidth to allow the clients to retrieve all of the documents. With the network essentially free, the slower computation of the agents (using Java rather than C++, and sharing the server rather than dispersing among the clients) makes the mobile-agent approach a less attractive option.

The variability of EMAA, and the differences between EMAA and D'Agents, are better examined in terms of the task times and jump times, below.

4.2 Task times

Each plot in Figure 6 shows the task time for all agent systems. Recall that the task time is the time for computation of the filtering task only. The x axis is the number of client agents, which also corresponds to the number of client machines. The y axis is the average task time in milliseconds.

The most notable feature in these graphs is the dramatic difference between the NOMADS times (which have a separate y-axis scale) and the other agent systems. This difference is due to the home-grown JVM implementation in the NOMADS project, which has not been tuned. The NOMADS data grows linearly with the number of clients, indicating that the server's CPU is always the limiting factor for NOMADS.

The D'Agents task time is the fastest, in large part because it uses an *older* version of the JVM than EMAA, a version with native (rather than Java) implementations of the critical string-manipulation routines. Our document-scanning application stresses those routines, leading to better performance for D'Agents in this case.

The D'Agents time is largely constant, because the query rate (average 1 query per client per 2 seconds, or less) is low enough to not stress the server CPU. EMAA's task times, in contrast, are high enough to stress the server CPU. We believe that the cause is the time needed to serialize and transmit the returning agent; note that the task times increase for larger agents (20% graphs) at the higher bandwidths (10 and 100 mbps).

The EMAA task times show two surprising dips in each of the 5% cases. These dips are repeatable. We are investigating the causes of these dips.

4.3 Jump times

Each plot in Figure 7 shows the per-query jump time for each system. Recall that the jump time is the total query time minus the task time, so it includes all of the computational overhead of a jump (serialization, network protocol, deserialization, and reinstating an agent) as well as the network time. The x axis is the number of client agents, which also corresponds to the number of client machines. The y axis is the average jump time in milliseconds (note there is a separate scale for the NOMADS data on *some* of the plots).

The jump times are most difficult to interpret, because they depend on the load of both the network and the server's CPU. The higher NOMADS times, for example, are likely due to the heavy load on the CPU impacting the time needed for serialization, transmission, and deserialization of jumping agents. We believe that the unsteady curves in the NOMADS results are due to the limited number (20) queries we ran in NOMADS experiments. We plan further experiments so that the numbers better represent the steady-state performance.

EMAA's time is largely steady, although there is a significant jump in the 10 mbps 20% environment, possibly due to significant server load. The D'Agents curve is most dramatic in the 1 mbps 5% environment. We are still investigating the cause for this behavior.

Interestingly, D'Agents is slower than EMAA in two of the graphs, about the same in one, and faster in three. The reason is that D'Agents is more affected than EMAA by the network load and bandwidth: note that the difference between the D'Agents jump time and the EMAA jump time decreases steadily (from positive to increasingly negative) as the network bandwidth increases or as the network load increases (from 5% to 20% selection). D'Agents is more affected because its agents tend to be bigger.

4.4 Ratio of client/server time to agent time

Each plot in Figure 8 shows the "performance ratio," which is the client/server query time divided by the mobile-agent query time. A ratio of 1 indicates that the agent approach and the client/server approach are equivalent in performance; higher than 1 indicates that agents were faster. The x axis is the number of clients, which also corresponds to the number of client machines. The y axis is the performance ratio. The NOMADS ratio is indistinguishable from zero because the NOMADS times were so large.

For EMAA and D'Agents, there are three different effects, dependent on bandwidth.

In the 1 mbps curves, we see that the performance ratio for both EMAA and D'Agents climb, and then fall or level off. For small numbers of D'Agents agents, the performance ratio improves quickly because the client/server approach is bandwidth limited, while the agent approach is not. With a few more agents, D'Agents reaches the network bandwidth limit and becomes slower, reducing the performance ratio. Once both client/server and agent performance have reached the same slope, the performance ratio levels off. We believe EMAA sees the same effect.

In the 10 mbps curves, we see a different effect. Here, the agents never hit the network limit, but the client/server implementation hits the limit at 9 clients. The performance ratio suddenly improves. We believe that, with more than 10 clients, the performance ratio would level off once the client/server implementation reaches a new slope.

In the 100 mbps curves, the performance ratio for both D'Agents and EMAA declines steadily as more clients are added. We believe that the agent approaches are more sensitive to the CPU overhead of CPU contention (as the CPU becomes loaded, its efficiency drops as more time is spent in context-switching).

Now, think of each of the plots in Figure 8 a slice in a larger three-dimensional plot. Each agent system's curve becomes a surface over the variation in bandwidth and the number of clients. Figures 9 and 10 show those three-dimensional curves, in both contour and 3-D format, for D'Agents and EMAA.

Each plot in Figure 9 shows the performance ratios for D'Agents. The x axis is bandwidth in megabits per second; note that this axis is neither logarithmic nor linear. The y axis is the number of clients, which also corresponds to the number of client machines. In all plots, the color represents the performance ratio. In the 3-D plots on the right, the z axis is the performance ratio. The bottom four plots show the same data; the two contour plots are identical and the two 3-D plots show two different viewing angles.

In the 5% graphs, we can see that the performance ratio peaks and then (for 1 mbps) falls as the number of clients increases. As we mention above, at this point the D'Agents implementation hits the network limitation and the performance ratio weakens.

In the 20% graphs (all four bottom graphs of Figure 9) there appear to be two peaks, but the lower-right graph helps to show that this may be a visual effect arising from a lack of data points in the 3-10 mbps range. The actual surface likely reflects a "ridge," such that the peak performance moves up in the number of clients as the bandwidth increases.

Similarly, each plot in Figure 10 shows the performance ratios for EMAA. These results are easier to interpret, showing higher performance ratio for lower bandwidths or larger numbers of clients.

5 Conclusion

In our experiments we have found that the scalability of mobile-agent systems, in comparison to an alternative client/server implementation of the same application, depends on many different environmental parameters. Overall, the three mobile-agent systems we examined scale reasonably well from 1 to 10 clients, but when we compare them to each other and to a client/server implementation they differ sometimes dramatically. The client/server implementation was highly dependent on sufficient network bandwidth. The agent implementations saved network bandwidth at the expense of increased server computation. Thus, the agent approaches fared best, relatively, in low-bandwidth situations.

The performance of NOMADS clearly suffered from the untuned virtual machine. The relative performance of EMAA and D'Agents varied depending on the mix of computation and network in the application, reflecting their different mix of overheads.

Our experiments need to be extended, to larger numbers of clients and to a wider range of computational tasks. They also need to be filled in, with more data points in critical regions of interest. Finally, we would like to obtain more detailed measurements of some of the computational overheads so that we can better understand the nature of the performance we see.

References

[CMMS00] Daria Chacón, John McCormick, Susan McGrath, and Craig Stoneking. Rapid application development using agent itinerary patterns. Technical Report Technical Report #01-01, Lockheed Martin Advanced Technology Laboratories, March 2000.

[GCK+00] Robert S. Gray, George Cybenko, David Kotz, Ronald A. Peterson, and Daniela Rus. D'Agents: Applications and performance of a mobile-agent system. Submitted to Software Practice and Experience, November 2000.

[GCKR00] Robert S. Gray, George Cybenko, David Kotz, and Daniela Rus. Mobile agents: Motivations and state of the art. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2000. To appear. Draft available as Technical Report TR2000-365, Department of Computer Science, Dartmouth College.

[GKCR98] Robert S. Gray, David Kotz, George Cybenko, and Daniela Rus. D'Agents: Security in a multiple-language, mobile-agent system. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.

[Gra97] Robert Gray. *Agent Tcl: A flexible and secure mobile-agent system*. PhD thesis, Dept. of Computer Science, Dartmouth College, June 1997. Available as Dartmouth Computer Science Technical Report TR98-327.

[MCW00] Susan McGrath, Daria Chacón, and Ken Whitebread. Intelligent mobile agents in the military domain. In *Proceedings of the Autonomous Agents 2000 Workshop on Agents in Industry*, Barcelona, Spain, 2000.

[SBB+00] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, and Renia Jeffers. Strong mobility and fine-grained resource control in NOMADS. In *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA2000)*, volume 1882 of *Lecture Notes in Computer Science*, pages 2–15, Zurich, Switzerland, September 2000. Springer-Verlag.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley, 1994.

[Whi96] James E. White. Telescript technology: Mobile agents. General Magic White Paper, 1996.

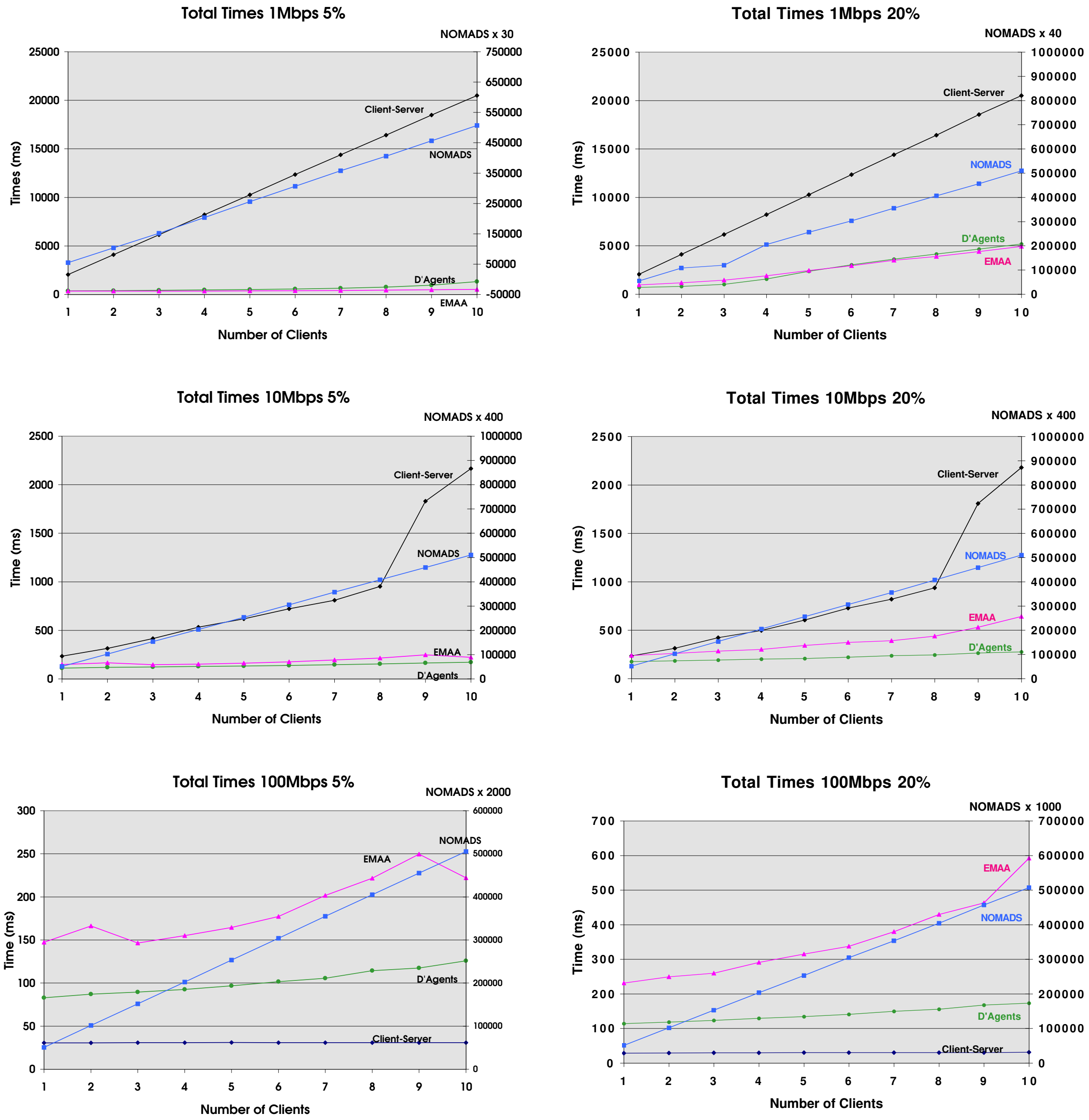

Figure 5: Total query times, for all systems, all three bandwidths, and both selection ratios. Note that the vertical scale varies. The NOMADS data should all be read using the scale on the right-hand side of the graph.

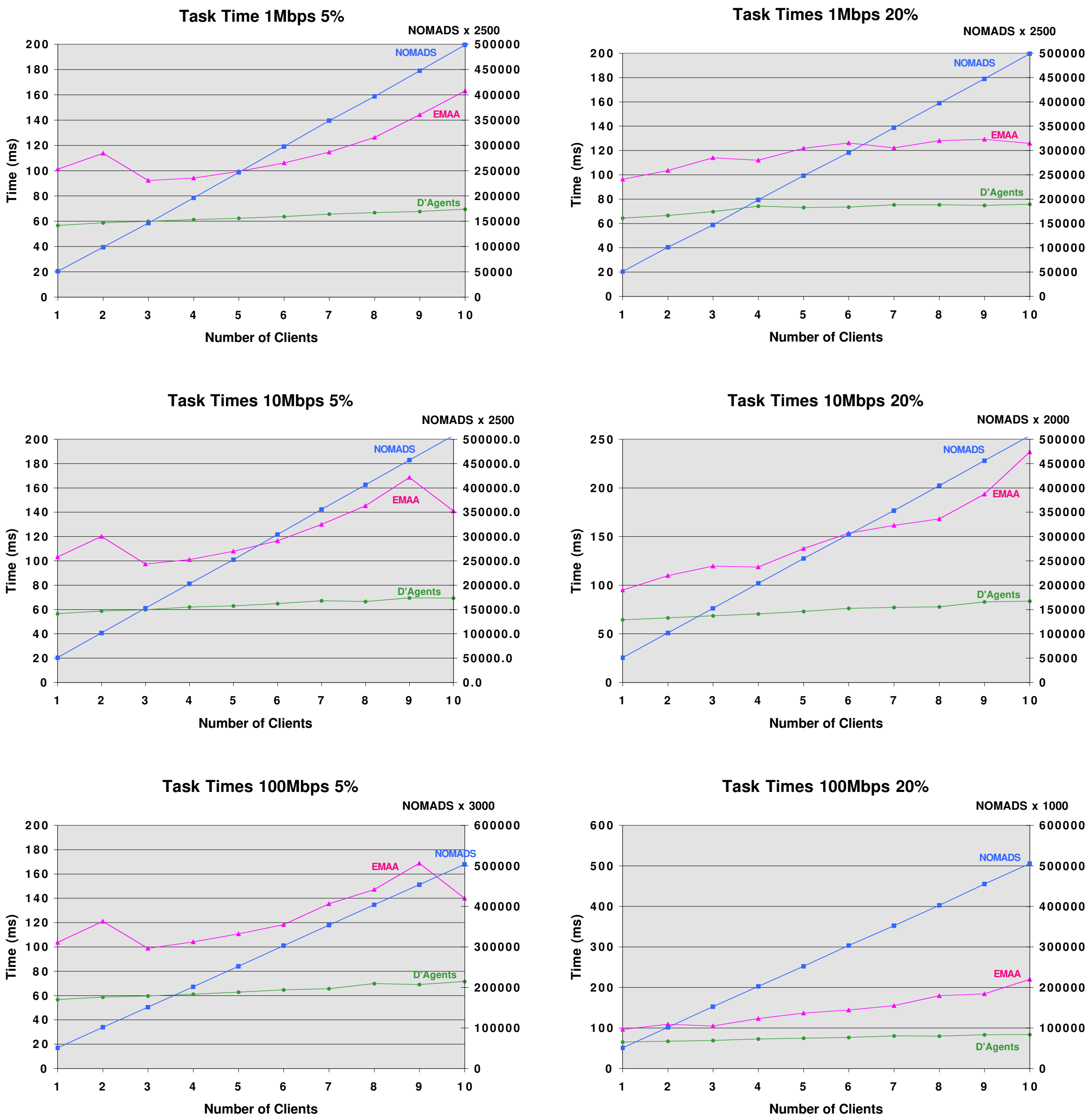

Figure 6: Task times, for all systems, all three bandwidths, and both selection ratios. Note that the vertical scale varies. The NOMADS data should all be read using the scale on the right-hand side of the graph.

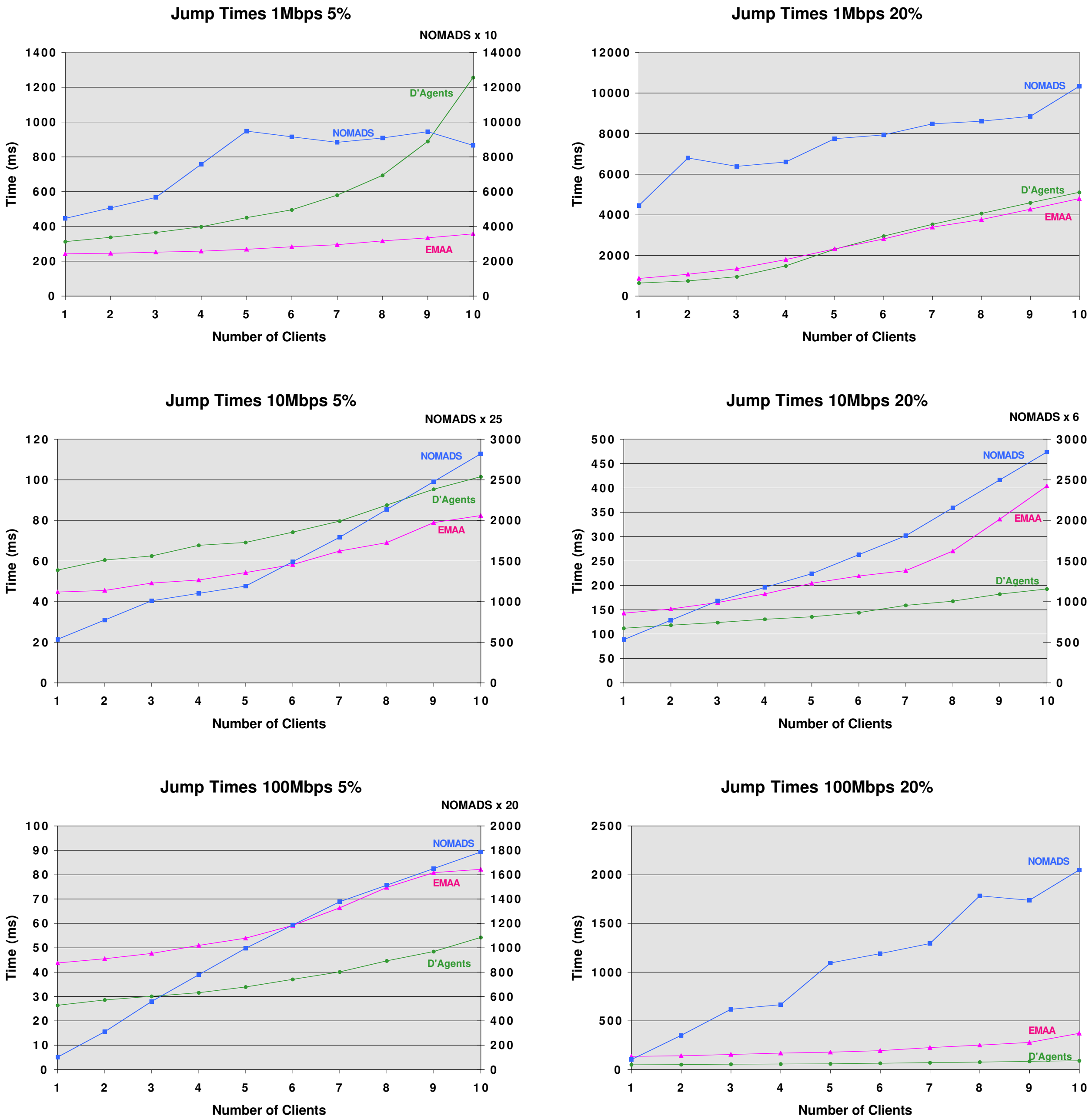

Figure 7: Jump times, for all systems, all three bandwidths, and both selection ratios. Note that the vertical scale varies. The NOMADS data should in many cases be read using the scale on the right-hand side of the graph.

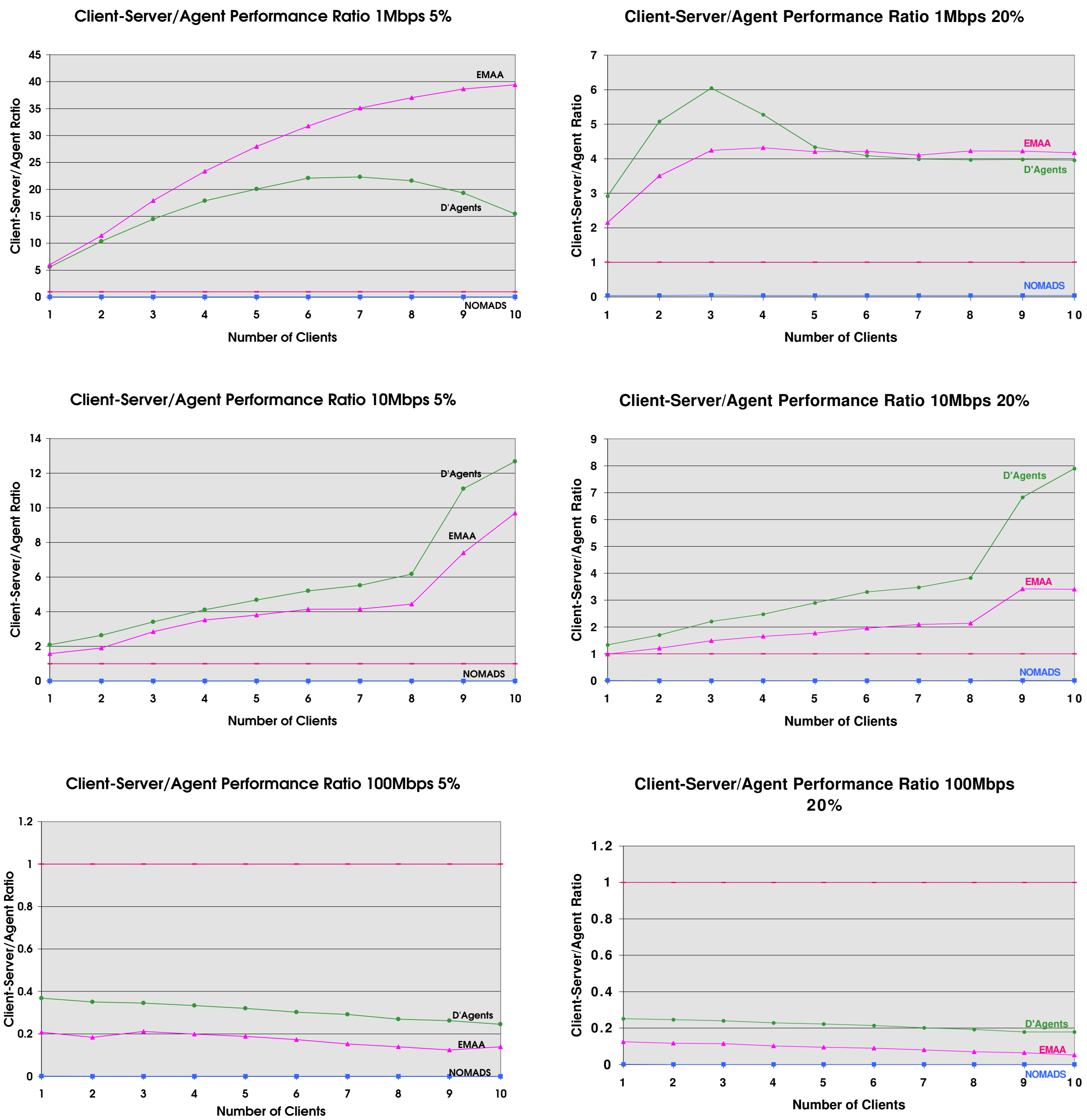

Figure 8: Performance ratios for all systems, for both selection ratios, combining all bandwidths on one plot. Note that the vertical scale varies.

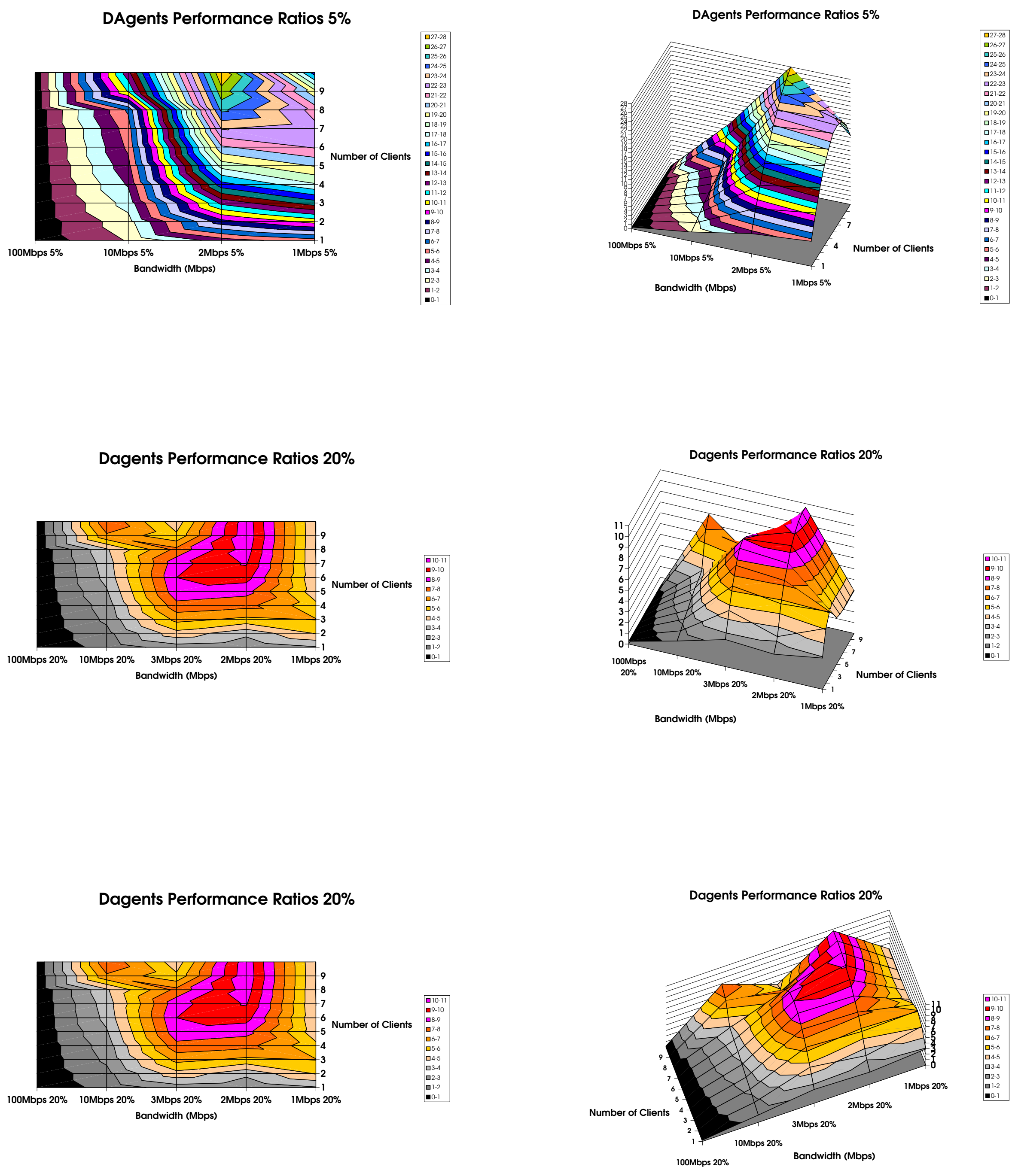

Figure 9: Performance ratios for D'Agents, for both selection ratios, combining all bandwidths on one plot. The second and third rows are all the same data presented from different views. Note that the bandwidth axis is neither linear nor logarithmic.

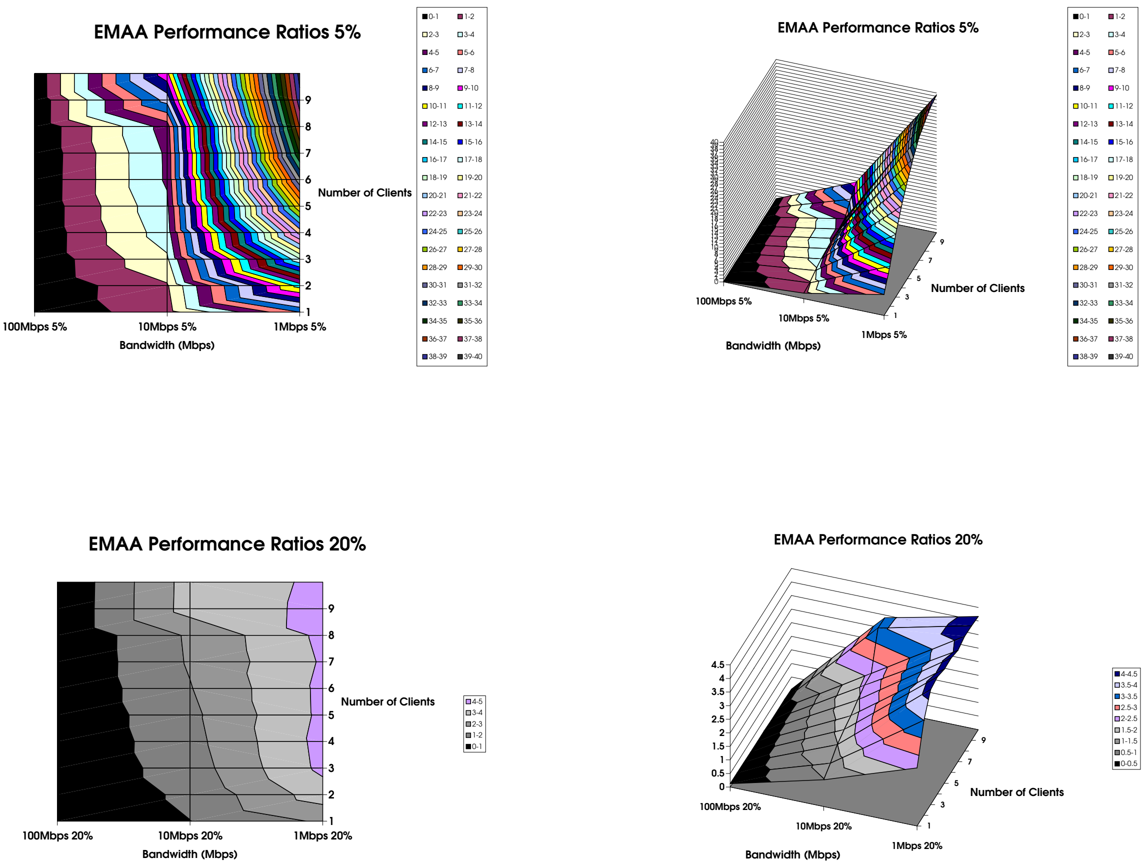

Figure 10: Performance ratios for EMAA, for both selection ratios, combining all bandwidths on one plot. Note that the bandwidth axis is neither linear nor logarithmic.