Dartmouth College Computer Science Technical Report
TR2004-499

Mobile Agents Simulation with DaSSF

Nikita E. Dubrovsky
Advisor: Robert S. Gray

June 9, 2004

Abstract

Mobile agents are programs that can migrate from machine to machine in a network of computers and have complete control over their movement. Since the performance space of mobile agents has not been characterized fully, assessing the effectiveness of using mobile agents over a traditional client/server approach currently requires implementing an agent system and running time-consuming experiments.

This report presents a simple mobile-agent simulation that can provide quick information on the performance and scalability of a generic information retrieval (IR) mobile-agent system under different network configurations. The simulation is built using the DaSSF and DaSSFNet frameworks, resulting in high performance and great configuration flexibility. This report also implements a real D'Agents mobile-agent IR system, measuring the performance of the system. A comparison of these real-world performance results and those given by the simulation suggest that the simulation has good accuracy in predicting the scalability of a mobile-agent system. Thus this report argues that simulation provides a good way to quickly assess the performance and scalability of an IR mobile-agent system under different network configurations.

1. Introduction

Mobile agents are programs that can migrate from machine to machine in a network of computers. An agent has complete control over the time and direction of its movement. Using mobile agents instead of traditional distributed approaches can significantly improve the performance of some applications, especially in low-bandwidth, low-reliability networks. A mobile-agent system will not always perform better than a client/server solution however, and the performance space for mobile agents has not been characterized fully. Therefore, assessing the effectiveness of using mobile agents over a traditional approach currently requires implementing an agent system and running time-consuming experiments.

To address this problem, this report presents a generic information retrieval (IR) mobile-agent simulation. The goals of the simulation are to provide a fast and simple way of assessing the performance and scalability of an IR mobile-agent system under different network topologies and conditions. The simulation is built using the DaSSF and DaSSFNet frameworks, to provide high performance and configuration flexibility. These simulation frameworks use DML for creating network models, allowing quick development of different network configurations.

The overall approach of using only simulation to evaluate MA performance has some obvious drawbacks, but also several advantages, making it a worthwhile research direction. Since MA is a complex system, it is difficult to achieve high accuracy with just simulation. Also, some experiments on a real MA system are still necessary to obtain parameters for the simulation (and usually, the more accurate a simulation is, the more parameters it requires). On the other hand, simulation has the advantages of great simplicity and speed. Using simulation, the performance of a system can easily and quickly be evaluated under many different network and system configurations.

The remainder of this report is divided into the following sections. Section 2 presents an overview of past research relating to mobile-agent performance, scalability and simulation. Section 3 presents the simulation software, while Section 4 describes the D'Agents mobile-agent system and the IR agent application used for the comparison experiments. Section 5 presents the experiments used to assess the accuracy of the simulation, and the experimental results and analysis. Finally, Section 6 presents the overall conclusions.

2. Related Work

This section outlines the existing research in the areas of mobile-agent performance, scalability and simulation. MA performance evaluation research extends into several different areas—the papers most relevant to this report are described below.

In [6] and [7], R. Gray et al use an information retrieval application similar to the one presented in this report to evaluate and compare the performance of mobile agents and an equivalent client/server solution. Both papers focus on a single-hop agent scenario. [6] compares four different MA platforms (D'Agents, NOMADS, EMAA, KAoS), using different query pass ratios[1] (5% and 20%), network speeds (1, 10, and 100Mbps), and number of clients (1 to 20). [7] presents a detailed study of the scalability of D'Agents under varying network speeds (1, 2, 3, 7, 10, 100 Mbps), query pass ratios (5% and 20%), and number of clients (1 to 20). The papers provide insight as to when mobile code should be considered as a solution.

Along the same lines, [8] presents a performance evaluation of mobile agents versus the client/server paradigm. The study is very practical in its approach, using "real networks" (as opposed to a dedicated experimental network), including Internet links connecting Switzerland, France and the UK. For the application, both single-hop and multi-hop agents were used to retrieve a list of hotels and corresponding phone numbers (stored on different servers). The system is analogous to the IR system used in [6] and [7], and it presents similar results. The experiments use only a single agent, however, and do not assess the scalability of the system under different loads.

In [3] and [4], Dikaiakos et al develop a framework of benchmarks for quantitatively evaluating MA performance. Their structured, hierarchical approach allows analysis of the performance of an MA platform and the identification of existing bottlenecks. Using this framework, the performance of a mobile-agent platform can be characterized, thus making it possible to then predict the performance of an MA

[1] Although all of the documents are searched, the query pass ratio is used to determine the percentage of documents that actually "pass" the query. This allows the number of documents that the query agent returns to be controlled exactly.

application implemented on top of the platform. This approach still requires a time-consuming process of implementing and running the benchmark experiments. Even the simple simulation presented in this report, however, requires some benchmark experiments to be conducted in order to obtain simulation parameters.

Kotz et al [10] and Woodside [17] take a mathematical approach to MA performance evaluation. [10] presents an analytical model to examine the tradeoffs associated with a mobile-agent system versus a traditional solution. The model is then parameterized using data from previously conducted filtering experiments, and the model's implications are discussed, confirming that mobile agents provide a significant performance benefit in a wide range of situations. In [17], Woodside introduces a model for scalability to analyze the mobile-agent paradigm. Performance is expressed in terms of the time it takes for an agent to execute its "tour" (a task that involves visiting several hosts). The number of servers and agents can then be varied to evaluate the scalability.

Finally, [1] and [11] present simulation frameworks for evaluating MA systems. In [1], Cao et al propose a direct-execution-simulation approach for evaluating mobile-agent performance. The paper describes a generic simulation model named MADES, and an implemented prototype of the model. The simulation environment allows a mobile-agent application to be directly executed, while the simulation collects and processes performance information about the application. A similar approach is taken by Liotta, Ragusa and Pavlou in [11]. They propose a hybrid simulation framework, which allows execution of prototype agent code over simulated networks. The framework uses *NS* for network simulation, while implementing a virtual execution environment for mobile agents. Synchronization between MA processing and network events was not yet fully developed however, making the simulation only suitable for MA systems whose processing is not tightly bound to networking events.

As this overview shows, this report takes a somewhat unique approach to MA performance evaluation, using only simulation with just a few simple parameters.

3 Simulation Software

This section describes the software pieces that make up the mobile-agent simulation. Section 3.1 and 3.2 present DaSSF and DaSSFNet, the two simulation frameworks on top of which the MA simulation is built. Section 3.3 provides a brief overview of the DML language used for model configuration. Finally, the last sections, 3.4 and 3.5, describe the MA simulation implementation and configuration.

3.1 DaSSF

DaSSF or Dartmouth Scalable Simulation Framework is a C++ implementation of the Scalable Simulation Framework (SSF). SSF is an object-oriented API for a public domain standard for discrete-event simulation. SSF was designed for parallel simulators and allows easy porting of simulation models across different platforms and applications. The SSF API provides a self-organizing, scalable simulation infrastructure. This infrastructure is based on five simple classes (*SSF_Entity*, *SSF_Process*, *InChannel*, *OutChannel*, and *Event*), which provide a generic framework for modeling systems as a collection of objects communicating via event exchange. The details of the SSF API can be found in [2].

DaSSF provides several advantages over other implementations. In the design of DaSSF, performance was a top priority, resulting in an implementation that is highly efficient in memory usage and provides high performance even for very complex systems. DaSSF is fast, portable and capable of simulating very large systems. The DaSSF homepage [12] provides a full description of the framework implementation. What DaSSF lacks however is a higher-level infrastructure specifically for simulating networks. This shortcoming was alleviated by the development of DaSSFNet, described below.

3.2 DaSSFNet

DaSSFNet implements a network simulation infrastructure. It is a framework built on top of SSF, consisting of a collection of simulation components for modeling communication networks. DaSSFNet provides a complete suite of network elements for simulating hosts, routers, protocols, interfaces, etc. The DaSSFNet framework has its own object model, providing classes representing specific network elements. Compared to DaSSF, which is a very generic framework, DaSSFNet provides an infrastructure specifically for network simulation, and allows the intuitive construction of large-scale network models from familiar, real network components. More extensive descriptions of DaSSFNet are available at [9] and [13].

DaSSFNet consists of the following main classes. The *Net* class is a container for all *Machine*s and *Link*s and other *Net*s. A *Machine* class models hosts and routers, as well as acting as a container for *Interface*s and *SSF_Protocol*s, providing utility functionality between the two. The *Link* and *Interface* classes simulate all communication. Finally, the *ProtoGraph* and *SSF_Protocol* classes describe the protocols running on a simulated host. *ProtoGraph* is simply a container for all protocols and is implemented as a DAG (Directed Acyclic Graph), since each protocol can have more than one protocol on top of it, but always only one below. *SSF_Protocol* is a base class for all simulated protocols. The *Hardware* class is a protocol that is at the root of the *ProtoGraph*. Protocols that are implemented as part of DaSSFNet include IP, TCP and UDP, sockets, and HTTP. These simulated protocols implement the familiar functions from their corresponding real-world counterparts, such as bind(), accept(), connect(), send(), recv(), etc for sockets. Thus, in order to model a particular application (for example a Web browser, or in this case, a mobile-agent client and server), one simply needs to implement a class or classes derived from both *SSF_Protocol* and *SSF_Process*, simulating the application functionality, and then add those classes to the *ProtoGraph* along with the lower-level protocols used by the application.

As a network simulator, DaSSFNet has several advantages. Since it also is implemented in C++, DaSSFNet is capable of running very large network models with high performance in terms of both memory usage and speed. More importantly for this mobile-agent simulation, DaSSFNet also allows the creation and configuration of simulated networks very easily and quickly through DML, as described in the next section. These advantages prompted the choice of using DaSSF and DaSSFNet as the simulation frameworks. Other simulation packages (such as *NS*) have comparable strengths, however, and the MA simulation does not depend on any features unique to a particular simulation package.

3.3 DML

Domain Modeling Language or DML is a simple language that allows the specification of properties for attributes. DML is the language used by DaSSFNet for model specification and configuration, allowing easy and intuitive creation of simulation networks. A DML expression is just a list of key value pairs separated by whitespace; keys are strings, while values can be either strings or other DML expressions. DaSSFNet then specifies a set of keys that are used to define a simulation model. [15] gives a full description of DaSSFNet/SSFNet DML attributes, while [14] and [16] provide detailed tutorials for defining simulation models for DaSSFNet using DML. Section 3.5 describes more details of configuration, specifically for the MA simulation.

3.4 Mobile Agents Simulation

The simulation models an IR mobile-agent system. Information retrieval is a common and relatively generic application for mobile agents. Thus, using this system for the model allows the simulation to be applicable to many different real-world applications. In an IR scenario, a client needs to obtain some information stored on a remote server. The client sends a query agent to the server, where the agent scans all the documents. The agent determines which documents are relevant based on its query string, and returns to the client with just those files.

In the simulation, agent clients generate query agents that jump to one or more servers to execute the query, then return to the client with the results of the query. The simulated agent servers wait for and receive agents, simulate processing the query, and finally let the agent jump to another server or back to its source client with the query's results.

The simulation is based on the following parameters: the size of the agent and query results, the agent serialization/un-serialization times for the client and server, the query execution time, and the number of servers an agent must visit to process the query. The different network scenarios that can be modeled by the simulation in DaSSFNet is practically unlimited.

The MA simulation code is split among four files: AgentsClient.h/.cc and AgentsServer.h/.cc. The implementation details for the simulated agent client and server are described below.

3.4.1 Simulation agent client.

The client code is divided among five classes: *AgentsClient*, *AgentInfoTable*, *AgentSender*, *AgentReceiver* and *ClientAgentHandler*. *AgentInfoTable* implements a simple dynamic array to keep track of generated agents, including their id, size and start time. *AgentsClient* is derived from the *SSF_Protocol* and *SSF_Process* classes, as it is both a DaSSFNet protocol and an individual simulation process. When the class is instantiated, SSF automatically calls its *Configure()*, *Initialize()* and *action()* functions, which serve as the entry point and whose implementation is required. *Configure()* simply reads in and stores the client's DML parameters, while *Initialize()* gets the client's IP address and opens a new socket interface. The simulated agent client then begins execution in the *action()* function. The client first creates a new *AgentReceiver* class to wait for and accept connections from servers as the agents return. The client then drops into a loop to generate agents. The loop first sleeps (using the SSF *waitFor()* function) for a random delay, calculated using an exponential distribution whose mean is obtained as a parameter from the DML configuration. A new agent then is generated by creating an entry in the *AgentInfoTable* and instantiating a new *AgentSender* object with the new agent's id passed as an argument. The agent's size is set according to the parameter specified in the client configuration. The loop ends when the number of generated agents reaches the cap set in the client's DML configuration. After that, the client sleeps until the end of the simulation. When the simulation ends, the client's destructor outputs the client's overall statistics for the run.

AgentSender, *AgentReceiver* and *ClientAgentHandler* are all derived from *SSF_Process*. These classes serve essentially as containers, as they merely call the corresponding *SendAgent()*, *ReceiveAgents()* and *HandleAgent()* functions of *AgentClient*when they are instantiated by the simulation. The reason why *AgentClient* does not call the functions directly, but goes through these helper classes, is so that SSF will create separate simulation processes for each of these actions and simulate their parallel execution.

The *SendAgent()* function records the start time for the agent and then takes care of simulating the agent's jump to the server. The server is selected at random from the list of possible servers specified in the traffic DML configuration parameter (see Section 3.5). The agent's serialization time is simulated by sleeping for the amount of time specified by the client's serialization time parameter (also in the DML configuration). The client then creates a new socket, opens a TCP connection to the server, sends the agent, and waits for an ACK (all simulated using the DaSSFNet socket protocol, which provides the familiar connect(), send() and recv() functions). The number of bytes sent and received for this agent jump is added to the client's global variables that keep track of the amount of incoming and outgoing data.

The *ReceiveAgents()* function simply creates a new socket and loops forever accepting connections. When a server attempts to connect, the function creates a new socket for the connection and instantiates a new *ClientAgentHandler* object, passing the socket to it as the argument. The *HandleAgent()* function called by the *ClientAgentHandler* class then takes care of receiving the agent trying to jump back to the client. After successfully receiving the agent, the function sends an ACK back to the server and closes the connection. As in *SendAgent()*, the client then simulates the agent un-serialization. Finally, the number of bytes received and sent, and the total time taken by the agent, are calculated and added to the client's overall statistics.

3.4.2 Simulation agent server.

The server code consists of just two classes: *AgentsServer* and *ServerAgentHandler*. After its initialization, which parallels that of the client (through the *Configure()* and *Initialize()* functions called automatically by SSF), the server starts executing in the *action()* function. That function is essentially identical to the *ReceiveAgents()* function of the client. The server loops forever accepting connections, creating new sockets and *ServerAgentHandler* objects so that a separate SSF process handles each connection. The *ServerAgentHandler* class then merely calls *AgentsServer*'s *HandleAgent()* function.

After receiving the agent, *HandleAgent()* simulates the agent un-serialization and query processing. In order to simulate the execution of these actions, a simple FIFO queue is used as the model for process scheduling and execution. A global semaphore represents the server CPU. *HandleAgent()* waits on the semaphore ("waits for the server CPU to be free"), simulates execution while holding it, and "releases the CPU" (i.e. signals the semaphore) when the execution is "finished." The execution itself is simulated as on the client, by sleeping for the amount of time specified by the corresponding parameter in the server DML configuration.

The advantage of this FIFO queue approach is its great simplicity, requiring only one parameter. Simplicity, however, is also the main disadvantage of the model, since it does not account for many of the factors involved in executing code, including the far more efficient process scheduling used by a real system, the overlap of I/O tasks with CPU usage, and caching. Still, this model provides adequate accuracy for agents that are not overly complex.

Initially, the simulation used this "CPU" approach only for simulating the query processing. After running experiments on D'Agents however, it was discovered that agent un-serialization becomes a major bottleneck for the agent system under heavier loads (see section 4.3). Therefore un-serialization is also simulated using the "server CPU," unlike the client's serialization/un-serialization. After thus "processing" the query, *HandleAgent()* adds the simulated results to the agent and sends it on to the next server or back to its source client. Serialization of the agent before sending it is simulated as on the client, by simply sleeping, without waiting to get the "CPU." This is because serialization time is very small (approximately 0.003 sec for the D'Agents IR application used) and never causes a bottleneck on the real D'Agents server (since the rate at which agents need to be sent is already involuntarily capped by the un-serialization and processing time). Using the "CPU" model for serialization in the simulation would therefore cause some agents to have unrealistically long execution times, if they simply got stuck behind another agent's query processing in the "CPU" queue.

3.5 Simulation Configuration

Through DML configuration, the simulation can easily and quickly be configured to use the desired network topology and characteristics. Section 3.1 contains several references to detailed DaSSFNet DML manuals. A sample MA simulation configuration is shown in Appendix A. On a higher level, the simulation's DML configurability allows the user to create simulated networks by specifying a set of hosts and routers connected by links. Every host or router's interface speed, latency and reliability can be specified individually. Additional static delays also can be assigned to links and interfaces individually. For every host, the user also specifies the simulated protocols running on it (in this case: IP, TCP, sockets, and either AgentsClient or AgentsServer). Finally, a traffic pattern parameter allows the user to define which hosts communicate with each other; for the MA simulation, this parameter defines, for each host, a list of servers to which the host can send its agent (a server then can be chosen at random for each agent). Thus, one can very easily construct different network configurations, and see the resulting effects on MA performance.

4. Experiment Software

This section outlines the real mobile-agent software used to evaluate the accuracy of the simulation. Section 4.1 describes the D'Agents mobile-agent system. Section 4.2 then explains the simple IR application implemented using D'Agents for the comparison experiments. Finally, Section 4.3 explains the D'Agents instrumentation that was done in order to obtain parameters for the simulation.

4.1 D'Agents

D'Agents is a mobile-agent system developed at Dartmouth College. D'Agents agents can be written in several languages, including Java and Tcl (Tool Command Language). Tcl is a powerful, yet easy to learn, high-level scripting language, allowing rapid development of complex agents. It was therefore the language chosen for the implementation of the mobile-agent application used in this report. A D'Agents Tcl agent has access to all of the standard Tcl commands, plus a set of special mobile-agent commands, allowing it to migrate from one machine to another, communicate with other agents, and obtain information about its current state (such as network location and identities of other agents, etc). D'Agents provides a very effective platform for experimentation, especially since the system's source code is available, allowing instrumentation to get more precise time measurements. Details of the D'Agents system (previously known as Agent Tcl) can be found in [5].

In addition to using D'Agents for assessing the accuracy of the simulation, the system also was used as reference for the development of the simulation. Most important for the simulation is the way in which an agent accomplishes migration. D'Agents implements an *agent_jump* command to handle this task. The command captures the current state of the agent, packages the state image, and sends this agent image to the destination host (over a TCP connection). The destination receives the image and starts a Tcl interpreter, which restores the agent's state and resumes the agent execution immediately after the jump command (strong mobility). The process of capturing state and packaging the agent and its reverse are referred to as agent serialization and un-serialization, both of which are modeled by the simulation.

4.2 D'Agents Query Agent

The experimental IR system used for evaluating the simulation consists of a main parent agent that generates child Query agents. The Query agents jump to one or more servers and conduct a simple text search on several test documents, returning to the client with only the documents satisfying the query.

The parent agent consists mainly of a simple loop that submits new Query agents and receives messages from returning agents. A random delay between each new agent is calculated using an exponential distribution, given a mean delay parameter. The loop ends when the parent reaches the cap on either the number of agents to generate or the experiment duration. Messages from any remaining agents that have not yet finished are then received, and the overall statistics for the run are calculated and output.

The Query agent's parameters include: a list of servers, the path to the test documents directory, and a relevance percentage specifying how many of the documents should end up satisfying the query (all of the test documents are searched, but this parameter is used to actually determine how many of them should "satisfy the query" for experimental control). The Query agent visits the same number of severs as is contained in the list of servers argument, but it does not necessarily visit each server exactly once, since the next server to which to jump is always picked at random. For the single-hop scenario, only one server is specified, so that all agents go to the same server. After jumping to the server, the agent reads in all of the test documents and runs a string search on each one. It then keeps just the number of documents specified by the relevance percentage parameter (all documents have the same size) and discards the rest. When visiting multiple servers, the Query agent replaces the documents obtained on a previous server with those resulting from running the query on the next server. Once the Query agent returns to its home machine, it sends a message to the parent agent, containing the timing information for the query, including the total round trip time.

4.3 D'Agents Instrumentation

The D'Agents system code was instrumented in order to obtain agent timing information to parameterize the simulation. To determine how much time agent serialization and un-serialization takes, logging was added to the D'Agents server. The server obtained and output a timestamp just after an agent image was read from the network and a timestamp just before writing an agent image to the network. The size of the incoming and outgoing agents also was output. The Query agent itself output the time when query processing starts (right after the *agent_jump* command returns), as well as the time when the query finishes. Subtracting the times logged by the instrumented D'Agents server and the Query agent provided the time for agent serialization and un-serialization. These values, along with the agent size, then were used as parameters for the simulation.

The D'Agents instrumentation also provided insight to at least one of the bottlenecks faced by the agent server under heavier loads. Below are two snapshots of the server log, one under normal load and another under heavy load. The logs record the time (in sec), agent id, and event. The un-serialization time is calculated by subtracting the "Agent Received" time from the "Query Exec Start" time.

Normal Load (0.5 agents / sec)

Timestamp	**Agent Id**	**Event**
1084076223.673090	1927	Query Exec End
1084076223.749990	1928	Agent Received
1084076223.788000	1928	Query Exec Start
1084076223.918370	1928	Query Exec End
1084076223.949860	1929	Agent Received
1084076223.987550	1929	Query Exec Start
1084076224.122450	1929	Query Exec End
1084076224.186730	1930	Agent Received
1084076224.163540	1930	Query Exec Start
1084076224.307960	1930	Query Exec End
1084076224.301630	1931	Agent Received
1084076224.321210	1932	Agent Received
1084076224.355740	1931	Query Exec Start
1084076224.395940	1932	Query Exec Start
1084076224.656220	1932	Query Exec End
1084076224.663750	1931	Query Exec End

Heavy Load (1.5 agents / sec)

Timestamp	Agent Id	Event
1084075887.227360	1902	Query Exec End
1084075885.552780	1904	Agent Received
1084075885.554390	1905	Agent Received
1084075885.556030	1906	Agent Received
1084075885.557720	1907	Agent Received
1084075885.559370	1908	Agent Received
1084075885.562690	1909	Agent Received
1084075885.564390	1910	Agent Received
1084075885.566070	1911	Agent Received
1084075885.567790	1912	Agent Received
1084075885.578840	1913	Agent Received
1084075885.577130	1914	Agent Received
1084075885.575430	1915	Agent Received
1084075885.573740	1916	Agent Received
1084075885.572060	1917	Agent Received
1084075885.569490	1918	Agent Received
1084075885.580560	1919	Agent Received
1084075885.561040	1920	Agent Received
1084075885.992530	1903	Agent Received
1084075887.887150	1917	Query Exec Start
1084075887.644960	1916	Query Exec Start
1084075887.727300	1913	Query Exec Start
1084075888.076500	1911	Query Exec Start
1084075888.155040	1910	Query Exec Start
1084075887.505990	1909	Query Exec Start
1084075887.395620	1907	Query Exec Start
1084075887.448290	1905	Query Exec Start
1084075887.474690	1904	Query Exec Start

Under a normal load the un-serialization time is approximately 0.05 sec. As the load on the server increases, however, the time explodes to about 2 sec. This is likely due to high CPU contention, since creating a new interpreter for each new agent that arrives is a CPU intensive task. This observation prompted the simulation of agent un-serialization on the server using a simulated CPU, described in Section 3.4.2.

5. Experiments and Results

This section describes the experiments that were run on the D'Agents system and simulation, in order to evaluate the simulation's accuracy. Section 5.1 gives an overview of the experiment scenarios, while Section 5.2 presents the results. The results are discussed in Section 5.3.

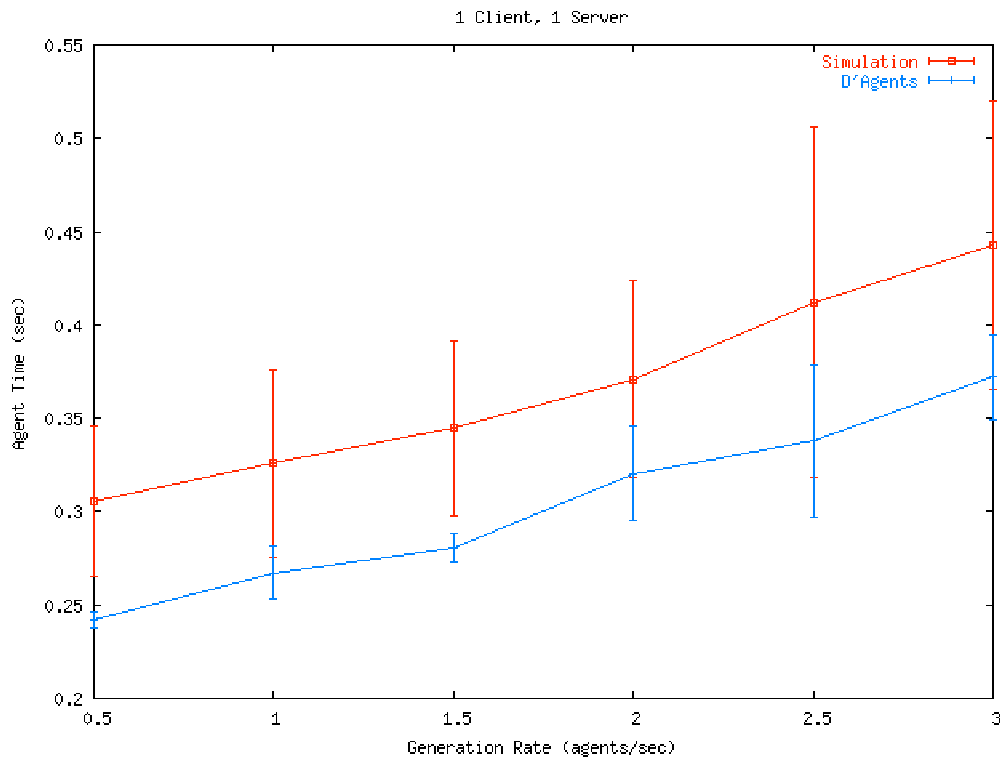
5.1 Experimental Design and Setup

In addition to a single-client, single-server baseline experiment, two main scenarios were used for the experiments. The first scenario varies the number of client hosts generating a single-hop agent that visits only one server to execute the query. The second scenario uses a single client that generates multi-hop agents, varying the number of servers that the agent has to visit to process the query. Having access to only 11 physical machines[2], this resulted in the following network configurations: "1 server, {2, 4, 6, 8, 10} clients" and "1 client, {2, 4, 6, 8, 10} servers".

5.2 Results

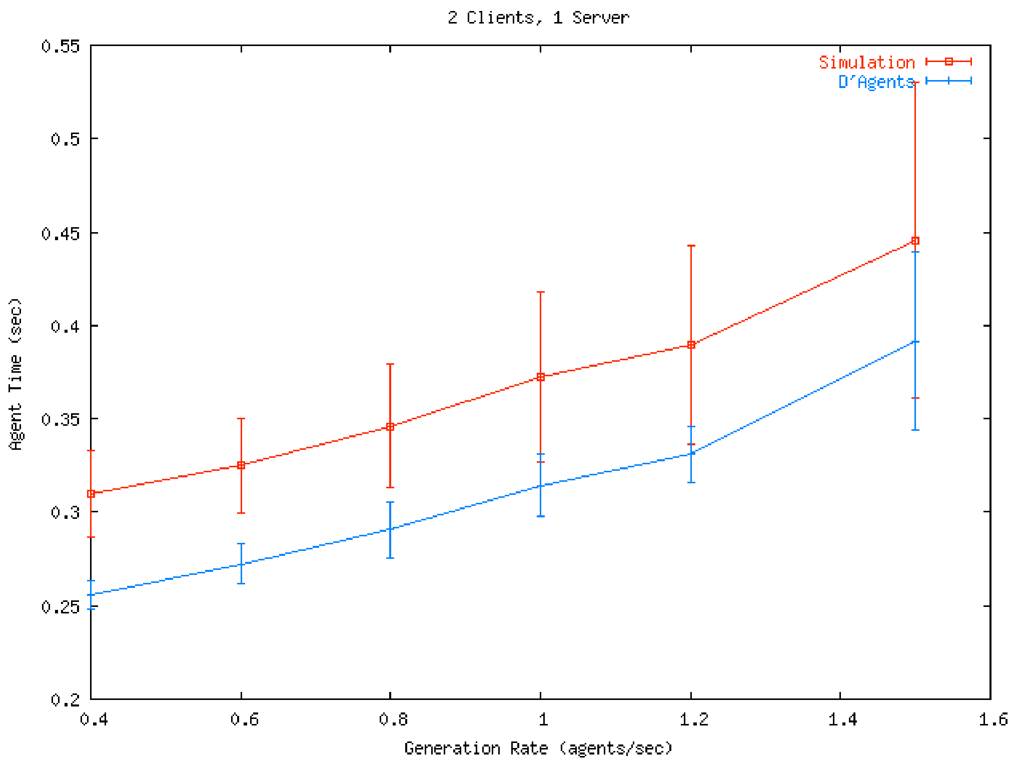
For each experiment, every client was set to generate a total of 100 agents, while the agent generation rate was varied. Five to ten trials were run for each generation rate. The "agent time" (the whole roundtrip time of an agent) was then obtained by averaging across all the agents. Each of the plots below shows the agent time corresponding to the generation rate used. Appendix B contains tables summarizing the results.

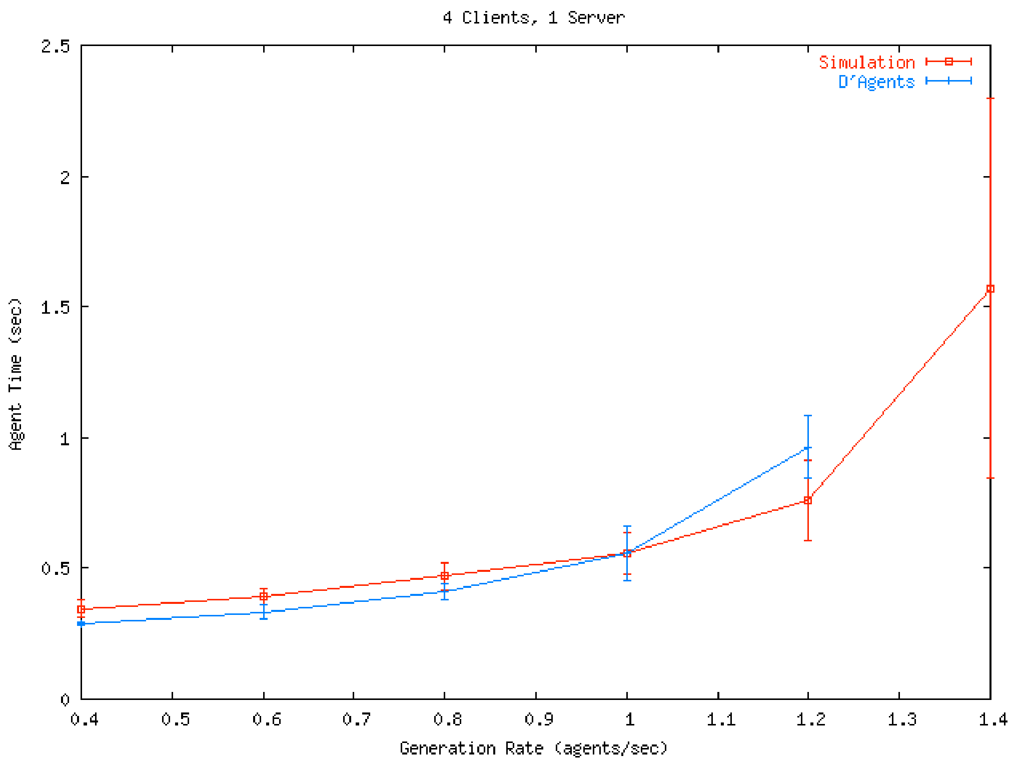
Baseline

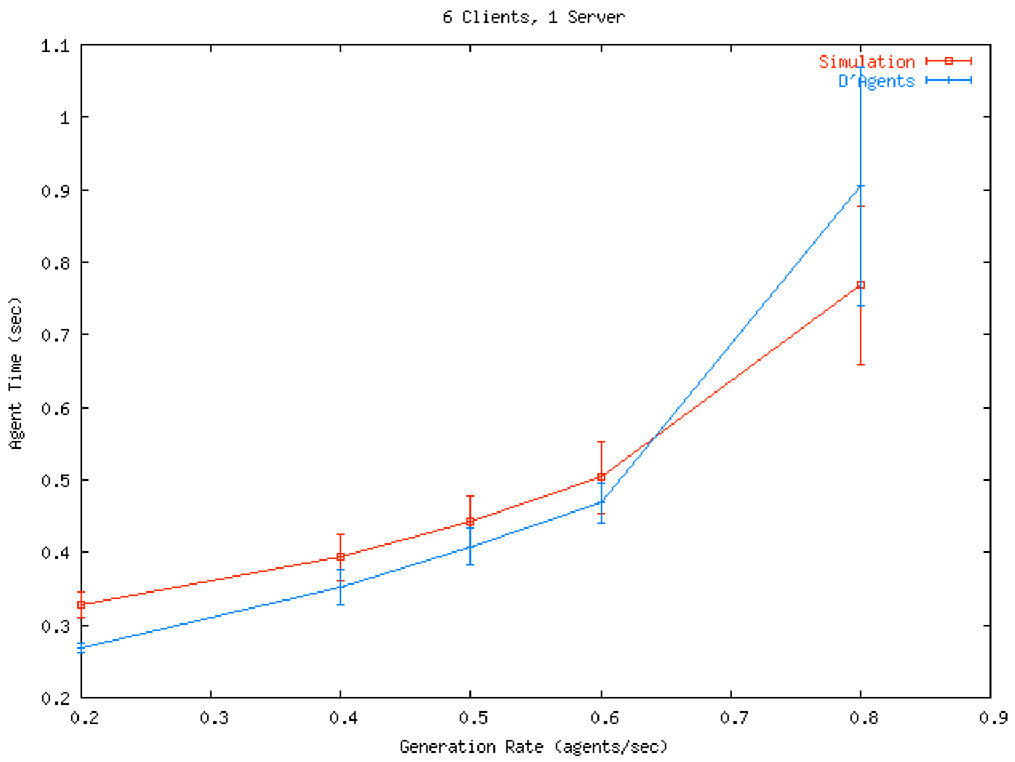


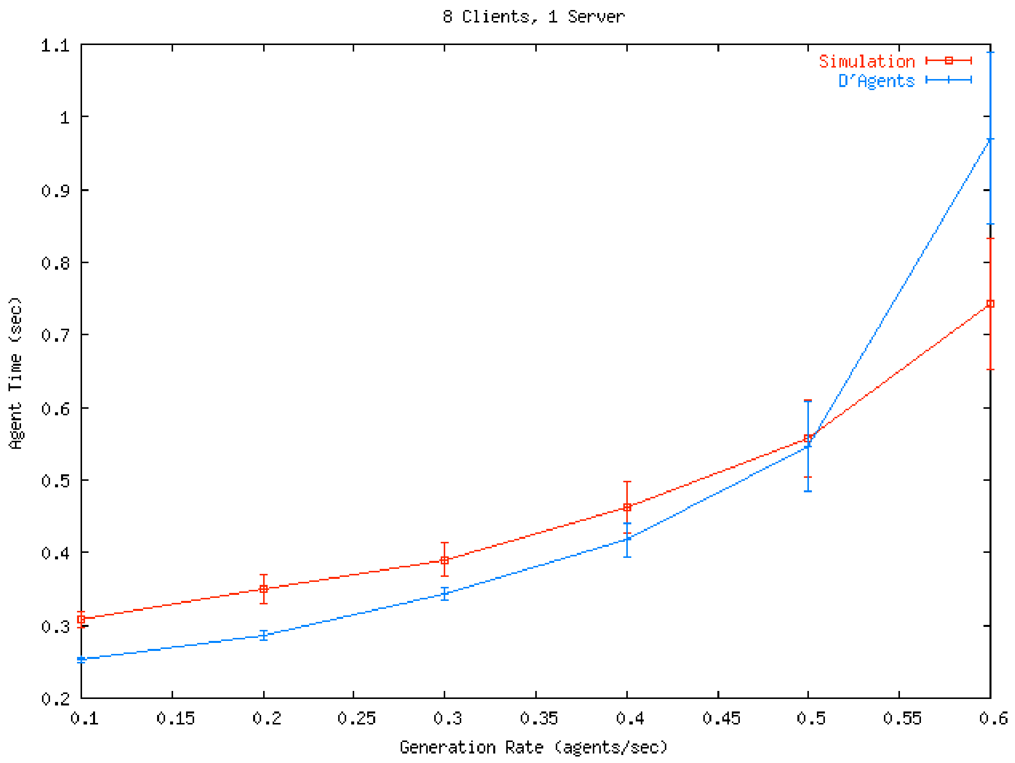
[2] The machines used were Gateway Solo 9300 laptops with 128MB RAM, 20GB hard drives, and 3COM 10/100Mbps Ethernet cards. The laptops were running Linux kernel 2.2.19 and PCMCIA version 3.2.4. The machines were connected through a 100Mbps hub on a dedicated network.

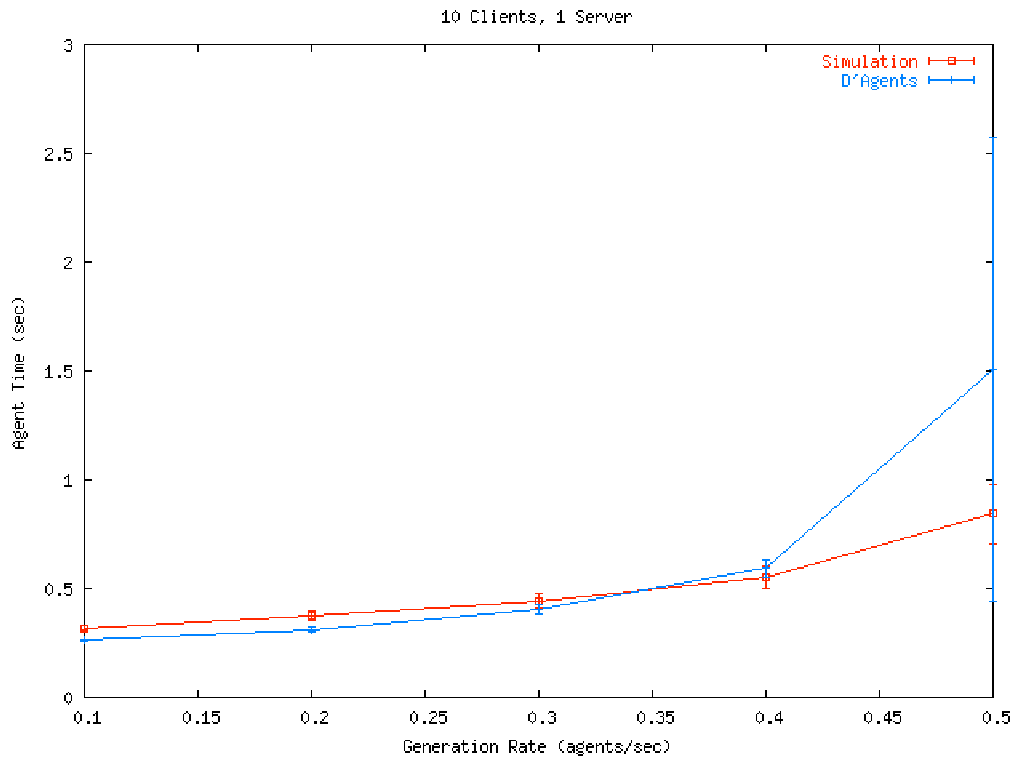
Single-Hop


2 Clients, 1 Server
Simulation
D'Agents
Agent Time (sec)
Generation Rate (agents/sec)
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6

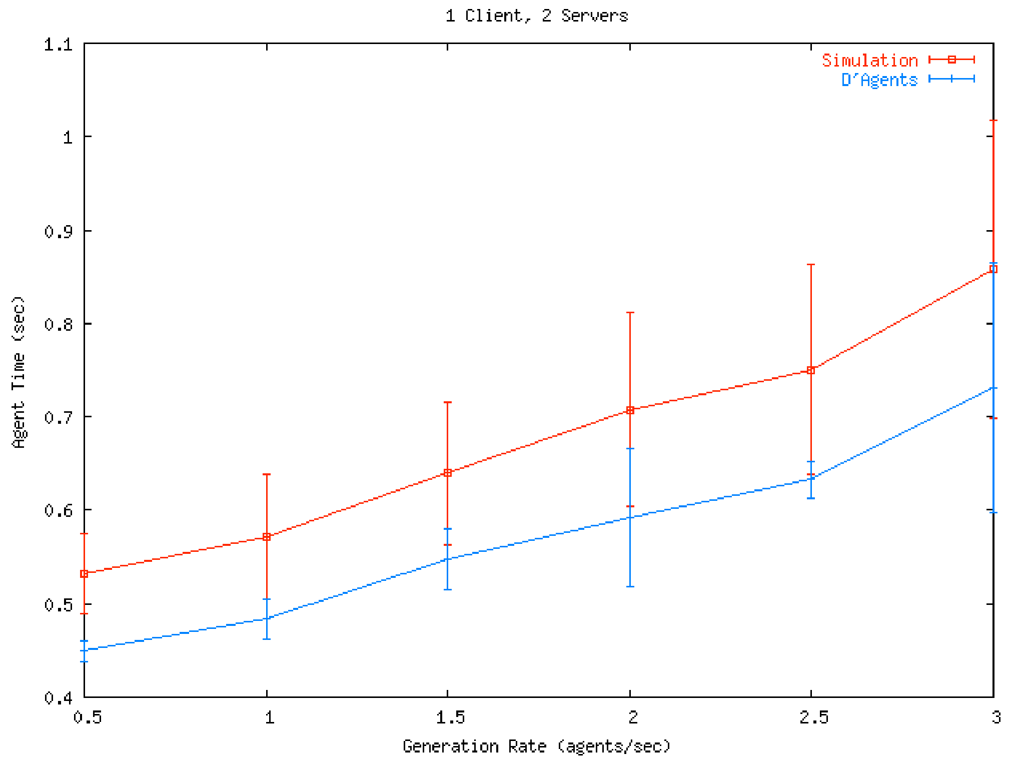

4 Clients, 1 Server
Simulation
D'Agents
Agent Time (sec)
Generation Rate (agents/sec)
2.5
2
1.5
1
0.5
0
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
1.2
1.3
1.4

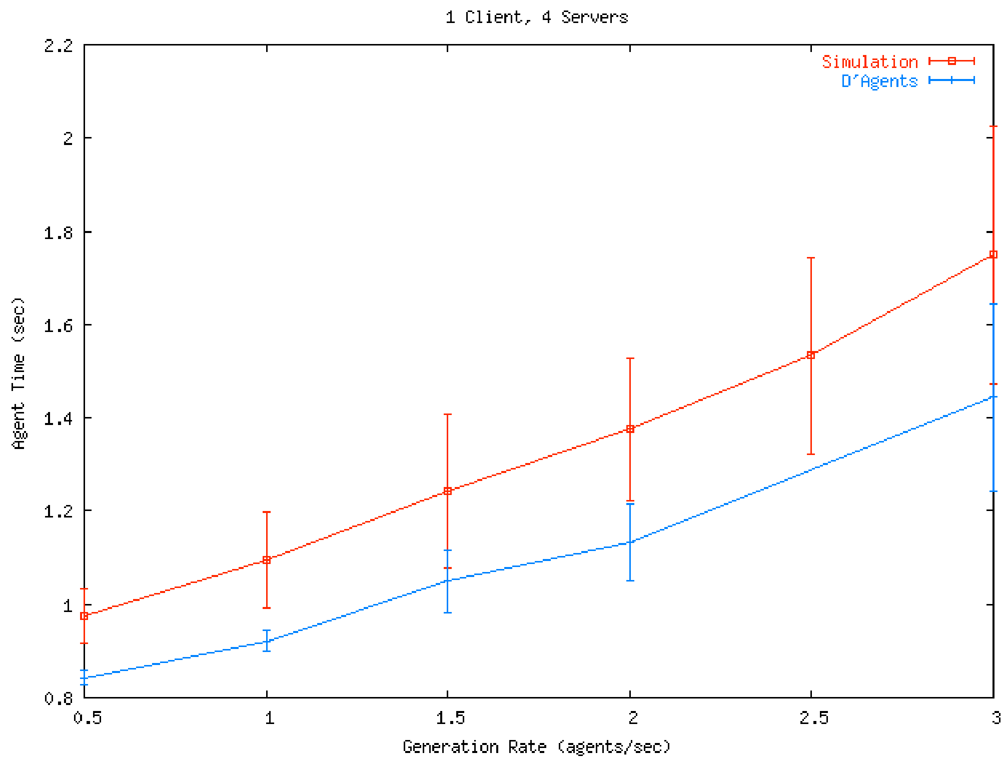

6 Clients, 1 Server
Simulation
D'Agents
Agent Time (sec)
Generation Rate (agents/sec)
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1

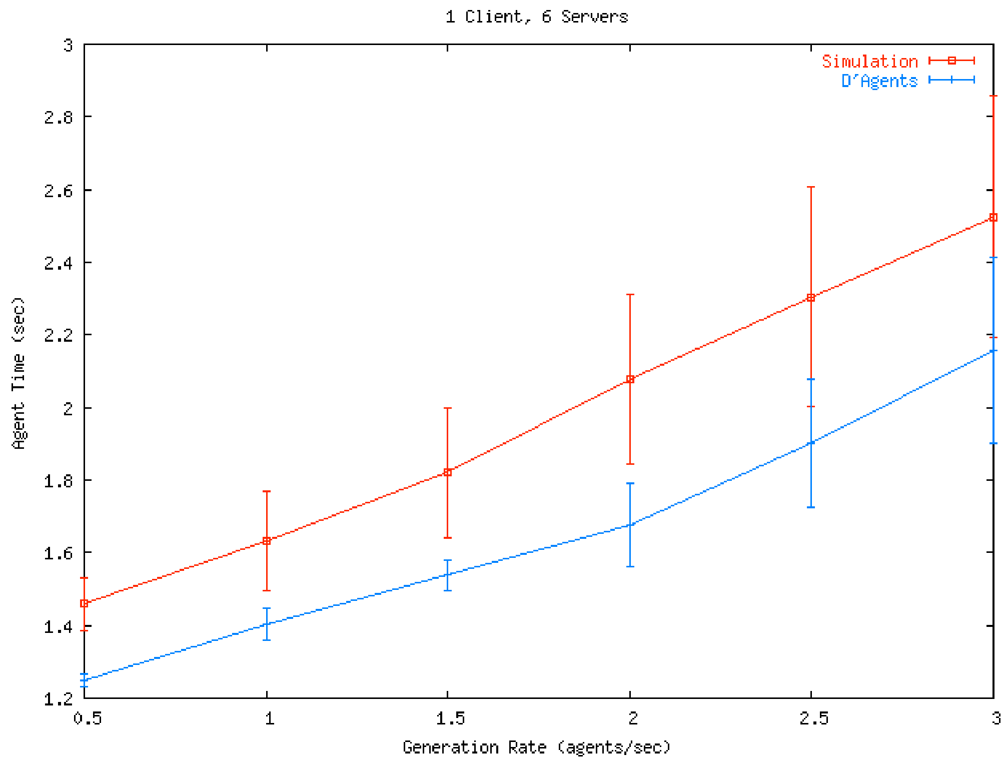

8 Clients, 1 Server
Simulation
D'Agents
Agent Time (sec)
Generation Rate (agents/sec)
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0.55
0.6

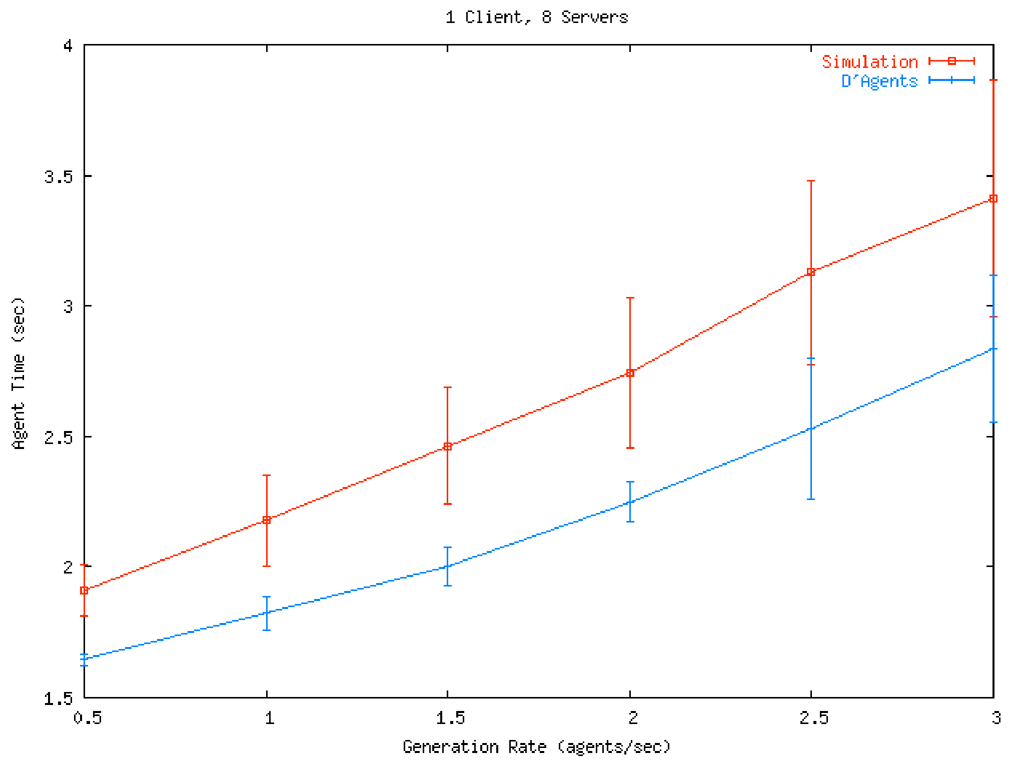

10 Clients, 1 Server
Agent Time (sec)
Generation Rate (agents/sec)
Simulation
D'Agents
0
0.5
1
1.5
2
2.5
3
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5

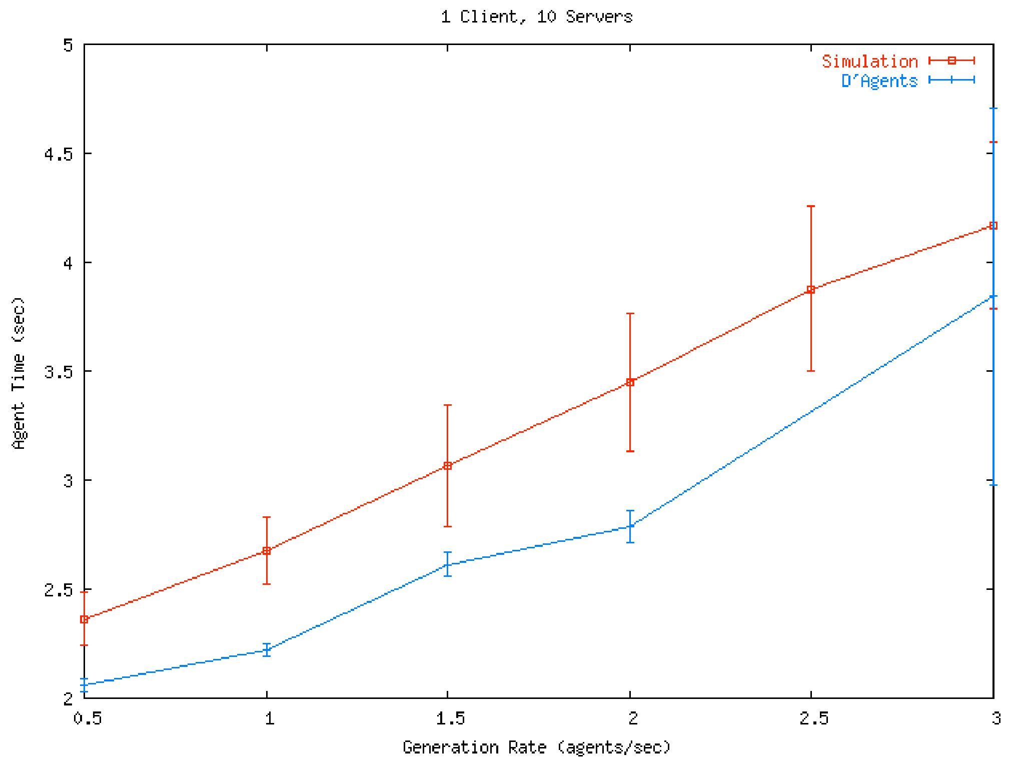
Multi-Hop


1 Client, 2 Servers
Agent Time (sec)
Generation Rate (agents/sec)
Simulation
D'Agents
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
0.5
1
1.5
2
2.5
3

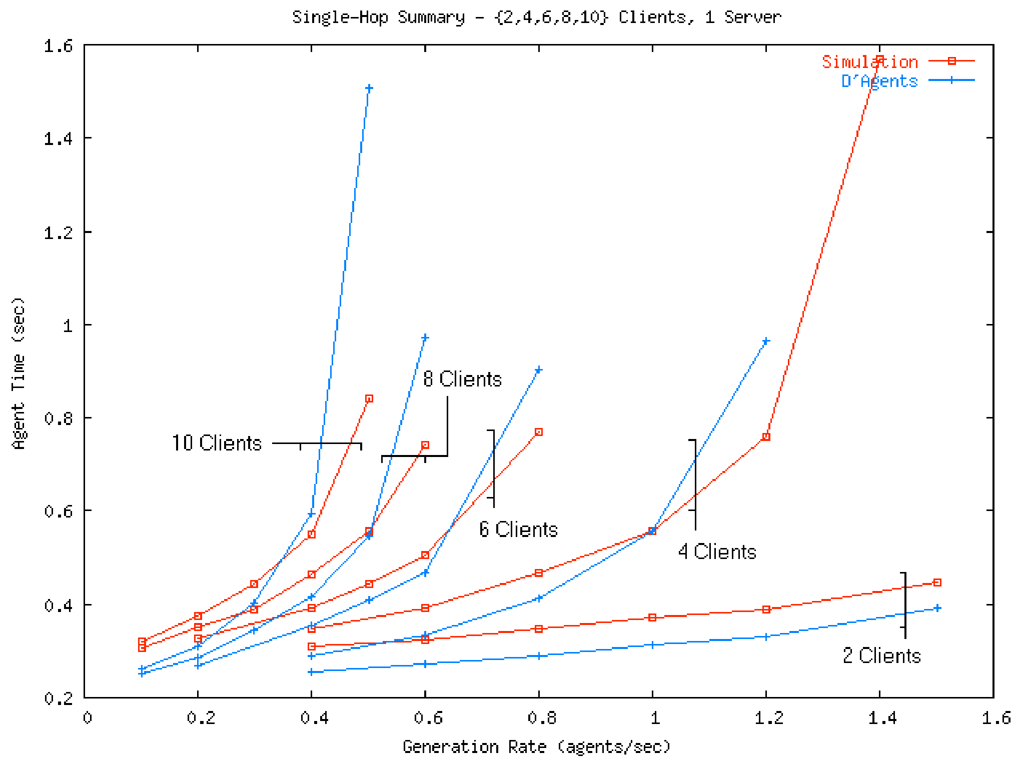

1 Client, 4 Servers
Simulation
D'Agents
Agent Time (sec)
Generation Rate (agents/sec)
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.5
1
1.5
2
2.5
3

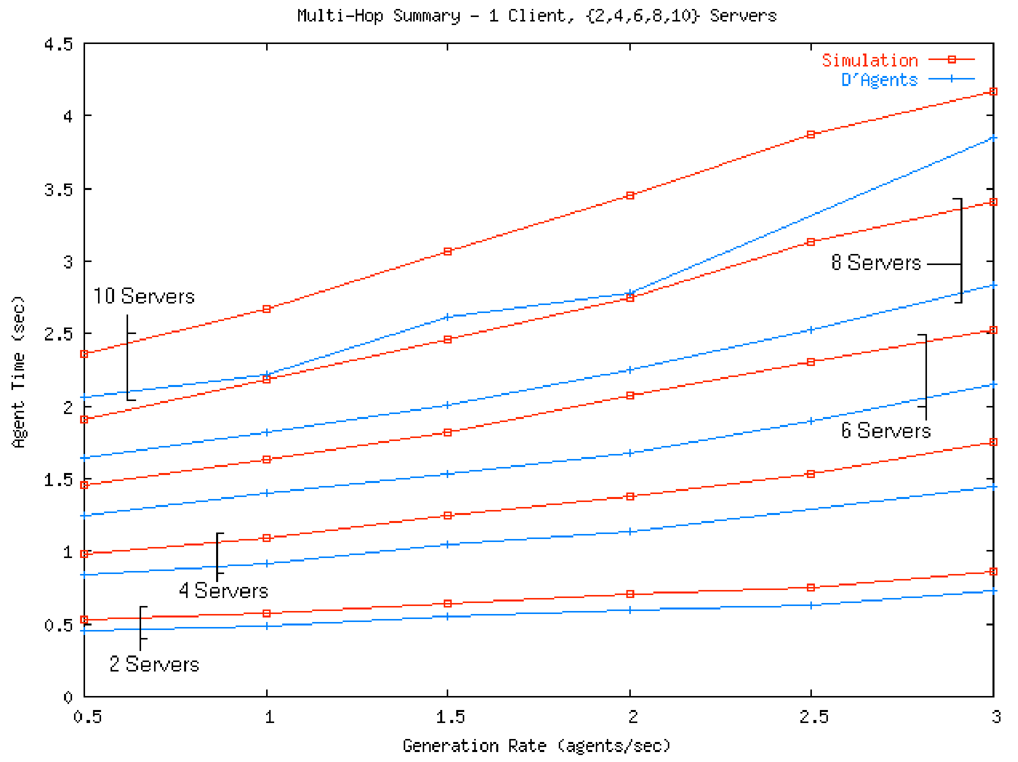

1 Client, 6 Servers
Simulation
D'Agents
Agent Time (sec)
Generation Rate (agents/sec)
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
0.5
1
1.5
2
2.5
3


1 Client, 8 Servers
Simulation
D'Agents
Agent Time (sec)
Generation Rate (agents/sec)
4
3.5
3
2.5
2
1.5
0.5
1
1.5
2
2.5
3


1 Client, 10 Servers
Simulation
D'Agents
Agent Time (sec)
Generation Rate (agents/sec)
5
4.5
4
3.5
3
2.5
2
0.5
1
1.5
2
2.5
3

Summaries


Single-Hop Summary - {2,4,6,8,10} Clients, 1 Server
Simulation
D'Agents
10 Clients
8 Clients
6 Clients
4 Clients
2 Clients
Agent Time (sec)
Generation Rate (agents/sec)
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6


Multi-Hop Summary - 1 Client, {2,4,6,8,10} Servers
Simulation
D'Agents
10 Servers
8 Servers
6 Servers
4 Servers
2 Servers
Agent Time (sec)
Generation Rate (agents/sec)
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3

5.3 Discussion

The agent times predicted by the simulation are clearly slightly off from the real-world performance results produced by D'Agents. The plots also show, however, that the simulation is off fairly consistently. For the single-hop experiments, the simulation is off by about 0.06 sec, until the server starts getting overloaded. When the load starts becoming too heavy for the server, the real-world time catches up with the simulation and explodes slightly before the simulation prediction. For the multi-hop experiments, the simulation predicts higher agent times across all generation rates, and the difference between the simulation and D'Agents grows as the number of servers increases.

There are several possible reason for the simulation's higher values. First, the simple FIFO queue model used for the "server CPU" to simulate un-serialization and query execution is certainly responsible for at least some of the extra time predicted by the simulation, since such a model is not nearly as efficient as real process scheduling. Another possible cause is that the network settings used for the simulation may not be completely accurate. The exact speed, latencies and static delays on the different parts of real network were not measured, so the values used to configure the simulation are best estimates. Therefore, the simulation's network configuration may be slightly off from the real network, causing higher agent time predictions. Finally, another reason for the lower D'Agents times is the caching in the real system, which would improve the speed of processing agents and executing the queries, as the same process is repeated again and again.

Since the simulation times are higher for the single-hop experiments, it makes sense for the gap between the simulation and real-world values to grow as the number of servers and agent migrations is increased. In the multi-hop experiments, the agents do both more network travel and more query processing; therefore, the offset already present in the single-hop experiments accumulates for every extra server, resulting in higher discrepancies.

Although the actual times predicted by the simulation do differ from the real-world results, the experiments also show that the simulation provides good accuracy in predicting the scalability of an agent system. As can be seen from the summary graphs, the simulation produces curves that closely parallel those produced by the real mobile agents system. Thus, even though the simulation is not completely accurate in predicting the exact performance of an agents system, it does make good predictions as to the scalability of the system under varying network and load conditions.

6. Conclusion

Since the performance space of mobile agents has not been characterized fully, there is currently no easy way of determining how well a mobile-agent solution will perform, and whether it is better than a traditional client/server approach. To address this problem, this report presents a simple mobile-agent simulation. The simulation models a generic information-retrieval agent system, since IR is a common area of mobile-agent application, allowing the simulation to be relevant to many different problems. Although the simulation cannot predict the exact performance of a system, the experiments presented in this report show that the simulation provides good accuracy in predicting the overall performance and scalability of a mobile-agent system. In particular, the simulation does well in predicting the scalability of an agent system under varying network and load configurations. Finally, the greatest advantage provided by the simulation is the ability to quickly and easily evaluate a mobile-agent system. Through DML configuration, the simulation allows for fast development of different network models and their evaluation. Additionally, since the simulation is built on top of the high-performance DaSSF and DaSSFNet frameworks, it provides high speed and efficient memory usage when assessing complex models. Therefore, this report argues that simulation provides a good tool for quickly evaluating and predicting the overall performance of a mobile-agent system under varying network configurations.

There are also several areas where additional work could make the simulation an even better alternative for evaluating mobile-agent performance and scalability. First, more accurate simulation of execution on the server would greatly improve the overall accuracy of the simulation. Even a simple round-robin model for simulating process scheduling (instead of the FIFO queue) should improve accuracy. Secondly, the MA simulation would provide more precise information on agent performance if more of the different agent platform pieces are simulated. The model presented here only simulates the agent serialization/un-serialization performed by the agent platform. Breaking up these general processes into their smaller sub-tasks and simulating them individually would provide better accuracy and allow the simulation to predict the scalability of the different platform parts. And thirdly, development of an effective simulation (or mathematical model) for the equivalent, traditional client/server application would allow fast comparison of the two paradigms under varying conditions.

References

[1] Jiannong Cao, Xuhui Li, So King, Yanxiang He, "Direct Execution Simulation of Mobile Agent Algorithms", *Proc. 2003 Internationbal Symposium on Parallel and Distributed Processing and Applications* (ISPA2003), July 2-4, 2003, Aizu-Wakamatsu City, Japan. Lecture Notes in Computer Science (Springer-Verlag).

[2] J. H. Cowie, ed. *Scalable Simulation Framework API Reference Manual* (1998). Available at http://www.ssfnet.org/SSFdocs/ssfapiManual.pdf (1 June 2004).

[3] M. Dikaiakos, M. Kyriakou, G. Samaras, "Performance Evaluation of Mobile-agent Middleware: A Hierarchical Approach," in *Proceedings of the 5th IEEE International Conference on Mobile Agents*, J.P. Picco (ed.), Lecture Notes of Computer Science series, vol. 2240, pages 244-259, Springer, Atlanta, USA, December 2001.

[4] M. Dikaiakos, G. Samaras, "A Performance Analysis Framework for Mobile-Agent Systems," in *Infrastructure for Agents, Multi-Agent Systems, and Scaleable Multi-Agent Systems, Proceedings of the First Annual Workshop on Infrastructure for Scalable Multi-Agent Systems, The Fourth International Conference on Autonomous Agents 2000*, Wagner and Rana (Eds.), Lecture Notes in Computer Science, Vol. 1887, pages 180-187, Springer, July 2001.

[5] R. 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 TR1998-327.

[6] Robert S. Gray and George Cybenko and David Kotz and Ronald A. Peterson and Daniela Rus. “D'Agents: Applications and Performance of a Mobile-Agent System.” *Software-- Practice and Experience*, 32(6):543-573, May, 2002.

[7] Robert S. Gray, David Kotz, Ronald A. Peterson, Joyce Barton, Daria Chacon, Peter Gerken, Martin Hofmann, Jeffrey Bradshaw, Maggie R. Breedy, Renia Jeffers, and Niranjan Suri. “Mobile-Agent versus Client/Server Performance: Scalability in an Information-Retrieval Task.” In *Mobile Agents*, pages 229-243, 2001.

[8] L. Ismail and D. Hagimont. “A performance evaluation of the mobile agent paradigm.” *ACM SIGPLAN Notices*, 34(10):306-313, October 1999.

[9] M. Iyigun. *DaSSFNet: An Extension to DaSSF for High-Performance Network Modeling*. Undergraduate honors thesis, Dept. of Computer Science, Dartmouth College, June 2001. Available as Dartmouth Computer Science Technical Report TR2001-405.

[10] David Kotz and George Cybenko and Robert S. Gray and Guofei Jiang and Ronald A. Peterson and Martin O. Hofmann and Daria A. Chacón and Kenneth R. Whitebread and James Hendler. “Performance Analysis of Mobile Agents for Filtering Data Streams on Wireless Networks.” *Mobile Networks and Applications*, 7(2):163-174, April, 2002.

[11] A. Liotta, C. Ragusa, G. Pavlou, “Running Mobile Agent Code over Simulated Inter-networks: an Extra Gear Towards Distributed System Evaluation”, *Proceedings of the 2nd WSEAS International Conference on Simulation, Modeling and Optimization* (ICOSMO'2002), Skiathos Island, Greece, pp.404-409, September 2002.

[12] J. Liu. *Dartmouth S.S.F.* (2002). Available at http://www.cs.dartmouth.edu/research/DaSSF/intro.html (1 June 2004).

[13] J. Liu. *DaSSFNet—A High-Performance Network Simulator* (2002). Available at http://www.crhc.uiuc.edu/~jasonliu/projects/ssfnet/ (1 June 2004).

[14] J. Liu. *Introduction to SSFNet DML* (2002). Available at http://www.crhc.uiuc.edu/~jasonliu/projects/ssfnet/dmlintro/ssfnet-dml-intro.html (1 June 2004).

[15] SSF Research Network. *SSFNet DML Reference* (2002). Available at http://www.ssfnet.org/InternetDocs/ssfnetDMLReference.html (1 June 2004).

[16] SSF Research Network. *How to write DML network models* (2002). Available at http://www.ssfnet.org/InternetDocs/ssfnetTutorial-1.html (1 June 2004).

[17] M. Woodside. “Scalability metrics and analysis of mobile agent systems.” In *Infrastructure for Agents, Multi-Agents, and Scaleable Multi-Agent Systems*, Volume 1887 of LNCS, pages 234-245. Springer-Verlag, 2001.

Appendix A: Sample MA Simulation DML Configuration File

```
# c6-r-s.dml
#
# 6 clients, 1 router, 1 server.
# Agent itinerary: C(21,22,23,24,25,26)->S11
#                  S11->C(21,22,23,24,25,26)


Net [

  frequency 1000000000    # 1 nanosecond time resolution

  randomstream [
    generator "MersenneTwister"
    stream "seedstarter1"
    reproducibility_level "timeline"
  ]

  router [
    id 1
    graph [ProtocolSession [name ip use SSF.OS.IP]]
    interface [idrange [from 0 to 6] buffersize 16000 _extends .dictionary.100BaseT]
    route [dest default interface 0]
  ]

  host [
    idrange [from 21 to 26]
    interface [id 0 _extends .dictionary.100BaseT]
    route [dest default interface 0]
    _extends .dictionary.Client
  ]

  host [
    id 11
    interface [id 0 _extends .dictionary.100BaseT]
    route [dest default interface 0]
    _extends .dictionary.Server
  ]

  link [attach 1(0) attach 11(0) delay 0.001]  # Router 1 to Server 11 link.
  link [attach 1(1) attach 21(0) delay 0.001]  # Router 1 to Client 21 link.
  link [attach 1(2) attach 22(0) delay 0.001]  # Router 1 to Client 22 link.
  link [attach 1(3) attach 23(0) delay 0.001]  # Router 1 to Client 23 link.
  link [attach 1(4) attach 24(0) delay 0.001]  # Router 1 to Client 24 link.
  link [attach 1(5) attach 25(0) delay 0.001]  # Router 1 to Client 25 link.
  link [attach 1(6) attach 26(0) delay 0.001]  # Router 1 to Client 26 link.


  # This attribute allows us to assign a list of servers to each individual host,
  # specifying to whom it should send its agents (picked at random if more than one).
  traffic [

    pattern [
      client 21
      servers [nhi 11(0) port 1600]
    ]
    pattern [
      client 22
      servers [nhi 11(0) port 1600]
    ]
    pattern [
      client 23
      servers [nhi 11(0) port 1600]
    ]
    pattern [
      client 24
      servers [nhi 11(0) port 1600]
    ]
    pattern [
      client 25
      servers [nhi 11(0) port 1600]
    ]
    pattern [
      client 26
      servers [nhi 11(0) port 1600]
    ]

    pattern [
      client 11  # the server
      # There is no other server (any received agents will
      # be sent back to their source after processing).
    ]
  ]

] # end of Net


dictionary [

  100BaseT [
    bitrate 100000000
    latency 0.0001
  ]

  Client [
    graph [
      ProtocolSession [
        name client
        use SSF.OS.TCP.test.AgentsClient

        # Verbosity.
        _find .dictionary.appsession.DEBUG
        _find .dictionary.appsession.OUTPUT_INDIVIDUAL_REPORT

        # Send and receive ports, and ACK size (same for all clients and servers).
        _find .dictionary.appsession.RECEIVE_PORT
        _find .dictionary.appsession.SEND_PORT
        _find .dictionary.appsession.ACK_SIZE

        # Agent generation frequency and cap parameters.
        DELAY_LAMBDA 0.5  # DELAY_LAMBDA is 1/(desired mean delay in seconds)
        MAX_AGENTS   100

        # Agent simulation parameters.
        AGENT_SIZE       29803
        SERIALIZE_TIME   0.002500
        UNSERIALIZE_TIME 0.044000
      ]
     ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
     ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster _find .dictionary.tcpinit]
     ProtocolSession [name ip use SSF.OS.IP]
    ]
  ]

  Server [
    graph [
      ProtocolSession [
        name server
        use SSF.OS.TCP.test.AgentsServer

        # Verbosity.
        _find .dictionary.appsession.DEBUG
        _find .dictionary.appsession.OUTPUT_INDIVIDUAL_REPORT

        # Send and receive ports, and ACK size (same for all clients and servers).
        _find .dictionary.appsession.RECEIVE_PORT
        _find .dictionary.appsession.SEND_PORT
        _find .dictionary.appsession.ACK_SIZE

        # Cap on the number of servers an agent should visit.
        MAX_SERVERS 1

        # Agent simulation parameters.
        UNSERIALIZE_TIME  0.038200
        SERIALIZE_TIME    0.003200

        # Query execution and results parameters.
        QUERY_EXEC_TIME   0.132000
        QUERY_RESULT_SIZE 11972
      ]
     ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
     ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster _find .dictionary.tcpinit]
     ProtocolSession [name ip use SSF.OS.IP]
    ]
  ]

  # Shared TCP configuration.
  tcpinit [
    ISS 10000               # initial sequence number
    MSS 1000                # maximum segment size
    RcvWndSize  32          # receive buffer size
    SendWndSize 32          # maximum send window size
    SendBufferSize 128      # send buffer size
    MaxRexmitTimes 12       # maximum retransmission times before drop
    TCP_SLOW_INTERVAL 0.5   # granularity of TCP slow timer
    TCP_FAST_INTERVAL 0.2   # granularity of TCP fast(delay-ack) timer
    MSL 60.0                # maximum segment lifetime
    MaxIdleTime 600.0       # maximum idle time for drop a connection
    delayed_ack false       # delayed ack option
    fast_recovery true      # implement fast recovery algorithm
    show_report true        # print a summary connection report
  ]

  # Shared client/server parameters.
  appsession [
    DEBUG false                     # print detailed client/server diagnostics
    OUTPUT_INDIVIDUAL_REPORT false  # print individual agent reports

    RECEIVE_PORT 1600
    SEND_PORT 1601
    ACK_SIZE 4
  ]
]
```

Appendix B: Results Summary

Experiments			
1 Client, 1 Server	**Simulation**	**D'Agents**	
Generation Rate (agents/sec)	Agent Time (sec)	Agent Time (sec)	□
0.5	0.305183	0.241839	0.063344
1	0.325815	0.267223	0.058592
1.5	0.344741	0.280793	0.063948
2	0.370884	0.320288	0.050596
2.5	0.412087	0.337871	0.074216
3	0.442907	0.372026	0.070881
2 Clients, 1 Server			
Generation Rate (agents/sec)	Agent Time (sec)	Agent Time (sec)	□
0.4	0.309788	0.256108	0.053680
0.6	0.325118	0.272341	0.052777
0.8	0.346244	0.290876	0.055368
1	0.372190	0.314440	0.057750
1.2	0.389636	0.331078	0.058558
1.5	0.445724	0.391520	0.054204
4 Clients, 1 Server			
Generation Rate (agents/sec)	Agent Time (sec)	Agent Time (sec)	□
0.4	0.346080	0.288114	0.057966
0.6	0.391316	0.333767	0.057549
0.8	0.469356	0.413282	0.056074
1	0.557155	0.558361	-0.001206
1.2	0.758563	0.964808	-0.206245
1.4	1.569288		
6 Clients, 1 Server			
Generation Rate (agents/sec)	Agent Time (sec)	Agent Time (sec)	□
0.2	0.328167	0.268662	0.059505
0.4	0.393377	0.352773	0.040604
0.5	0.442860	0.408331	0.034529
0.6	0.503973	0.468254	0.035719
0.8	0.768206	0.904998	-0.136792
8 Clients, 1 Server			
Generation Rate (agents/sec)	Agent Time (sec)	Agent Time (sec)	□
0.1	0.308009	0.252035	0.055974
0.2	0.350311	0.285418	0.064893
0.3	0.390239	0.343428	0.046811
0.4	0.462686	0.417322	0.045364
0.5	0.557830	0.546398	0.011432
0.6	0.742258	0.970799	-0.228541
0.8	7.295608	11.907552	-4.611944
10 Clients, 1 Server			
Generation Rate (agents/sec)	Agent Time (sec)	Agent Time (sec)	□
0.1	0.319164	0.261035	0.058129

0.2	0.373287	0.310817	0.062470
0.3	0.443026	0.402261	0.040765
0.4	0.551099	0.593644	-0.042545
0.5	0.842917	1.507537	-0.664620
0.6	4.050119	10.192165	-6.142046

1 Client, 2 Servers

Generation Rate (agents/sec)	Agent Time (sec)	Agent Time (sec)	□	□ / Server
0.5	0.531861	0.449236	0.082625	0.0082625
1	0.570851	0.483709	0.087142	0.0087142
1.5	0.639904	0.547563	0.092341	0.0092341
2	0.707835	0.591642	0.116193	0.0116193
2.5	0.750844	0.632569	0.118275	0.0118275
3	0.858560	0.731644	0.126916	0.0126916

1 Client, 4 Servers

Generation Rate (agents/sec)	Agent Time (sec)	Agent Time (sec)	□	□ / Server
0.5	0.976537	0.842433	0.134104	0.0134104
1	1.095040	0.920933	0.174107	0.0174107
1.5	1.243388	1.049681	0.193707	0.0193707
2	1.374950	1.133136	0.241814	0.0241814
2.5	1.533766			
3	1.750053	1.443673	0.306380	0.030638

1 Client, 6 Servers

Generation Rate (agents/sec)	Agent Time (sec)	Agent Time (sec)	□	□ / Server
0.5	1.458659	1.248616	0.210043	0.0210043
1	1.631510	1.402502	0.229008	0.0229008
1.5	1.821449	1.538172	0.283277	0.0283277
2	2.078145	1.676694	0.401451	0.0401451
2.5	2.303826	1.901068	0.402758	0.0402758
3	2.524360	2.155687	0.368673	0.0368673

1 Client, 8 Servers

Generation Rate (agents/sec)	Agent Time (sec)	Agent Time (sec)	□	□ / Server
0.5	1.910696	1.644476	0.266220	0.026622
1	2.179204	1.822479	0.356725	0.0356725
1.5	2.462558	2.002469	0.460089	0.0460089
2	2.743328	2.250280	0.493048	0.0493048
2.5	3.129285	2.529987	0.599298	0.0599298
3	3.413358	2.837030	0.576328	0.0576328

1 Client, 10 Servers

Generation Rate (agents/sec)	Agent Time (sec)	Agent Time (sec)	□	□ / Server
0.5	2.362934	2.059154	0.303780	0.030378
1	2.672887	2.22162	0.451267	0.0451267
1.5	3.066415	2.61152	0.454895	0.0454895
2	3.449798	2.784701	0.665097	0.0665097
2.5	3.875105			
3	4.171138	3.84412	0.327018	0.0327018