A Framework for Observing Dynamics of Agent-Based Computations

Jarosław Kawecki and Maciej Smołka

Institute of Computer Science, Jagiellonian University, Kraków, Poland
smolka@ii.uj.edu.pl
http://www.ii.uj.edu.pl/~smolka/

Abstract. The paper contains a description of a framework designed for observing dynamics of mobile-agent computational applications. Such observations are thought to provide a basis for the experimental verification of an existing stochastic model of agent-oriented distributed computations. Some test results are also provided, which show that the proposed observational environment does not disturb an observed system's dynamics.

1 Introduction

Multi-agent systems (MAS) are considered one of significant paradigms for distributed system design in the industry (cf. [1]). In the science they are used to solve some complex problems such as evolutionary global optimization (cf. [2], [3]). However it is still not very common to apply multi-agent paradigm in the implementation of large-scale distributed computational systems, even if the idea of self-organizing computational application being a collection of mobile tasks which migrate over a network according to a diffusion-based policy in order to find the best environment for computations is known for several years ([4]). Thus it was quite straightforward to merge the two ideas, i.e. to enclose a computational task together with its data in a mobile agent box, giving the agent the abilities to migrate, to communicate with other agents, to split itself and to sense its environment properties, and finally providing the agent with some logic to decide which abilities to use in order to perform its task. Such a computational multi-agent system has been constructed ([5], [6]) on the basis of Java/CORBA platform. Sec. 2 describes its main features.

During the development of the system many theoretical questions has been raised. As an answer a formal model of multi-agent computations has been proposed ([7], [8]). It describes a multi-agent computational application as a controlled Markov chain. The model provides us with the definition of optimal scheduling and some results on the existence and characterization of optimal scheduling strategies. The model is briefly described in Sec. 3.

However the model itself needs experimental validation. The first step towards this goal is the construction of a framework for monitoring the dynamics of quantities appearing in the model. The present paper describes such a framework. It is designed as an extensible additional module for the above-mentioned

computational MAS platform. Design and implementation issues are covered in Sec. 4, some test results showing good features of the monitor are contained in Sec. 5.

2 Computational MAS

As the above-mentioned computational MAS is the basis for our new-projected framework let us first briefly describe its main architectural principles. For a more complete description and some implementation details we refer the reader to [5] and [6].

The architecture of the system is composed of *a computational environment* (MAS platform) and *a computing application* being a collection of mobile agents called *Smart Solid Agents* (SSA). The computational environment is the triple $(\mathbf{N}, B_H, perf)$, where:

$\mathbf{N} = \{P_1, \ldots, P_N\}$, where P_i is a MAS server called *a Virtual Computational Node* (VCN). Each VCN can maintain a number of agents.

B_H is the connection topology $B_H = \{\mathcal{N}_1, \ldots, \mathcal{N}_N\}, \mathcal{N}_i \subset \mathbf{N}$ is a direct neighbourhood of P_i (including P_i as well).

$perf = \{perf_1, \ldots, perf_N\}, perf_i : \mathbb{R}_+ \to \mathbb{R}_+$ is a family of functions where $perf_i$ describes the relative performance of VCN P_i with respect to the total memory request M^i_{total} of all agents allocated at the node.

The MAS platform is responsible for maintaining the basic functionalities of the computing agents. Namely it delivers the information about the local load concentration, performs agent destruction, hibernation, partitioning and migration between neighbouring VCN's and finally supports the transparent communication among agents.

We shall denote an SSA by A_i where index i stands for an unambiguous agent identifier. Each A_i contains its computational task and all data necessary for its computations. Every agent is also equipped with a shell which contains the agent logic. At any time A_i is able to denominate the pair (E_i, M_i) where E_i is the estimated remaining computation time measured in units common for all agents of an application and M_i is the agent's RAM requirement in bytes. An agent may undertake autonomously one of the following actions: *continue* executing its internal task, *migrate* to a neighbouring VCN or decide to be *partitioned*, which results in creating two child agents.

3 Formal Model Overview

In this section we introduce some key concepts appearing in our formal model of computing multi-agent systems. The detailed description of the model can be found in [7] and [8].

The main idea behind the model is the following. Instead of considering a single agent's behaviour we observe the time evolution of some global quantities characterizing the state of the whole computational application. The set

of state variables may typically include: the total number of allocated agents with respect to a VCN, the total remaining time of computations with respect to a VCN, the total memory requirement of agents with respect to a VCN. In addition some control variables have been introduced in the model. Namely we control the number of agents migrating between nodes, the number of agents splitting themselves and optionally the number of agents being hibernated or dehibernated. Finally equations of system evolution have been formulated describing a computational multi-agent system as a controlled Markov chain. It allows us to answer many fundamental questions (such as the question about the existence of optimal control strategies) by means of the stochastic control theory machinery.

Such a model needs an experimental verification. We are going to observe the dynamics of existing and new computational applications in order to check if the model describes them properly. But to this end we need a monitoring infrastructure which would register the time evolution of interesting application parameters and which, on the other hand, would not disturb the system behaviour significantly.

4 Monitoring Subsystem Architecture

4.1 Overview

The subsystem monitoring the agent dynamics is designed as a wrapper for a single MAS server and it focuses only on tracing the state of its host. Different wrappers do not communicate with each other hence they do not generate any network traffic. However such an approach needs an external synchronization mechanism to gather data with global time stamps for further processing and analyzing.

The monitor uses the SNTP protocol to set global time among servers. Each of monitors carries its own time service. Measurements taken during a monitor's work are stamped with time provided by this service. To stay accurate the time service periodically sends synchronization requests to the NTP server. The result is stored as a time offset and it is used to calculate the global time.

Calculations on the MAS platform are performed by agents. Significant aspects of the agent life-cycle activities (such as agent creation, migration or destruction) are traced using an event-driven approach. Namely every time an agent is born, leaves a server, enters another server or dies a trigger can be fired. It is marked with the current time stamp and with the agent identifier, so the whole agent traffic can be reconstructed after the experiment. The event-driven approach is reliable because it takes into account every event that occurs at the monitored server. However it is not possible to implement triggers totally outside the platform environment. The event infrastructure needs code integration with the platform server and the agent.

Some quantities should not be traced using the event-driven approach due to performance issues. Triggers are not efficient when a quantity value changes

really often. On the other hand some quantities can change in an unpredictable way, so there is no place where the trigger could be hooked. A snapshot mechanism is meant to support triggers in such situations. It is a kind of task that periodically takes a snapshot of a server's parameters. Such an approach avoids drawbacks of the event-driven approach and allows to gather more data for further analysis. Without it the server memory consumption or the processor utilization could not be traced.

4.2 Implementation issues

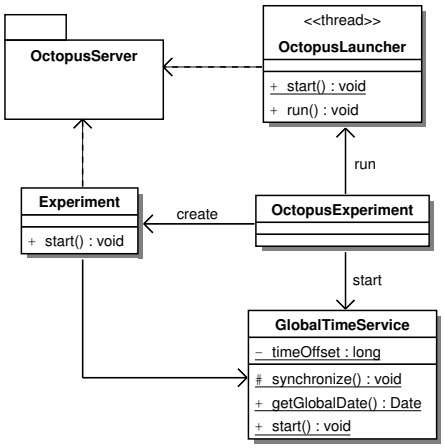

Fig. 1. Main components of a server monitor (UML class diagram)

A single monitor server consists of three main parts that are sequentially initialized (see Fig. 1). The first to launch is the monitor's observation subject, i.e. a MAS server (cf. [5]). The server is running on a dedicated thread. The time service is started next. It is used for periodical sending of SNTP requests to the time server in order to keep the most up-to-date offset to the global time. Third essential component called the Experiment is started as the last. It initializes triggers, snapshots and file output where measurement data are stored. The global time provided by the time service is used to stamp measurements. Experiment class is closely coupled to the MAS server and this dependency is bidirectional. Using snapshots the Experiment observes the state of the server, while the server fires triggers which are stored in the Experiment.

The time service uses the SNTP client to get the offset value from the time server. This is performed periodically. To this end the service spawns its own thread which sleeps during the most of its lifetime. It is only awaken once for a defined period and then it tells the service to update the time offset. Each time

the update is done the service invokes the synchronization trigger. It allows us to store the retrieved time offset in an output file.

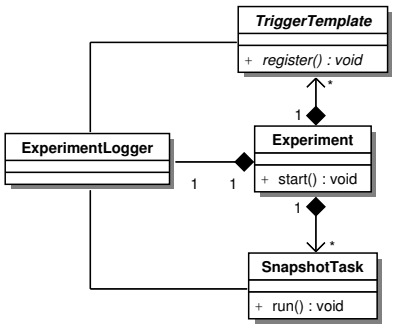

Fig. 2. Experiment (UML class diagram)

The main responsibility of the Experiment is to create all configured snapshots, triggers and instantiate the experiment logger (see Fig. 2). When all these items are created the experiment may be started. Then the experiment logger is initialized, all snapshots are scheduled and finally all triggers are registered. It means that since then triggers are able to receive calls from the MAS code. It is worth noticing that scheduling snapshots does not mean an immediate execution. The idea of the snapshot is to catch the consistent state of the whole platform. To this end two conditions must be met: all snapshots start at the same time and they are repeated with the same frequency. Both of these parameters are passed by the configuration file. The start time is defined as the globally defined moment common for all monitors. Before that moment no snapshots are taken, therefore no experiment activity should be performed. Only after the start time the platform is capable to work and all parameters are properly monitored.

A snapshot itself (see Fig. 3) does not observe a server state directly. It only provides environment for execution of measurements such as memory or processor utilization. The static part of the snapshot allows to assign some measurements. Then the dynamic part runs the snapshot. As s consequence all the assigned measurements are taken and sent to the experiment logger. As mentioned before the snapshot runs periodically. The facility that provides this behavior belongs to the Java standard library. The Timer object can schedule running objects whose classes extend the TimerTask abstract class. There is a major advantage of using the Java Timer, namely it is able to correct delays in the task execution. As a result the snapshots do not accumulate time shift.

A measurement (Fig. 4) is a piece of code which is responsible for measuring one parameter of the server. It takes the measure, stores the result and makes the value of the measure available for the experiment logger. The measurement is always considered in the context of a snapshot. Currently two kinds of mea-

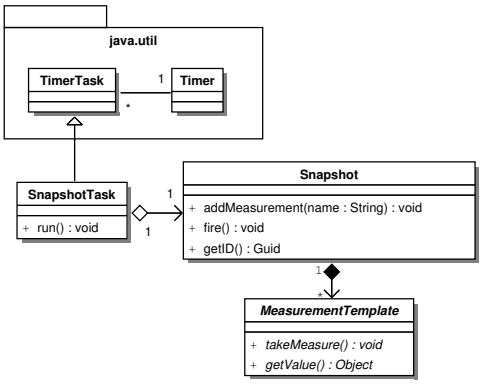

Fig. 3. Snapshots (UML class diagram)

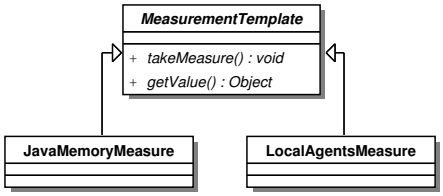

Fig. 4. Measurements (UML class diagram)

surements are implemented. The memory measurement monitors free memory. The agents measurement checks how many agents are currently located on a MAS server.

A trigger (Fig. 5) from a point of view is similar to a measurement described before. It is also connected with a single parameter of the MAS server. Despite the similarity the trigger can work without any snapshot. Not being a part of a snapshot has a downside effect. The code of trigger needs to be executed directly from the MAS platform code. Two MAS platform classes had to be modified in order to launch triggers related to agents: SerializedObjectStreamService and TaskBase (cf. [5]). Code changes were kept as small as possible. Currently five types of triggers are implemented. Serialization and deserialization triggers are responsible for tracing migration and hibernation of agents. Next two trigger types are designed for monitoring agent creation and destruction, hence in particular they can be used to observe partitioning. The last trigger type is fired when a synchronization with the time server occurs.

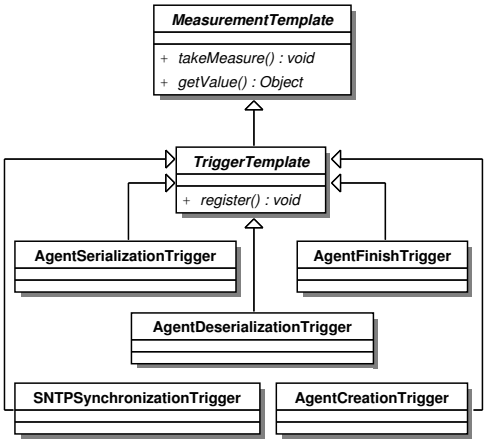

Fig. 5. Triggers (UML class diagram)

5 Monitor performance tests

The MAS platform monitor was designed to impose the lowest overhead possible. Of course since the monitor integrates with the platform code, it impacts utilization of such resources as processor, memory and network. Hence in order to determine the scale of the actual overhead some tests have been performed. They compared the total execution time for the *Subdomain-by-Subdomain* (SBS) distributed linear solver (cf. [5]).

The platform consisted of 12 PCs organized in a star topology and equipped with the Fedora operating system. There was also one PC outside the platform dedicated for the administration purposes such as: launching MAS servers in a specified order, initializing and deploying SBS computational agents and obtaining the test results. The monitor configuration consisted of:

- SNTP synchronization run every 10 seconds using `vega.cbk.poznan.pl` NTP stratum 1 server;
- all 5 triggers enabled (i.e. synchronization, agent serialization, deserialization, creation and destruction);
- first snapshot run every second tracing current number of agents located on a server;
- second snapshot run every 2 seconds logging free server memory.

All agents were deployed on the central server (s213-03) and then they were allowed to migrate according to a diffusion-based policy (cf. [6]).

The tests were organized into 16 groups. They varied by the number of agents involved in the computations and the size of the matrix computed by one agent. The tests took into account the total computation time. Each of the test groups was run 8 times for the unobserved MAS platform and other 8 times for the platform observed by the monitor.

Size of Matrix	Number of Agents: 32	64	128	256
72	54875 ms	106125 ms	194500 ms	362250 ms
144	66250 ms	105375 ms	206250 ms	391750 ms
288	67875 ms	118125 ms	232625 ms	448500 ms
576	108375 ms	191500 ms	348750 ms	677857 ms

Size of Matrix	Number of Agents: 32	64	128	256
72	54375 ms	103375 ms	196875 ms	353125 ms
144	62125 ms	105125 ms	203125 ms	395375 ms
288	66375 ms	117375 ms	231625 ms	429625 ms
576	106500 ms	187500 ms	348500 ms	678000 ms

Fig. 6. Average execution time without (on the left) and with (on the right) the monitor

Figs. 6 and 7 show that computations on the platform with the monitor are not significantly slower (at least in average) than those on the platform without the monitor. Thus the test results allow us to treat observations performed by the monitor as quite reliable, which means that the monitor does not disturb the agent dynamics.

Size of Matrix	Number of Agents: 32	64	128	256
72	3257 ms	8462 ms	11576 ms	18012 ms
144	27954 ms	7761 ms	9189 ms	20553 ms
288	6030 ms	5372 ms	4998 ms	29270 ms
576	3672 ms	9539 ms	7462 ms	21557 ms

Size of Matrix	Number of Agents: 32	64	128	256
72	2288 ms	7936 ms	12752 ms	2368 ms
144	10959 ms	9545 ms	10105 ms	28026 ms
288	2118 ms	4973 ms	10086 ms	15588 ms
576	2449 ms	4243 ms	7665 ms	15344 ms

Fig. 7. Execution time standard deviation without and with the monitor

6 Sample observation of formal model quantities

Next let us present some capabilities of the monitor in the area of observing quantities appearing in the previously-described formal model. The data presented in this section were obtained using the same topology as for the verification purposes. 256 agents were created at the external server and deployed onto the platform. Each of them resolved a linear equation system with the matrix of size 72. The data is presented with the precision of 500 milliseconds.

Fig. 8 shows the number of agents allocated on platform servers. The data were gathered using snapshots. Another view of the same quantity is given by Fig. 9 which shows serialization and deserialization activity on the servers registered by triggers. It turned out that SBS was quite predictable when it comes to the migration of agents. The first phase lasted about 90 seconds. Agents were deployed to the s213-03 server which was the centre of the star connection topology. There was noticeable increase of agent number at this server till 50th second. Then the immigration to the s213-03 stopped and agents were only migrating to leaves of the topology. After allocation agents to servers the migration process ended and all agents were busy with computations. The computation phase

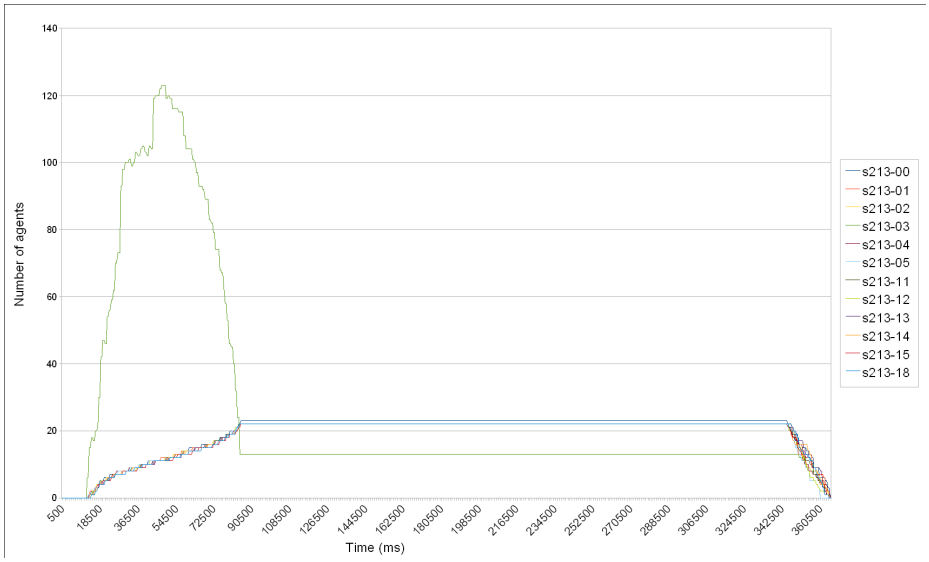

Fig. 8. Number of agents on platform servers

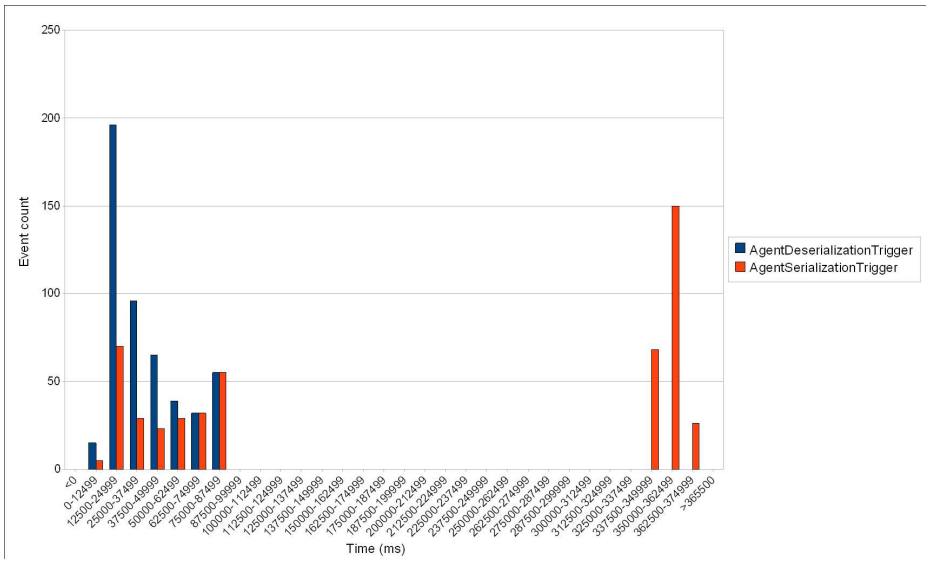

Fig. 9. Serialization and deserialization events

lasted approximately the same amount of time for all agents. At the end every agent returned the computation results to its parent and left the platform.

7 Conclusions and further research

An important goal for the designed monitoring subsystem was to avoid any disturbance in observed system dynamics. As test results suggest, the goal has been achieved. Moreover the constructed framework introduced only very small changes in the MAS platform code. The resulting framework is already quite useful because it can measure all basic formal model quantities, yet it is extensible enough to forecast some future needs such as new kinds of triggers or snapshot measurements possibly introduced in more sophisticated versions of the model. A natural consequence of the monitor construction shall be gathering observation data from computational applications other than SBS and validating the model through the analysis of the data.

References

1. Wooldridge, M.: An Introduction to Multi-agent Systems. Wiley (2002)
2. Cetnarowicz, K., Kisiel-Dorohinicki, M., Nawarecki, E.: The application of evolution process in multi-agent world (MAW) to the prediction system. In Tokoro, M., ed.: Proceedings of 2nd International Conference on Multi-Agent Systems (ICMAS'96), Osaka, Japan, AAAI Press (1996)
3. Byrski, A., Kisiel-Dorohinicki, M.: Agent-based evolutionary and immunological optimization. Lecture Notes in Computer Science **4488** (2007) 928–935
4. Luque, E., Ripoll, A., Cortés, A., Margalef, T.: A distributed diffusion method for dynamic load balancing on parallel computers. In: Proceedings of EUROMICRO Workshop on Parallel and Distributed Processing, San Remo, Italy, IEEE Computer Society Press (1995) 43–50
5. Uhruski, P., Grochowski, M., Schaefer, R.: Multi-agent computing system in a heterogeneous network. In: Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC 2002), Warsaw, Poland, IEEE Computer Society Press (2002) 233–238
6. Grochowski, M., Schaefer, R., Uhruski, P.: Diffusion based scheduling in the agent-oriented computing systems. Lecture Notes in Computer Science **3019** (2004) 97–104
7. Smołka, M.: A formal model of multi-agent computations. Lecture Notes in Computer Science **4967** (2008) 351–360
8. Smołka, M.: Task hibernation in a formal model of agent-oriented computing systems. Lecture Notes in Computer Science **5103** (2008) 535–544