AGENTS OVER THE GRID: AN EXPERIENCE USING THE GLOBUS TOOLKIT 4

Franco Cicirelli, Angelo Furfaro, Libero Nigro, Francesco Pupo
Laboratorio di Ingegneria del Software
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria
87036 Rende (CS) – Italy
Email: {f.cicirelli,a.furfaro}@deis.unical.it, {l.nigro,f.pupo}@unical.it

KEYWORDS
M&S, complex systems, agent-based computing, grid computing, actors, Globus, Java.

ABSTRACT

This paper describes an experience of porting the THEATRE agent architecture on top of the grid. The agent architecture consists of light-weight actors and computational theatres which have been proven to be well suited for modeling and simulation of complex systems. THEATRE nodes act as agencies that provide common services of message scheduling and dispatching to mobile actors. THEATRE is currently implemented in Java and can work with different transport layers and middleware. In the last years it was successfully interfaced to HLA/RTI, Terracotta, Java Sockets and Java RMI. The work described in this paper aims at experimenting with THEATRE over the grid, using in particular the Globus toolkit. The goal is to open THEATRE to the exploitation of virtual organizations of computing resources with secure communications, and to favor simulation interoperability through grid services. The paper summarizes THEATRE, describes a design and prototype implementation of THEATRE on top of the Globus Toolkit 4 (GT4), and demonstrates its practical use by means of a modeling example.

INTRODUCTION

Grid computing (Foster *et al*., 2001) enables hardware/software resources belonging to distinct organizations/institutions to be shared globally along with a concept of a Virtual Organization (VO) with an associated secure communication model. A grid infrastructure is founded on the service-oriented paradigm, where grid services are deployed, advertised, discovered and ultimately exploited by typically large distributed applications. A grid service is a (stateless) web service plus a (stateful) resource. A well-known software toolkit supporting grid computing is Globus (Globus, on-line)(Sotomayor & Childers, 2006).

In the last years, the research theme emerged of integrating agents with computational grids, which has been tackled by different researchers with different goals. In (Fukuda & Smith, 2006) the mobile agent system UWAgents is used as a technology starting point for building a grid infrastructure, e.g. enabling search of computing resources by agent navigational autonomy, exploiting migrating agent status for remote job submission and result collection etc. In (Moreau, 2002)(Avila-Rosas *et al*., 2002) an integration of the SoFAR agent system with web services (WS) is developed, where agents are created, deployed and published as WSs, with the goal of opening grid computing to agent-based applications.

In this work an original approach is proposed which uses a Globus grid as a middleware for supporting the Java-based THEATRE agent infrastructure (Cicirelli *et al*., 2009). The realization purposely embeds mobile agents in the grid and delivers an effective framework for modelling and executing large, high-performance, decentralized, VO based THEATRE multi-agent systems. A key difference from the above mentioned agent systems is that THEATRE agents are thread-less actors that execute in computing nodes -*theatres*- which act as *agencies* which furnish basic message scheduling/dispatching, migration, communication and time management services to local actors. Theatres can coordinate to one another e.g. for global time management or to ensure termination conditions in untimed applications. Actor behaviour is modelled as a finite state machine or through a statechart (Cicirelli *et al*., 2011b). Agents can migrate from a theatre to another at runtime, e.g. for functional requirements or for load balancing. Being light-weight in character, a huge number of actors can be created to populate a complex distributed model (Cicirelli *et al*., 2009)(Cicirelli *et al*., 2011a). In the proposed approach, only theatres are exposed as grid services. Actors remain transparent to the grid. All of this favours interoperability, e.g. theatre services could be implemented in different languages and grid services could permit integrating legacy services e.g. devoted to visualization. THEATRE is currently interfaced and can also work with HLA/RTI, Terracotta, Java Sockets and Java RMI.

The rest of this paper is structured as follows. In the next section basic concepts of THEATRE are summarized. Then the software engineering design process underlying the proposed mapping of Theatre on top of GT4 is highlighted. After that, as a testbed, a scalable distributed computing example is presented, which is

Proceedings 26th European Conference on Modelling and Simulation ©ECMS Klaus G. Troitzsch, Michael Möhring, Ulf Lotzmann (Editors)
ISBN: 978-0-9564944-4-3 / ISBN: 978-0-9564944-5-0 (CD)

based on a variant of the Minority Game (Cicirelli *et al*., 2011c)(Challet & Zhang, 1997)(Challet *et al*., 2005). Finally, conclusions are given with an indication of on-going and future work.

CONCEPTS OF THEATRE

Features of the THEATRE infrastructure (Cicirelli *et al*., 2009) are logically split between (a) the *execution platforms*, i.e. theatres, which provide the environmental services supporting actor execution, migration and interactions. Services are made available to actors through a suitable API; (b) *actor components*, i.e. the basic building blocks which are programmed in Java and capture the application logic. Basic components in a theatre platform (see also Fig. 1) are (i) an instance of the Java Virtual Machine (JVM), (ii) a Control Machine (CM), (iii) the Transport Layer (TL); (iv) the Local Actor Table (LAT) (v) a Network Class Loader (NCL). The Control Machine hosts the runtime executive of the theatre, i.e. it offers basic services of message scheduling/dispatching which regulate local actors. CM organizes all pending (i.e., scheduled) messages in one or multiple message queues. During the basic *control loop*, a pending message is selected (e.g., the or one of most imminent in time) and dispatched to its destination agent by activating the relevant handler() method. At the handler() termination, the control loop is re-entered, it schedules new sent messages of last activated agent and, finally, starts its next cycle. The Transport Layer furnishes the services for sending/receiving network messages and migrating agents. Concretizations of TL refer to Java Sockets or Java RMI, HLA/RTI (Cicirelli *et al*., 2009) or Terracotta (Cicirelli *et al*., 2010). In this work TL depends on the services of the Globus Toolkit 4 (GT4). The Local Actor Table contains references to local agents of the theatre. The Network Class Loader is in charge of getting dynamically and automatically the class of an object (e.g. a migrated agent) from a network Code Server.

Actors are *reactive objects* which encapsulate a data state and communicate to one another by asynchronous message passing. Messages are typed objects. Actors are at rest until a message arrives. Message processing is atomic and constitutes the unit of scheduling and dispatching for a theatre. The dynamic behaviour of an actor is modelled as a finite state machine or a distilled statechart (Cicirelli *et al*., 2011b) which is programmed in the *handler(message)* method which receives the message to process as a parameter. Responding to a message causes in general the following reactions: (i) new actors are (possibly) created (ii) some messages are sent to known actors (*acquaintances*). For proactive behaviour, an actor can send to itself one or more messages (iii) the actor migrates to a different theatre (iv) current state of the actor is changed (*become* operation). User-defined actor classes extend the Actor abstract base class. Message classes are derived from the Message abstract base class. Actors do not have internal threads. As a consequence, message handling naturally extends the control thread of the theatre within which the agent runs. Being thread-less, a huge number of application actors can be created, with very limited demand on the underlying operating system resources. All of this improves model scalability and ensures the achievement of good execution performance (Cicirelli *et al*., 2009, 2010, 2011a). Theatres and actors are assumed to have unique names (string). At its creation, the Java reference of an actor is stored in the Local Actor Table. Subsequently, the agent can decide to move to another theatre etc. The Java reference, though, of an agent persists despite migration. After migration, in the Local Actor Table of the source theatre the agent reference is kept but now refers to a *proxy* version of the agent, which behaves as a forwarder. The proxy keeps the network information about the destination theatre where the agent migrated. Dispatching a message to a proxy actor automatically generates a network message to the destination theatre. In general, a certain number of hops can be required before reaching an agent.

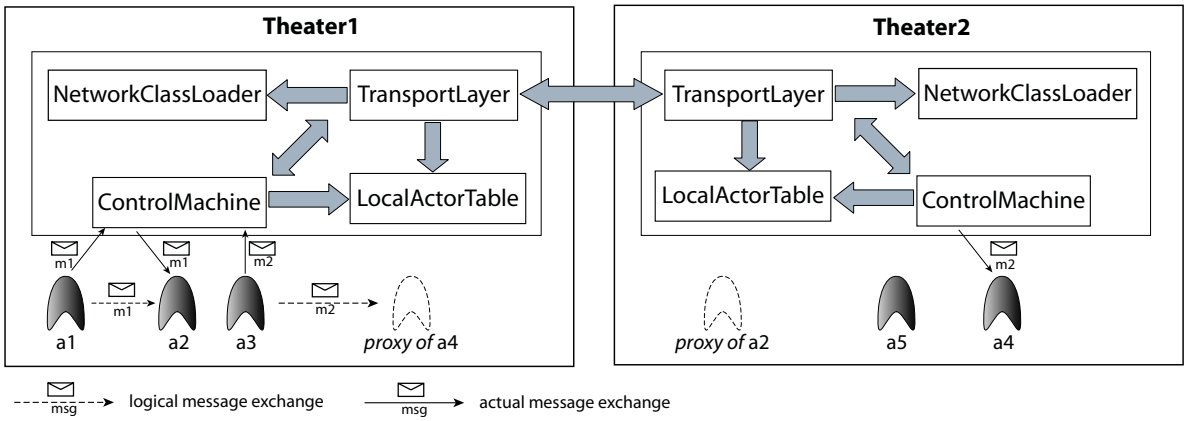

Figure 1. A THEATRE system

Of course, an agent can come back to a theatre where a proxy of itself exists. In this case, the proxy is replaced by a normal version of the actor which gets its state updated from that of the arrived agent. Agent migration

implies the relation proxy/normal of its acquaintances to be updated according to the viewpoint of the destination theatre. Some acquaintances become proxies because the corresponding actors reside in a remote theatre. Other acquaintance references can change from proxy to normal in the case the referred actor is local to the reached theatre. The update operation relies on the Local Actor Table information and the location data carried by the migrated actor. For efficiency of communications, a network message actually counts the number of hops realized for reaching its destination and automatically asks for an update of the addressing information in the proxy agent in the originating theatre.

PROTOTYPING THEATRE ON TOP OF THE GLOBUS TOOLKIT 4

The Globus Toolkit (Globus, on-line) is a de facto standard software package for developing grid systems. It includes several high-level services (for resource monitoring, service discovery, job submission infrastructure, security infrastructure, data management) useful for building grid applications. GT4, in particular, meets the requirements of the Open Grid Service Architecture (OGSA) and implements the specifications of the Web Service Resource Framework (WSRF), i.e. WS-ResourceProperties, WS-ResourceLifetime, WS-ServiceGroup, WS-BaseFaults, and related specifications of WS-Notification and WS-Addressing. In this work the Java version of GT4 was chosen for building grid services (Sotomayor & Childers, 2006), in particular Java WS Core version 4.0.8. Grid services are achieved by combining (stateless) web services with (stateful) resources according to some common design patterns. Grid service communications are based on SOAP XML-based messages which can be transported by different transport protocols (e.g. HTTP). The runtime infrastructure is the Java Web Service *container* which combines a SOAP engine, an application server and HTTP server. The container is responsible of making a grid service available to its clients.
Theatres map naturally on grid services. Fig. 2 portrays the structure of a theatre service based on GT4.

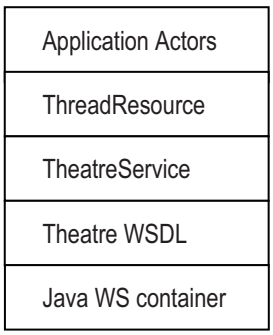

Figure 2. Design of a GT4 based theatre

Each theatre service runs on a distinct container allocated on a distinct computing node. The glue of a GT4 federation of theatres is Globus and SOAP. Each container is supposed to listen about incoming connection requests on a distinct port (8080 is the default port). TheatreService is a Java class implementing the service. The service itself is formalized through a WSDL (Web Service Description Language) file which specifies each operation the service can accept and the parameter types and return type of each operation. The WSDL also declares which SOAP messages will accompany each operation request.
As discussed in (Sotomayor & Childers, 2006), status information can be added to a stateless web service through the design of one or multiple resource instances. In this work only one resource instance is associated with a given theatre service, according to the *singleton* design pattern (Gamma *et al*., 1994) which is directly supported by Globus. TheatreResource implements in Java the internal organization of a theatre (see Fig. 1) which gives support to the execution of a collection of local application actors.
The interface of TheatreService is described in a WSDL XML file (Theatre.wsdl) which logically corresponds to the following Java interface:

```
interface Theatre{
    String getName();
    void setName( String theatreName );
    void receive( String obj );
    void start();
    void launch( String starterClass );
    void stop();
    void pause();
    void resume();
    void reset();
}//Theatre
```

As a design principle, the service interface only makes public the admitted operations (methods), i.e. it does not expose resource properties (RPs). RPs are hidden in the implementation class TheatreResource and can only be queried/modified by invoking the service operations. All the methods in TheatreServices are synchronized for concurrency control, can throw RemoteException and delegates to the actual methods of TheatreResource for the implementation aspects.
The *receive(String object)* method/operation transmits to the target theatre service an XML serialized object (actually a message or an actor). The object is first XML deserialized. Then if it is an *application message*, it gets scheduled in the control machine message queues. In the case of a *control message*, i.e. one which coordinates control machines to one another e.g. for time management, it is directly interpreted by the receiving theatre. If the object is a migrating actor, it is handled so as to update the proxy/normal relationship of the actors in the target theatre.
The *start()* method configures the target theatre so as to bootstrap its internal data structures and starts the

control thread of the control machine (note that initially there is no pending message therefore the control thread immediately blocks as it starts. The control thread will awake as soon as some message arrives).
The effective execution of a theatre federation begins by invoking the *launch(starterClass)* method on a given theatre service. This method receives the name of a starter class which gets loaded and then it is responsible of creating and initializing a first actor (e.g. a master or model actor) by sending to it an init message, on the target theatre. The master can then create further actors and send them messages thus spreading the execution.
The methods *stop(), pause(), resume()* control the execution of the theatre control machine. *reset()* resets all the theatre data structures, e.g. message queues of the control machine.

A critical point in the design framework of theatres as GT4 services, is the process of XML serialization/deserialization of messages and actors. From this point of view, the internal data component of such an object can be saved/restored into/from an associated Java bean. In addition, to each message/actor class is tied an XMLEncoder class (provided by Java) which gets the bean of the class and transforms it into standard terms of XML serialization. The encoding phase is in general complicated by the fact that a message or an actor can have acquaintance actors as internal fields and so forth recursively. The XML serialization process is capable of distinguishing the proxy/normal status of an actor in a given theatre, and generating the XML accordingly. In the case of a proxy actor, only grid service location information are generated during the serialization. The XML deserialization process in a destination theatre is the responsibility, after a minimal recourse to Java reflection, of the XMLDecoder class which does not necessitate of any adaptation.
After defining the WSDL of the service, and having implemented the TheatreService and TheatreResource classes, the operational lifecycle of GT4 grid services continues by specifying the deployment phase. A deployment descriptor file (deploy-server.wsdd) is prepared which tells the Java WS container how it should publish the theatre service. Deployment information is completed by an JNDI (Java Naming and Directory Interface) file (deploy-jndi-config.xml) which specifies, e.g., the class to be used for supporting the singleton pattern adopted by TheatreService. Java WS core improves the process of publishing a grid service by requiring a GAR (Grid ARchive) file to be created from the WSDL, the Java service+resource classes and wsdd+jndi files plus files from the Globus library. The creation of the GAR file (assisted by Apache Ant+Phyton) also compiles and generates all the *stub* classes (e.g. associated to operations parameters and return types, classes for the addressing/location of a service dynamically etc.) accompanying the given service. A Globus *deploy* command can then unpack the GAR and actually publish the service on to the container.
A federation of theatre grid services is put into execution by a client application which e.g. invokes the launch method upon a theatre service by passing to it a suitable starter class.

A DISTRIBUTED MODEL BASED ON MINORITY GAME

The prototype implementation of THEATRE on top of GT4 outlined in the previous section, was tested experimentally in a significant case using a distributed simulation of a model based on the Dynamic Sociality Minority Game (DSMG) (Cicirelli *et al*., 2011c). DMSG is a novel variant of the classical Minority Game (MG) (Challet & Zhang, 1997). In MG a fixed number of people have to decide about making use of a shared resource e.g. a bar. Since the space in the bar is limited (finite resource), the sojourn is considered enjoyable only if the number of attendances remains under a specified threshold. MG considers N (supposed odd) players that make a choice between the two options at each turn, i.e. attending the bar or stay at home. Winners are those that belong to the minority side, which is chosen by at most (n-1)/2 players. Each player gets initially a fixed and randomly chosen set of strategies that it may use to determine its next choice on the basis only of the past outcomes of the game. MG generalizes to the study of how many individuals, competing in a resource constrained environment, may reach a collective solution to a problem under adaptation of each one's expectation about the future without resorting to cooperation strategies.
DSMG assumes that information about the outcome of the previously played game step is only known to players that really attended the bar (Lustosa & Cajueiro, 2010), and that a dynamically established acquaintance relationship is available to propagate such information to non-attendant players. DSMG argues, in particular, that the capability of exploiting dynamic sociality behavior can be a key issue for modeling realistic scenarios of daily life. Consider, for instance, a player which can move on a territory. Situations can occur where acquaintances depend on the specific position owned by a player during a game step. In addition, the number of acquaintances may vary with time and can also be related to the ability of a player to establish (or maintain) social relations with other people in its nearness.
As a concrete DSMG modeling example, a road traffic scenario is considered where a single road (shared and constrained resource) connects a city and a resort, and people have to decide if to go on holyday or returning home avoiding traffic.
As in MG, N (supposed odd) players make a binary choice attempting to be in the minority side. Each player is initially fed with a randomly chosen set S of strategies that it uses to calculate its next choice on the basis only of the past M outcomes of the game. Since

there are only two possible outcomes, M is also the number of bits needed to store the history of the game. The number of possible histories is of course $P=2^M$, strategies are numerable and their number is 2^P. Players rank their own strategies on the basis of their respective ability to predict the winner side. Every player associates each strategy with a virtual score which is incremented every time the strategy, if applied, would have predicted the minority side. A penalty is instead assigned to bad behaving strategies. At each game step, a player uses the first ranked strategy. When there is a tie among possible strategies, the player chooses randomly among them. Classical MG supposes that information about the last game step is publicly available. As a consequence, the same history exists for all players.

At each game step, DSMG partitions players in three categories named *participant* (PA), *informed* (IN) and *non-informed* (NI). PA contains players that really attended "the bar" and directly know the game outcome. IN represents players that although not went to the bar they indirectly know the game outcome through their social network. NI denotes non-attendant players which remain unaware of the last game outcome. NI players are not able to update their strategies nor their history. As a consequence, players in the DSMG may accumulate a different history and may have a different view about the whole game status.

Formal definitions of DSMG

Let $O=\{-1,+1\}$ be the set of possible outcomes of the game, PL be the set of players and I be a subset of natural numbers corresponding to the game steps. Let $h: PL\times I \to O^M$ be a function modeling the history of a player, i.e. $h(p,i)$ returns the last M outcomes of the game of player *p,* preceding a given game step *i*; $h(p,0)$ is randomly set for each player. Let $S=\{S_1,...,S_n\}$ be the set of all the allowed strategies and $S_j: O^M \to O$ be a strategy function which guesses the next winner side by looking at the game history. Let $str: PL\times I \to N$ be the function which returns the index of the strategy used by player *p* at the game step *i*. The outcome of a player *p* at game step *i* is given by $S_{str(p,i)}^{h(p,i)}$. Once all players have determined their choice at step *i*, the sum of these choices defines the outcome *A(i)* of that step: $A(i)=\sum_{p\in PL} S_{str(p,i)}^{h(p,i)}$. Let $Acq: PL\times I \to 2^{PL}$ be a function determining the set of acquaintances of player *p* at game step *i*, and $Cat: PL\times I \to \{PA, IN, NI\}$ be a function which determines the category of player *p* at game step *i*:

$$Cat(p,i)=\begin{cases} PA & if \quad S_{str(p,i)}^{h(p,i)}=+1 \\ IN & if \quad S_{str(p,i)}^{h(p,i)}=-1 \wedge \exists p' \in Acq(p,i): Cat(p',i)=PA \\ NI & otherwise \end{cases}$$

Let $V_{Sj}: PL\times I \to \mathrm{Z}$, where Z is the set of integers, be the virtual score assigned by player *p* to strategy *Sj* at game step *i*. $V_{Sj}(p,0)=0$ for each strategy and for each player. Virtual scores are updated according to the following rule:

$$V_{Sj}(p,i+1)=\begin{cases} V_{Sj}(p,i) & if \quad Cat(p,i)=NI \\ V_{Sj}(p,i)-S_{str(p,i)}^{h(p,i)}A(i) & otherwise \end{cases}$$

Similarly, the history function of player *p* at game step *i* is not updated in the case $Cat(p,i)=NI$.

In this paper, the observable measures of the game are the *average of the game outcomes* $M_A=\frac{1}{T}\sum_{i=1}^{T}A(i)$, where T is the number of played game steps, and the fundamental variable for MG games in general which is the *per-capita fluctuation* of the game outcomes σ^2/N, where N is the number of players and $\sigma^2=\frac{1}{T}\sum_{i=1}^{T}(A(i)-M_A)^2$. The per-capita fluctuation is an important observable tied to player coordination. More precisely, a smaller value implies a better level of coordination among players (Sysi-Aho, 2005).

Modeling the road traffic example using THEATRE

The modeling example is split between two theatres running on two distinct network nodes (see Fig. 3). Two Win7 workstations, Pentium 4, 3.4GHz, 1GB Ram, interconnected by a 1Gbit Ethernet switch, were used for the experiments.

An odd number of players are randomly split between the two theatres. At every game step, a player who is in the city (resort) side has to decide whether to make a trip toward the resort (city) or to avoid traveling. Since the road has a limited traffic capacity, the enjoyable choice is that done by the minority of players. A player having +1 as outcome in a game step decides to make the trip. A player having -1 as outcome does not make use of the road. A player which does not make the travel may ask other players in the same place, i.e. its acquaintances, about traffic news. Due to departures and arrivals, the identity and the number of players in a place changes with time.

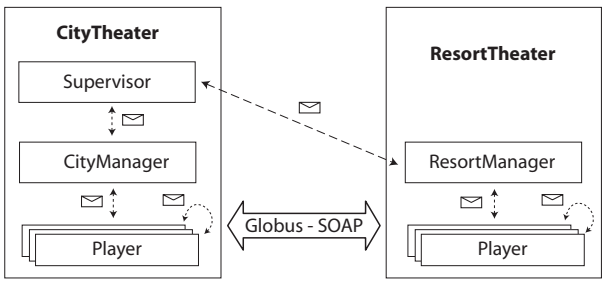

Figure 3. Theatre organization of the traffic road modeling example

Players are instances of a Player actor class. In each theatre there is a manager actor who controls the

realization of each game step by interacting with players through message exchanges. A Supervisor actor, supposed allocated to the CityTheatre, synchronizes the behavior of the two managers by signaling when the next step of the overall game can be started. The supervisor also collects statistics information about the game behavior.

Player agents which decide to move to the other side, leave a proxy version of themselves on their originating theatre and exploit the migration mechanism to transfer and operate in the partner theatre.

Fig. 4 portrays a sequence diagram about the messages exchanged during a game step. The supervisor starts the protocol by sending a StepMSG message to both managers which react to it by transmitting a StepMSG to the collection of managed players. Every player replies its own decision about the on-going game step to its manager through a ResultMSG message. Upon collecting all the replies from the relevant players, the manager replies in its turn to the supervisor with the player outcomes by another ResultMSG message. On receiving the two ResultMSGs, the supervisor first evaluates the winner side of the game (i.e. the minority side) and then sends an OutcomeMSG to the managers containing the current game outcome. After that, the managers broadcast the outcome to its managed players which update their local strategies/history. Then players reply to managers with a StepEndMSG message. Upon collecting all the required StepEndMSGs, the managers communicate to the supervisor a StepEndMSG which will allow updating the statistics in the supervisor and preparing for the next game step. For simplicity, Fig. 4 does not include the social interactions among players nor the action of migration of minority players to the other theatre.

Simulation experiments

Some simulation experiments were carried out with a fixed number of $N = 101$ players and $T = 5000$ game steps. Different configurations of the game were achieved by varying (a) the number of acquaintances that a player may contact at each game step, (b) the history size M and (c) the number of strategies assigned to players. The parameter values were chosen so as to reveal more detailed behavior hidden in the preliminary experiments documented in (Cicirelli *et al*., 2011c). In these first experiments, the number of acquaintances of a player was varied from 1 to 51, with a cutoff of behavior emerging when $|Acq|$=11. In particular, the average of game outcomes tends to be the same as for standard MG regardless both the number of strategies and history size assigned to each player. Moreover, the per-capita fluctuation of the game outcomes tends to be the same as for MG in the case M=8 whereas in the case M=2 this observable reaches its minimum to a smaller value than that obtained for the MG in the same configuration.

As the number of acquaintances reaches the value of 11, the number of *non-informed* players tends to zero, and the categories tend to become only *participant* and *informed*. As a consequence, the new experiments were planned with the following parameter values:

$M \in \{2,8\}$, $|Acq| \in \{1,3,5,7,9,11\}$, $\#strategies \in \{2,6\}$.

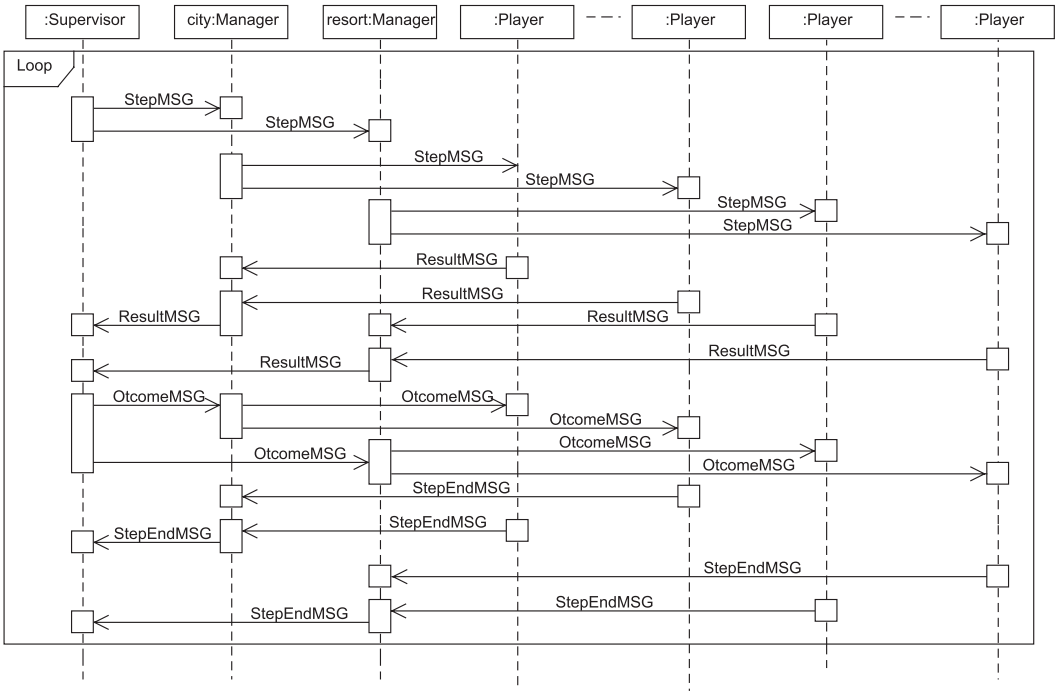

Figure 4. Message protocol at each game step

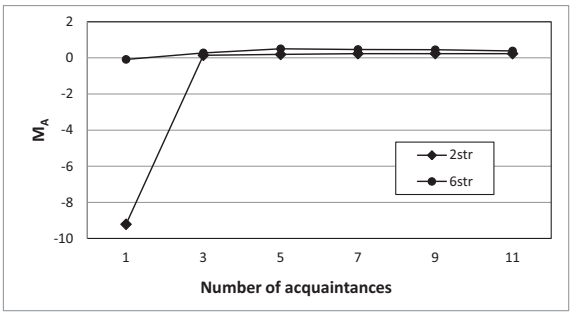

Figure 5. Average game outcomes, M=2

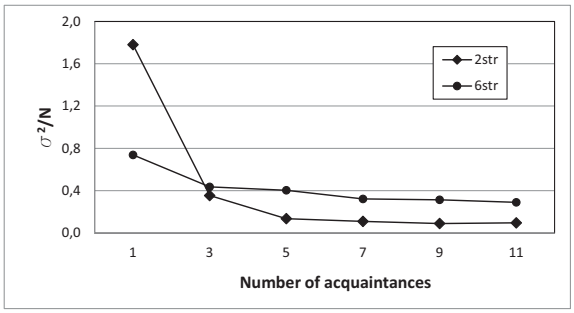

Figure 6. Per-capita fluctuation of game outcomes, M=2

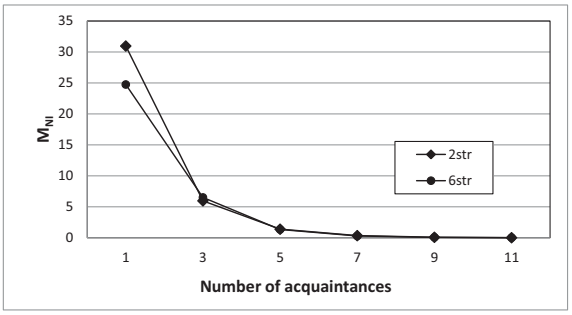

Figure 7. Average number of non-informed, M=2

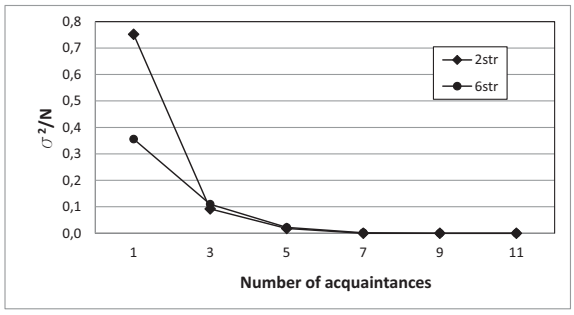

Figure 8. Per-capita fluctuation of number of non-informed, M=2

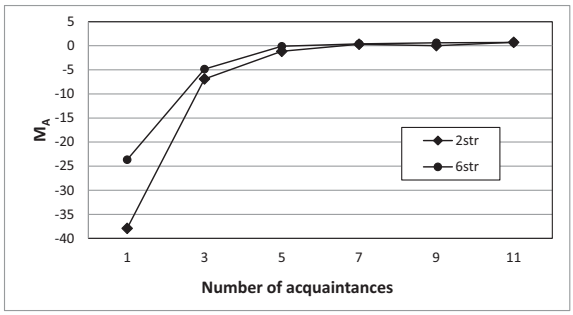

Figure 9. Average of game outcomes, M=8

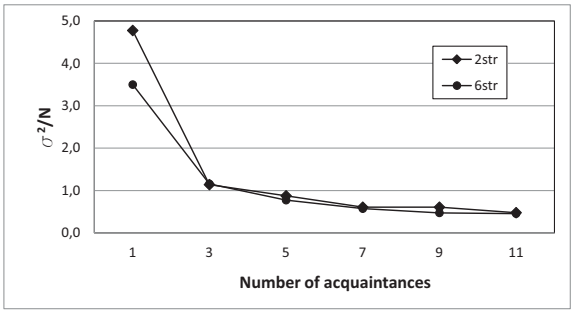

Figure 10. Per-capita fluctuation of game outcomes, M=8

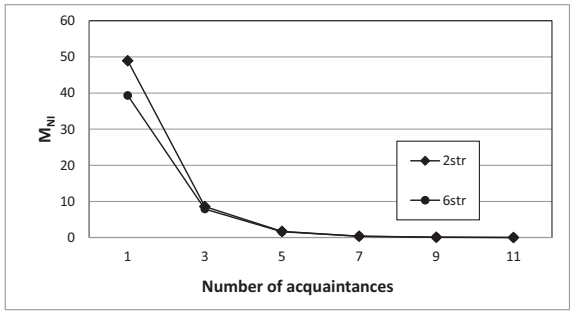

Figure 11. Average number of non-informed, M=8

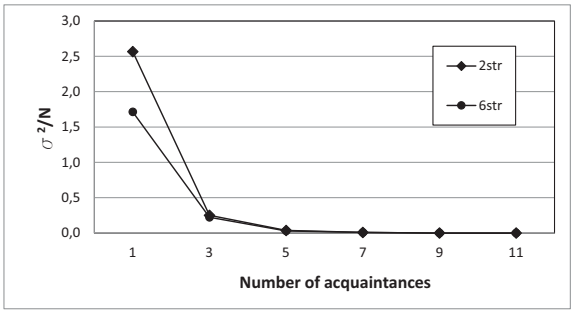

Figure 12. Per-capita fluctuation of number of non-informed, M=8

Each point in the figures from 5 to 12 was derived as the mean of five runs. Fig. 5 and Fig. 6 show respectively the average of game outcomes and the per-capita fluctuation of game outcomes when M=2. As one can see, as the number of acquaintances increases toward 11, a saturation phenomenon occurs and M_A tends to zero whereas σ^2/N reaches its minimum value. Moreover, as Fig. 6 witnesses, the case S=2 performs better than that S=6.

Fig. 7 and Fig. 8 confirm that by increasing the acquaintances number from 1 to 11, the behavior of the non-informed players tends to disappear definitely from the game because the number of non-informed players tends to zero. The same behavioral character is exhibited by the Fig. 9 to Fig. 12 which are concerned with the case M=8. In this scenario, though, as one can see from Fig. 10, the per-capita fluctuation is greater than that for M=2, for both S=2 and S=6. From the experiments seem to emerge that there is no benefit to

have a deeper history (i.e. M>2) with DSMG. Rather, it is more convenient to exploit a minimal social network (e.g. a number of acquaintances about $|Acq|=5$).

Scaling Issues

The chosen DSMG model used for the experiments is very challenging: after each game step about an half of the existing agents migrates to the other side of the road, making the model almost communication bound. Model scalability was studied by varying the number of players (from 11 to 100001) and measuring the amount of wallclock time (WCT) required by each game step. Fig. 13 shows the Normalized WCT (that is the ratio between the WCT and the number T of simulation game steps) vs. the number of players. Variables on both axes are reported in logarithmic scale. As Fig. 13 witnesses, the model seems to scale very well, thus talking about scalability of the whole Theatre/GT4 approach.

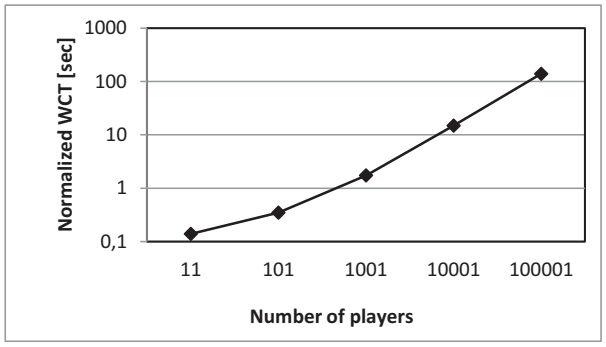

Figure 13. DSMG model scalability

CONCLUSIONS

This paper describes a novel approach and prototype implementation of using the THEATRE agent architecture (Cicirelli *et al*., 2009) on top of the Globus Toolkit 4 (Sotomayor & Childers, 2006), which aims at experimenting with grid computing in the M&S of complex multi-agent systems. The realization favors interoperability and opens to an exploitation of virtual organizations of computing resources and secure communications. A practical application of the proposed approach is demonstrated by a distributed simulation of an agent-based model which depends on the Dynamic Sociality Minority Game (Cicirelli *et al*., 2011c).

Future and on-going work is directed at:

- Improving the grid service implementation of theatres.
- Extending THEATRE/GT4 so as to give support to general time-management scenarios e.g. as described in (Cicirelli *et al*., 2009, 2010) which are based on a distributed conservative algorithm.
- Experimenting with THEATRE/GT4 in the M&S of large and scalable situated multi-agent systems (Cicirelli *et al*., 2011a) which relies on a composite time notion.
- Applying Dynamic Sociality Minority Game to more complex applicative scenarios.

REFERENCES

Avila-Rosas A., L. Moreau, V. Dialani, S. Miles, X. Liu. 2002. Agents for the Grid: a comparison with web services (Part II: Service Discovery). *Proc. of Workshop Challenges in Open Agent Environments*, pp. 52-56.

Cicirelli F., A. Furfaro, L. Nigro. 2009. An agent infrastructure over HLA for distributed simulation of reconfigurable systems and its application to UAV coordination. *SIMULATION - Transactions of the Society for Modeling and Simulation International*, vol. 85/1, pp. 17-32, SAGE.

Cicirelli F., A. Furfaro, A. Giordano, L. Nigro. 2010. Parallel Simulation of multi-agent systems using Terracotta. *Proc. of 14th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DSRT2010)*, pp. 219-222.

Cicirelli F., A. Giordano, L. Nigro. 2011a. Distributed simulation of situated multi-agent systems. *Proc. of IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DSRT2011)*, September 4 - 7, Manchester, UK, pp. 28-35.

Cicirelli F., A. Furfaro, L. Nigro. 2011b. Modeling and simulation of complex manufacturing systems using statechart-based actors. *Simulation Modeling Practice and Theory* 19/2, pp. 685-703, Elsevier.

Cicirelli F., A. Furfaro, L. Nigro, F. Pupo. 2011c. Dynamic Sociality Minority Game. *Proc. of European Conference on Modelling and Simulation*, 7-10 June, Krakow, pp. 27-33.

Challet D., Y.C. Zhang. 1997. Emergence of cooperation and organization in an evolutionary game. *Physica A: Statistical and Theoretical Physics*, 246(3-4):407–418.

Challet D., M. Marsili, Y. Zhang. 2005. *Minority games*. Oxford University Press.

Foster I., C. Kesselman, S. Tuecke. 2001. The anatomy of the Grid: Enabling scalable virtual organizations. *International J. of Supercomputer Applications*, 15(3).

Fukuda M., D. Smith. 2006. UWAgents: a mobile agent system optimized for grid computing. *Proc. of GCA'2006*, pp. 107-113.

Gamma E., R. Helm, R. Johnson, J. Vlissides. 1994. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Prof.

Globus, on-line: http://www.globus.org.

Lustosa B.C., D.O. Cajueiro. 2010. Constrained information minority game: How was the night at El Farol?. *Physica A: Statistical Mechanics and its Applications*, Vol. 389, Issue 6, pp. 1230-1238.

Moreau L. 2002. Agents for the Grid: a comparison with web services (Part I: Transport Layer). *Proc. of 2nd IEEE/ACM Int. Symposium on Cluster Computing and the Grid*, pp. 220-228.

Sotomayor B., L. Childers. 2006. *Globus Toolkit 4 – Programming Java Services*. Morgan Kaufmann, Elsevier.

Sysi-Aho M. 2005. *A Game Perspective to Complex Adaptive Systems*. Ph.D. Thesis, Department of Electrical and Communications Engineering, Helsinki University of Technology, Espoo, Finland.