# Scalability & Multi-Agent Systems

Ralph Deters
University of Saskatchewan
Department of Computer Science

deters@cs.usask.ca

## ABSTRACT
The multi-agent research community is currently faced with a paradox. While promoting the use of agents as the silver bullet for various software engineering problems, it faces difficulties in presenting successful deployments. Despite the countless multi-agent prototypes that have been developed, the number of actually deployed and in use MAS is at best very small [11]. And as long as multi-agent frameworks continue to encounter difficulties in scaling up, it seems unlikely that this will change.

This paper has two aims. First, it is an attempt to relate the scalability problem of multi-agent systems with that of executing large numbers of concurrent threads. Second, it presents a CORBA/Java middle-ware layer that enables transparent access to the resources of different physical machines. Using this layer is becomes possible to build multi-agent systems that require large numbers of concurrent threads and significant memory resources.

## Keywords
Distributed Multi-Agent System, Scalability, JESS.

## 1. Introduction
Scalability is an important issue in the successful deployment of modern software, since performance requirements can change over time. Therefore, it is often expected that the software can be scaled up or down to ensure the desired performance at minimal costs.

But unlike mainstream computer science, where scalability is a central design issue, the multi-agent research community is just beginning to realize its importance [7] – especially when trying to deploy systems.

While many different types of multi-agent scalability exist (e.g. number of messages, complexity of agents) increasing the number of agents is by far the most important one. When dealing with large numbers of agents it is important to distinguish between reactive and proactive agents. A reactive agent will only be executed if it receives a message. Consequently a reactive agent will only consume processor time if it computes a response to an incoming message. Therefore increasing the numbers of reactive agents is predominantly a memory (agent storage) problem. Large multi-agent systems consisting of reactive agents tend to focus on techniques to minimize the amount of agents in the memory. Idle agents are paged out (serialized to file) and agents that receive a message are paged in (de-serialized from file). Yamamoto & Tai [9] demonstrate in an impressive way that this approach can enable the hosting of several hundred thousand agents. Unfortunately, purely reactive agents are somewhat less interesting than proactive agents. Unlike their reactive counterparts, proactive agents don't need messages to start actions, they can initiate themselves complex tasks like the discovery of new services. As a result, a system consisting of proactive agents seems more lively and adaptive, making it a more attractive for researchers. But this agility of the proactive agents comes at the price of continuous execution. Each proactive agent is an object with one or more threads of control. Increasing the number of proactive agents leads to an increase in concurrent threads that sooner or later exceeds the possibilities of a single machine. The distribution of processor load is therefore a central issue in the development of multi-agent systems using proactive agents.

This paper starts with a discussion about scalability and multi-agent systems followed by a comparison of different approaches for achieving scalability. Then the concept of using atomic agent runtime environments as a means for ensuring scalable agent host is presented. This is followed by a presentation of the proposed agent-framework architecture and its empirical evolution. A summary and an outlook conclude the paper.

## 2. Scalability
Since scalability is the main topic of this paper a brief introduction into this topic is given below.

### 2.1 What is Scalability?
The term scalability is typically used in two different ways:

#### *2.1.1 Weak Scalability*
"It is the ability of a computer application or product (hardware or software) to continue to function well as it (or its context) is changed in size or volume in order to meet a user need. Typically, the rescaling is to a larger size or volume. The rescaling can be of the product itself (for example, a line of computer systems of different sizes in terms of storage, RAM, and so forth) or in the scalable object's movement to a new context (for example, a new operating system)." [3]

#### *2.1.2 Strong Scalability*
"It is the ability not only to function well in the rescaled situation, but also to actually take full advantage of it. For example, an application program would be scalable if it could be moved from a smaller to a larger operating system and take full advantage of the larger operating system in terms of performance (user

response time and so forth) and the larger number of users that could be handled." [3]

### *2.1.3 Scale-Up Scale-Down*

Scaling can be performed either up or down. A program is considered able to scale up, if the adding of certain resources (e.g. memory, processor) will lead to an increase in performance. While scaling up is the by far most often talked approach, it is important to realize that systems can also scale down. A program can be considered able to scale down if it is possible to remove/reduce the access of certain resources (e.g. less memory, slower processor). Especially with the rise of small and/or mobile computing devices the problem often arises if and how software can be scaled down to run in a more resource-constrained environment.

## 2.2 Types of Scalability

It seems useful to distinguish between systems that can dynamically adjust at runtime and those that require restarting or even recompiling.

Systems that can adjust at runtime have *total dynamic scalability*, ones that require restarting have only *partial dynamic scalability* and those that require recompiling have *static scalability*.

Clearly the systems that have the ability to adjust at runtime to changes are of highest interest since they provide the biggest flexibility.

## 2.3 Measuring Scalability: Metrics

Various metrics for scalability exist [12] and this paper will not attempt to add another one. Instead the scalability of the system will be viewed as the ratio between performance and resources. As the available resources increase, the performance should increase. This rather crude formula ignores many important aspects, like quality of service, but offers the advantage of simplicity.

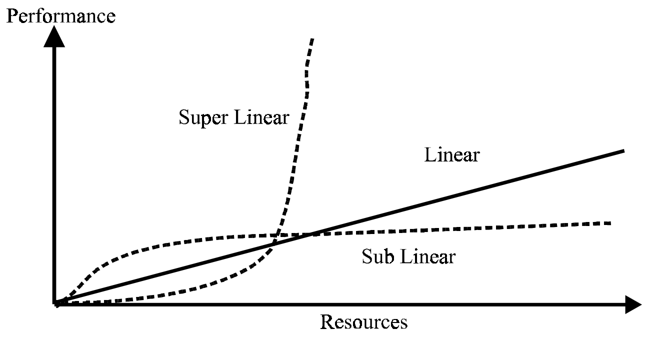

**Figure 5: Different scalability behavior**

## 3. Multi-Agent Systems & Scalability

In the multi-agent system community there is no commonly shared interpretation of scalability. While scalability can refer to many different aspects like agent complexity or message volume, this paper focuses only on the following three aspects.

**1) Number of Agents in the System/Host**

How many proactive agents can be hosted in the multi-agent system and/or any agent host? How many concurrent threads can the multi-agent system manage?

**2) Resource Consumption of Agents**

What is the upper limit of resources an individual agent can acquire in a multi-agent system? How large (memory consumption) can the agents become?

**3) Number of Messages**

Is it possible to increase the number of messages and still ensure a fast and reliable message routing? What is the limit for the message routing and can it be distributed?

## 3.1 Factors That Impact Scalability

Two main factors can be identified, that impact the scalability of a multi-agent system: load and complexity. Even a medium sized multi-agent system with only a few hundred agents is already a significant workload, which would require the use of several high end PC or a multi-processor WS.
Using just hundred simple (Java) agents of the FIPA-OS on a Celeron 533 MHz with 96 MB (Linux) is already challenging. The memory footprint adds up to 130 MB resulting in a close to 100 % usage of available CPU.
But it is important to note, that the scalability problem of multi-agent systems is not only a question of computational resources - it is also one of software complexity.
Fault management [1] [4] and consequently agent design patterns [2][6][8] are equally important problems that impact the maintainability. While solving the resource problems is crucial for building a larger multi-agent system it seems clear, that without fault management and design patterns these systems would not be maintainable/extendable.

## 3.2 Bottlenecks in Multi-Agent Systems

Agent hosting and the routing of messages are the two main bottlenecks in a multi-agent system consisting of proactive agents.

### *3.2.1 Agent Host*

The hosting of proactive agents requires significant memory and processor resources since each proactive agent represents an object with its own concurrent tread of execution. Consequently every increase of proactive agents results in an increase of concurrent threads.
Only if the number of concurrent threads can be distributed over different physical machines, it is possible to scale up the number of agents in a multi-agent system without risking a decreased performance of individual agents.

### *3.2.2 Message Routing*

Agents rely on message passing as their major form of communication. Fast and reliable message delivery is hence of the utmost importance in order to avoid potentially chaotic behavior. To avoid an overload of the message router, its load has to be also distributed over multiple processors.

## 3.3 Enhancing Scalability – A closer Look

Currently the main approaches for significantly enhancing scalability are replicating the framework, component distribution, component replication and agent scheduling. Since the replication of the framework (Hive [13], Agora [14]) tends to waste resources and doesn't seem to be a promising solution for scaling up it will be ignored in the rest of the paper.

### *3.3.1 Component Distribution*

An easy way to ensure that the combined resources of several physical machines can be used is to manually distribute the main components of the multi-agent system over several machines. By hosting the components in different processes an inter-process communication (IPC) is required. This can be done by using only simple socket communication, via language specific approaches like Java's RMI or by using language and platform independent approaches like CORBA.

While providing an easy way to distribute, this approach has two major drawbacks, the first of which is the manual distribution. It is up to a human programmer to decide where components have to reside, which significantly complicates adjustments to the often-varying load situation of machines. The second and more serious disadvantage concerns the linkage of the components to the resources of only one physical machine. This means, that it is impossible to scale individual components of the multi-agent system beyond the limits of a single machine. While this might not be a concern for a registry component, message routers and agent hosts can easily outgrow the capabilities of single machines. As a result, the system is limited in its scalability since individual components can not spread over more than one physical machine.

### *3.3.2 Component Replication*

Instead of only distributing the components over different physical machines, it is also possible to replicate them. If the message router is expected to become a bottleneck, it is possible to have two or more message routers, each hosted on a different machine. Assuming that some load balancing is in place, this solution helps to significantly boost the performance of the multi-agent system. Typically, the agent hosts and message services (e.g. router) are replicated since they are the most likely to be performance bottlenecks. Like component distribution, the problem of inter-process communication has to be solved by either implementing proprietary protocols via sockets or by using standardized ones like CORBA. It is important to note, that with duplication of components the amount of necessary inter-process communication is likely to increase. It is, therefore, important to design the system in a way that the load of IPC is distributed e.g. by using IIOP.

Component inflation is a very common strategy to ensure scalability within a multi-agent system. The replication of hosts and message routers is a common technique and by using mobile agents that can migrate between the agent hosts, a decentralized load balancing can be achieved. But it is important to realize, that the replication of components has two major disadvantages, resource consumption and increased complexity. By adding complete components like agent hosts, resources are wasted since all the host specific services are also replicated. In addition, the system becomes more complex, since more components have to be managed. Load balancing is now an additional problem!

While having multiple message routers might not be problematic, an inflation of hosts can quickly become a maintenance challenge, especially if large numbers of agents are floating between them. It is therefore no surprise, that there are literally no reports about successful deployed systems using this technique.

### *3.3.3 Agent Scheduling*

To increase the capacity of individual agent hosts, special agent/thread schedulers are often used. Agent scheduling tries to optimize the distribution of the sparse resources among a large numbers of agents. Agents that are not performing tasks are deactivated thus preserving resources for the others. Agent scheduling typically requires that agents be split into a large group of deactivated agents and a small group of active agents. The deactivated agents consume beside memory no other resources, while the active agents have access to all resources. Agent scheduling requires a "good" scheduling policy to determine which agents are to be moved from deactivated to active state and the ability to control the individual agent's access to system resources. Various scheduling policies are possible that use ranking of importance, heuristics and statistics.

The most common form of scheduling is the event or message oriented scheduling where an event or the arrival of messages priories agents. In such systems that typically consist of reactive agents, only agents that received messages are moved from the deactivated queue into the group of active agents (for the duration of the message processing). A variation of this approach can be seen in [9] where (reactive!) agents are moved upon events (user logs on/off) from the deactivated to the active group or vice versa. Agent scheduling is the only technique that has a proven track record of enabling the execution of large numbers of (reactive) agents. When using scheduling for proactive agents the CPU time slice for each agent is reciprocal to the number of agents, which renders this approach useless for large numbers of proactive agents. The fact that scheduling itself is also a computationally expensive operation might even worsen an already existing CPU shortage. Consequently this technique is not very useful for systems with predominately proactive agents.

## 4. The Need for Transparent Access

All three mentioned approaches enhance the scalability of a multi-agent system, but fall short of providing a possibility to enable components to scale beyond the limitations of their underlying physical machines. To ensure that all components of the multi-agent system can scale, it is important to ensure that the resources of several physical machines can be accessed by its components. Only if the components themselves, most notably the agent host, have *transparent* access to the distributed resources scalability can be achieved.

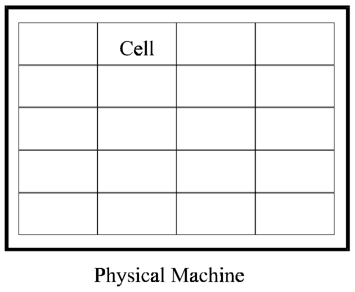

**Figure 6: Machine = Array of Cells**

Ensuring that this access is transparent helps in avoiding additional complexity within the components. Access to resources is transparent if it hides details like resource location from the requesting component ensuring a simple access and the flexibility to add or remove resource providers without the resource requestors knowing.

The resources most likely needed by a host are threads (processor) and objects (memory). Therefore the transparent resource management layer needs to be able to use/create threads and objects within other processes.
As shown in figure 6 a physical machine is viewed as a collection of atomic cells, each representing a container for one object to which one thread of control is associated. The cells are used as agent runtime environments. Adding an agent to the system is now a process of locating a cell and placing the agent into the cell. The cells are safe sandboxes for executing the agents. Since the agent in a cell is unaware of its physical location it is relatively easy to move the cells between processes and to achieve a basic load balancing.

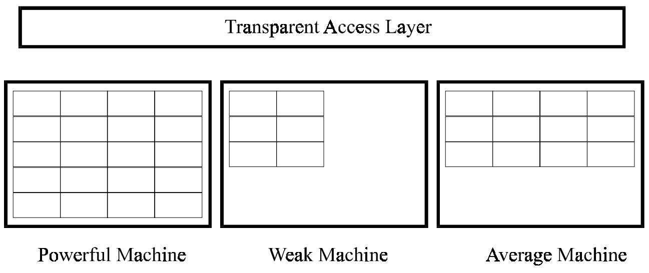

**Figure 7: Different Machines = Different Arrays of Cells**

Another advantage of using cells is that machines with different capabilities can be viewed as machines with more or less cells. This uniformity allows for the simplicity and efficiency in accessing the resources of different machines.
To be able to view a machine as a collection of cells it is necessary to have at least one process running on the machine. This so-called provider process will create the cells and offer them to the transparent access layer. Provider processes can create all cells in an initialization phase. But it is also possible to dynamically create and destroy cells to adjust to the requests from the transparent access layer.
The transparent access layer allows a host to farm out the execution of agents. Only by distributing the load it becomes possible to ensure that a large number of agents reside in a single agent host.
Besides being a means to access resources of different machines, cells are also a useful concept ensuring security and safety. The cells are designed to protect the agent from hostile events and to ensure that the agent can access directly system resources. Each cell consists of two containers, one for an agent and one for an environment object.
Since the agent is prohibited to directly access any system resources or services it has to use the environment object as a proxy. Only via the environment object the agent is able to send messages or use standard services like the registries. The environment object can therefore be best described as proxy that keeps the implementation of its the public methods hidden. In the current implementation, the environment object contains a hash-table that can be modified by the higher management functions to modify the agent's access or the agent's perception of the system. By forcing the agent to channel every interaction through the environment object two goals are achieved, *fine-grained control* and *location independence*. Fine-grained control enables the system to completely disconnect troublesome agents or to redistribute sparse resources to agents performing vital services.

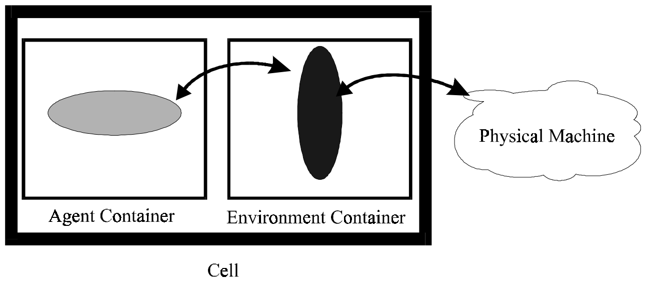

**Figure 8: The Cell – A secure runtime environment for agents**

The environment object also allows for the physical location independence of the agents. The agent interacts with the system only via the proxy and is kept unaware of location details. This allows the system to freely move agents between processes by moving their cells. In addition cells can be linked to any agent host by having the environment object returning references to the services of the same agent host.

## 5. Architecture of the Agent-Framework

The main components of our agent framework called DICE are providers, hosts and message routers. The providers create and host the cells that are used as safe agent execution environments. Consequently the providers are the processes that provide the required processor power and memory to execute the agents. Using several providers that reside on different physical machines allows to combine the resources of several physical machines and consequently to execute large numbers of agents. Figure 9 shows the architecture of the framework called DICE.

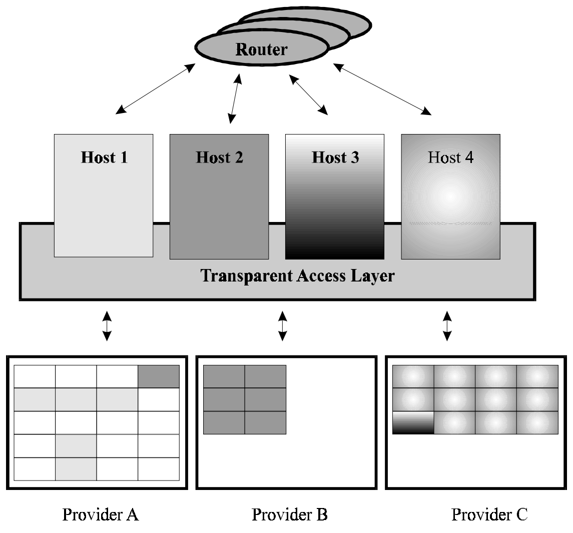

**Figure 9: Architecture of DICE**

Agent hosts are the logical containers of the agents and offer only basic registry services and messages. Whenever an agent is added to a host, the host will contact the providers and request a reference to a free cell. The agent is added into the cell (hosted by the provider process) and consequently consumes resources of the provider process. Message routers are used to enable the message

transportation between different agent hosts. Sending a message to an agent is performed in two steps, in which the router is only used to determine the agent host that hosts the recipient of the message. As soon as the router has determined the agent host it forwards the message to the agent host that will then deliver the message to the right agent. This two-step approach allows for a simple distribution of load between the router and the agent host, ensuring that the router is capable of handling a large message volume.
As shown in figure 9, there is a N:M relationship between hosts and providers. Several providers can serve a single host, and multiple hosts can share the same provider. Since all the location details of the cells are encapsulated in the transparent access layer it is also possible to add or remove hosts at runtime. Note that the transparent access layer is part of every host and not a separate process. Since the providers are passive and relatively simple, the hosts are required to discover and establish a connection with them.

## 5.1 Cells

A cell is an atomic agent runtime environment that resides within a provider process. Its main purpose is to ensure the safe and secure execution of an agent by prohibiting it to directly access system resources or services. Creating only as m any cells as the underlying physical machine can support ensures the required computational resources. Protection of an agent from hostile events is achieved by hiding the physical location of each agent from itself and consequently from other agents and by being able to completely withdraw resources from a hostile agent.

## 5.2 Environments

The environment object is the proxy though which an agent can interact with the system. Essentially the environment object is a collection of references to agent host specific services like messaging and registries. To logically link a cell to an agent host, it is necessary to inform the environment object about the new agent host. As soon as the environment object is informed about the new agent host, it will exchange its currently used references with those of the new host.

```
// Environment
interface Environment: DICE::Basic::CORBA::RootObject
{
void SendMessage(in DICE::Communication::CORBA::Message
NewMessage);

void             SetPolicy(in string Policy);
void             SetHost(in Host NewHost);
void             SetCell(in Cell NewCell);
YellowPages      GetYellowPages();
WhitePages               GetWhitePages();
ContractManagement       GetContractManagement();
};
```

**Code Snippet 1: IDL of am Environment**

## 5.3 Agents

Agents are the basic building blocks of any multi-agent system. The agents are supposed to offer a small number of methods of which the NextStep() method is by far the most important one. By calling an agents NextStep() method it is supposed to perform one complete compute cycle e.g. check for messages, respond etc. The agent is expected to return control back to the method calling thread as soon as possible. The use of this method enables the agent to perform one logical step, starting with a consistent state and ending in a consistent state.

## 5.4 Providers

As mentioned earlier the providers create and host the cells, which are the atomic agent execution environments. As shown in figure 10 the providers consist of a pool of threads and a two queues, one storing used cells, one unused cells.

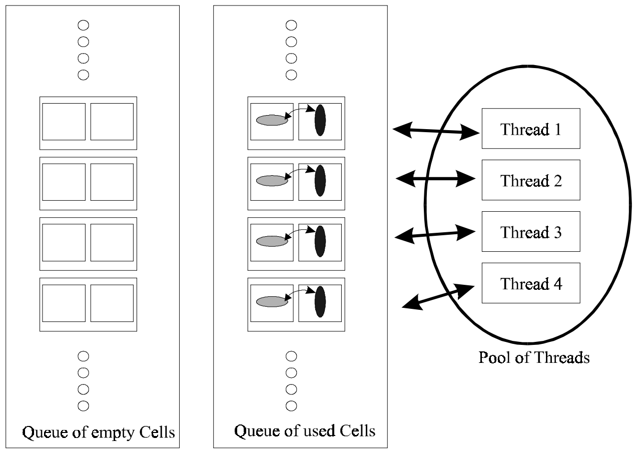

**Figure 10: The Provider**

Whenever a host requests a cell, the provider gets the next cell from the list of free cells, removes it from the free queue and adds it to the used queue. The host receives a reference to the cell, which can be used to add an agent.
The previously mentioned pool of threads monitors the queue of used cells. Each thread tries to de-queue a cell and to execute the contained agent by calling its NextStep() method. As mentioned earlier each agent is expected to compute its next step and then to return the control back to the thread.
Providers can vary by the amount of cells and the number of threads they use. As the amount of cells is increased the memory consumption is increased. As the number of threads rises the processor load increases. The amount of cells and the number of threads to be created in each provider are provided when it is started. Using various settings enabling the creation different provider types, each best suited for the underlying physical machine.

## 5.5 Agent Hosts

The agent host is nothing more than a lightweight process that offers elementary messaging and registry services and maintains references to the various cells it uses.
Since the host itself does not contribute any resources to the execution of the agents, it is not impacted by any quantitative or qualitative agent changes. As more agents are added to a host, the host has to request more cells from the provider processes. Only if all provider processes fail to offer any unused cells the host is unable to accept another agent. But since the provider processes can reside on different machines, it is easy to increase the number of provider processes and to distribute their load evenly over the available physical machines.

## 5.6 Message Routers

The message routers connect the different hosts and enable the message passing between agents of different hosts. The routers are relatively simple processes that use hash tables to route incoming messages to the corresponding agent host. The agent host performs message delivery to the individual agents.

## 6. Implementation

The agent framework is implemented in Java using Jdk1.3 and uses the standard Java CORBA tools of that version. Providers, Hosts, Routers are all designed as different processes that communicate via an object request broker, which can be different for every process. Using IIOP a simple cross ORB communication is enabled. The source code for DICE (including the code generated from the IDL) is about 560 KB. The current version of DICE is used in the I-Help[10] which consists of over 400 agents.

## 7. Evaluation

The empirical evaluation was performed the following questions:

- Can the multi-agent system handle 1000 or more agents?
- What is the average response time to incoming messages?
- Can provider processes be added or removed during runtime?

## 7.1 Configuration

The system is implemented in Java jdk1.3 and uses the jdk1.3 CORBA tools to provide IPC. The configuration used in the evaluation consisted of using 10 Sun Ultra 5 machines (128 MB, 64-bit ULTRASPARC IIi 333 MHz), one Sun Ultra II (1 GB, 4 x 64-bit ULTRASPARC II 400 MHz) and one NT Workstation (64 MB, P1, 200 MHz). The 10 Ultra 5 were hosting each a provider process, which contained 100 cells and used 20 native threads to executed them. The Sun Ultra II hosted the ORB, the agent host process and the router process.
Each provider process was configured to create 100 cells and use 20 threads to execute the contained agents in a round robin approach.

## 7.2 Test Conditions

The experiments were conducted over a period of low machine usage by other users. As a result the observed performance measurements are more conservative.

## 7.3 A few Experiments

Two types of experiments were conducted.
The first four experiments were aimed at investigating the behavior of the system under increasing agent loads. To measure the impact of larger numbers on hosts and agents, up-load time and average response time were measured.
The fifth and sixth experiment were used to determine the effects of adding and removing resources at runtime.

### *7.3.1 Adding thousand Agents*

To measure the impact of using a larger agent society on a host four different sets of agents were used. Each experiment uploaded 1000 agents and measured upload time and average response time to a message. Uploading refers to the creation of the agent in a launcher process, and the sending of the serialized agent to the host. The host de-serializes the agent and allocates a cell in which the agent will be placed. As the last step, the host registers the agent in its registry (white pages).

The first experiment (empty-agent-load) involved 1000 *no load* agents that did nothing except checking for incoming messages. As soon as a message was detected , it was returned to the sender. These agents represented the simplest type of agents that are possible using the DICE framework and are useful as a reference point for the other experiments.
The second experiment (*mix-load*) was aimed at investigating the impact of more CPU costly agents. To simulate a vivid agent society with varying loads, every agent was now expected to first calculate a randomly selected Fibonacci number between 0 – 30 and then to check for incoming messages. The response to an incoming message was to send it back to the sender. The second experiment was designed to represent a system consisting of simple agents that have varying loads over time.
In the third experiment a constant *heavy-load* for the system was simulated. The agents had to calculate the Fibonacci of 27 prior to responding to messages. While in principle any Fibonacci number could have been selected, 27 seemed a good choice given the available hardware.

Below are the graphs for the uploading time shown.

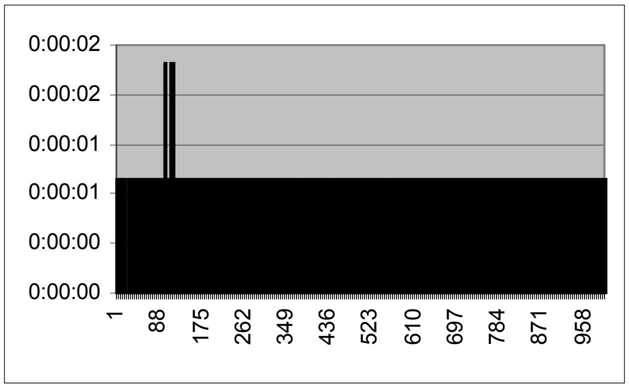

**Figure 11: 1000 “No-Load” agents**

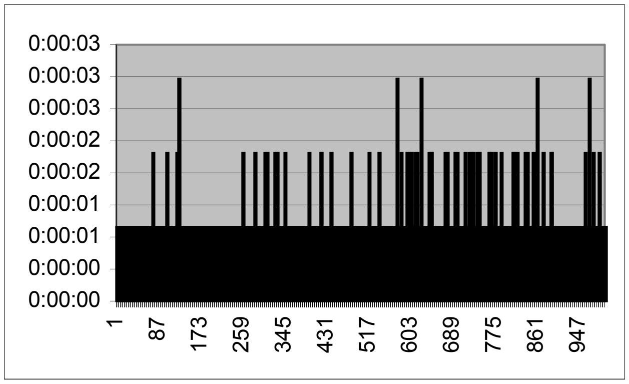

**Figure 12: 1000 “Mixed-Load” Agents**

Figure 11 shows the result for uploading 1000 empty agents to a host. Besides a few unexplainable spikes the time consumption is constant. This graph indicates that the upload time is more or less constant. Each provider process needed about 27 % of its CPU for hosting the agents.

The results of figure 12 show that with agents that create a load (calculating a random Fibonacci sequence) more spikes appear. The average agent needed about 1.5 seconds to calculate the sequence making it a moderate load. Each provider process needed about 87 % of its CPU for hosting the agents.

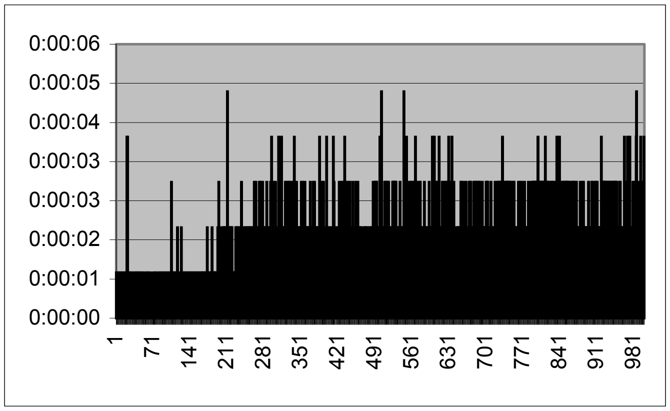

**Figure 13: 1000 "Heavy-Load" agents**

Figure 13 shows the result of uploading the heavy-load agents. Each agent calculated the Fibonacci sequence for 27 which took an individual agent in average more than 2 seconds. Each provider process needed about 96 % of its CPU for hosting the agents.

### *7.3.2 Sending 100 Messages to 1000 Agents*

After the uploading was completed, 10 processes on different hosts were used to send 100 messages to each agent. The agents were randomly selected and received a message a time. Each agent provided a timestamp in the return message that allowed for calculating message response averages. Please note that only the time between sending the message and having the agent creating a response is measured.

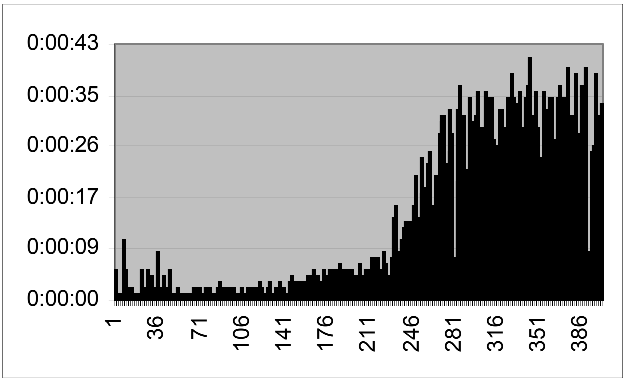

**Figure 14: 400 Jess Agents**

<u>**The results were as follows:**</u>

| | |
|---|---|
| Empty agent experiment: | average of 1 second |
| Mixed-Load agent experiment: | average of 2.5 seconds |
| Heavy-Load agent experiment: | average of 3 seconds |

### *7.3.3 Using Jess in an Agent*

After the promising results of uploading agents that were programmed using only Java, an experiment was conducted in measuring the impact of loading 400 agents that each contained the Jess rule interpreter. As clearly seen in the graph the load of only 400 Jess agents is already significant. As soon as they are up-loaded they begin to consume processor and memory space at a stunning rate. The average memory footprint for a provider hosting 100 jess agents is about 100 MB. Since only simple load-balancing was used, the more powerful machines are "filled up" first. The upload time is already initially significantly larger than for an average Java agent. A Java agent is anywhere between 400 and 800 bytes. The jess agent with only a few rules uses already around 10 K (4.5 KB for the serialized Rete engine alone). Since fewer agents were used the performance for responding to a message was keep to 2.5 seconds.

### *7.3.4 Adding & Removing Providers*

The adding of providers has proven to be simple. Using a command-line Java program new providers can be added at any given moment in time. In the current phase of development the new providers are only used when agents are uploaded or created. An automatic load-balancing is currently not in use.

Removing a provider has to be announced to avoid loss of data. If the provider announces to the hosts that it will go offline, the hosts try to allocate from the other providers additional cells to use as replacements. The time for moving from one provider to another is equivalent to the already presented up-load times. If a provider goes unexpectedly offline or attempts to allocate new cells from other providers fail, the agents hosted on the provider are lost.

## 8. Discussion

The results for the first three experiments were more or less predictable. As the load of the individual agents increases, so does the load for the provider processes.

That the adding of providers at runtime is without problems and that the removal of providers hosting agents is problematic in case that no notice is given or no alternative cells can be found, is more or less expectable. The interesting feature is that the number of physical machines can easily be scaled up. Other experiments used 32 different machines, 15 Sun workstations, and 17 NT boxes with 256 MB and Atholon 800 MHz. processors.

While it was expected that agents using Jess rules to define their behavior were to consume more resources than purely Java coded ones, the number is still surprising. According to the measured results, machines with powerful processors and large memory are needed to run even average size agent communities. In addition, the sudden rise to an upload time of 35 seconds is not very encouraging form a developers perspective. An explanation for this surprisingly high time is the time consuming de-serialization of the Jess rule engine. Since it is only partially serializable the facts have to be saved prior to serialization in a string array and later re-entered into the de-serialized Rete object. As a result the moving of Jess rule engines is slow. Being able to still keep the message response time around 2.5 seconds is a very good achievement since this indicates that given enough "resources" Jess agents can be used in a real world application.

## 9. Current Research

We believe that the design of more interesting multi-agent systems requires the use languages that offer a higher level of abstraction than Java. The JESS toolkit and its many extensions like Fuzzy-Jess seem a far more promising avenue. Unfortunately the use of such abstraction tools requires significant computational resources.

In the beginning we focused mainly on improving the distribution of complete agents over multiple machines. The current DICE package allows this in an easy way. But by using the JESS package and adding more and more rules to the agents we realize that we have to address the scalability of individual agents. How to design an agent that it can scale? Our current Jess agents consist of 5 different sets of rules (and 5 different instances of the rule interpreter) that are executed sequentially. One way to improve the performance is to distribute the load of executing an agent over multiple cells. Using more than one cell to execute an agent has many interesting advantages (e.g. replication, and performance improvements). But the use of multiple cells also introduces new problems - how to ensure that the different cells are executed more or less synchronized and how to ensure that they are not moved to different providers.

Another area, which has caught our attention, is to increase the fault tolerance of the system by replicating agents. By creating multiple replicates of agents and distributing them over different provider processes it becomes possible to increase the impact of losing a physical machine and to improve the overall system performance since costly recovery [5] can be avoided. Replicating the agent leads again to interesting problems like state-synchronization of the replicates, avoiding conflicting behavior of replicates.

## 10. Conclusion

Scalability is a central problem in the deployment of multi-agent systems. Among the many forms of scalability the increase in numbers of agents is the most important one.

In this paper a multi-agent framework called DICE was presented that uses atomic agent runtime environments called cells as a means to distribute the load. Hosting the cells on different physical machines and ensuring a transparent access layer enables an agent-host to farm out the execution of agents to different machines thus allowing for easy up-scaling. The evaluation results show that this approach is well suited for handling large numbers of agents by automatically distributing the load.

## 11. Sources

All sources are freely available upon request.

## 12. REFERENCES

[1] M. Klein: Exception Handling in Agent Systems, in Proceedings of Agents 99 (Seattle 1999), 62-69.

[2] Wooldrige M.: Agent-Based Software Engineering, IEEE Transactions on Software Engineering, 144 (1), 26-37,1997.

[3] What Is Scalability? Web page: http://iroi.seu.edu.cn/books/whatis

[4] Kumar S., Cohen P.: Towards a Fault-Tolerant Multi-Agent System Architecture, Proceedings Autonomous Agents 2000, Barcelona, 459 –466, 2000.

[5] Toyama K., Hager G.: If at First You Don't Succeed, Proceedings of the 14th National Conference on Artificial Intelligence, Rhode Island, 3-9, 1997.

[6] Wooldrige M.: Agent-Based Software Engineering, IEEE Transactions on Software Engineering, 144 (1), 26-37,1997.

[7] Rana O., Stout K.: What is Scalability in Multi-Agent Systems, Proceedings Autonomous Agents 2000, Barcelona, 56 –63, 2000.

[8] Jennings N., et al: Developing Industrial Multi-Agent Systems, Proceedings of the First International Conference on Multi-Agent Systems ICMAS-95, San Francisco, 423-430, 1995.

[9] Yamamoto G., Tai H.: Architecture of an Agent Sever Capable of Hosting Tens of Thousands of Agents, Proceedings Autonomous Agents 2000, Barcelona, 70-71, 2000.

[10] Vassileva J., Greer J., McCalla G., Deters R., Zapata D., Mudgal C., Grant S.: A Multi-Agent Approach to the Design of Peer-Help Environments, Proceedings of AIED'99, Le Mans, 1999, 38-45.

[11] Gasser L.: MAS Infrastructure Definitions, Needs And Prospects, in Proceedings of Workshop "Infrastructure for Scalable Multi-Agent Systems" at Agents 2000, Barcelona, 7 pages.

[12] Woodside M.: Scalability Metrics and Analysis of Mobile Agent Systems, in Proceedings of Workshop "Infrastructure for Scalable Multi-Agent Systems" at Agents 2000, Barcelona, 5 pages.

[13] Maes P. et al: Hive: Distributed Agents for Networking Things, in IEEE Concurency, April-June 2000, 24-33.

[14] Matskin M, et al: Agora: An Infrastructure for cooperative work support in multi-agent systems, in Proceedings of Workshop "Infrastructure for Scalable Multi-Agent Systems" at Agents 2000, Barcelona, 6 pages.