A Middleware for Interfacing with Simulation Systems of Multi-Agent Models

B.G.W. Craenen
School of Computer Science
University of Birmingham
Edgbaston, B15 2TT
Birmingham, United Kingdom
craenbgw@cs.bham.ac.uk

V. Suryanarayanan
School of Computer Science
University of Birmingham
Edgbaston, B15 2TT
Birmingham, United Kingdom
vys@cs.bham.ac.uk

G.K. Theodoropoulos
School of Computer Science
University of Birmingham
Edgbaston, B15 2TT
Birmingham, United Kingdom
gkt@cs.bham.ac.uk

ABSTRACT

As multi-agent systems (MAS) are used increasingly often to solve larger and more complex problems, distributed simulation is emerging as the only viable approach to cope with the resultant increased scale and complexity of MAS. While different methods have been proposed for distributing these simulations, the problem of interfacing a MAS with these distributed methods remains just as important. In this paper we use the PDES-MAS framework as a solution for distributing the MAS and propose a Middleware as an interface between MAS and the PDES-MAS framework. The Middleware supports designing and writing a MAS, translates interactions between the shared state of agents to events that the PDES-MAS framework can handle, and offers a number of commonly used services in a MAS. Practical experience with the Middleware thusfar has shown it to be an adaptable and efficient solution for designing and implementing different MAS.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; I.6 [**Simulation and Modelling**]: Model development

General Terms

Distributed systems, Simulations, Multi-agent systems

Keywords

Middleware, parallel discrete event simulation, multi-agent systems, interface

1. INTRODUCTION

Multi-Agent Systems (MAS) are commonly used to model and solve problems that are either difficult or impossible for an individual agent, or monolithic system to solve. Examples of problems appropriate for MAS research include online trading, disaster response, and modelling social structures. The problems MAS are tasked to solve or model are becoming increasing larger and more complex however, and the computational resources required by the MAS to do so have likewise increased to such an extend that they now often surpass the resources available of even the biggest single computer platforms. This has lead to research into distributing MAS over multiple computer platforms, all working together to solve or model the problem at hand.

Various approaches for distributing MAS over multiple computer platforms have been proposed [14]. One such approach focuses on partitioning the simulation topology into several semi-autonomous regions distributed over the computing resources available. Another approach is to partition the shared state of the simulation using a Distributed Shared Memory (DSM) system and distributing this over the computing resources available. An example of the latter approach is the PDES-MAS framework [9]. These, and other approaches all have specific advantages and limitations, making them particularly appropriate for certain MAS and problems, but less so for others. All however share the problem of having to translate MAS behaviour or topology into a format suitable to distribution using the approach used.

This paper will investigate an interface between the MAS in the format of an Agent-based Model (ABM) where the MAS is distributed using the DSM approach, specifically PDES-MAS. Conceptually this interface, which we will call the Middleware, is located between the ABM on top, and PDES-MAS at the bottom. The functionality of the Middleware can then be described as translating the interactions of agents in the MAS with the shared state as described by the ABM into events that can be handled by PDES-MAS. Likewise, however, specific return actions coming up from the PDES-MAS to the ABM of the MAS need to be handled as well, and this refers in particular to synchronisation management of the shared state. In addition, the Middleware has the purpose of hiding from the MAS the specific architecture used to distribute it, so that the ABM of the MAS can be developed independent from this architecture.

This paper is organised as follows. In the next section (2) we will briefly discuss the MAS and the PDES-MAS framework. In section 3 we will provide a detailed description of the Middleware itself and in section 4 we discuss a use-case of the Middleware as well as some practical issues. Finally, the paper will provide some conclusions in section 5.

DISIO 2011, March 21, Barcelona, Spain

DOI 10.4108/icst.simutools.2011.245572

2. MAS AND PDES-MAS

A MAS simulation defines a number of agents that are assumed to be intelligent, where the agents interact with each other and with their environment, often also captured within agents. In addition, a defining characteristic of agents is their autonomy [15]. Because the agents act independently and intelligently, based on information from its environment, it is often hard to predict these interactions in advance. Indeed discover them at all. Moreover, how the agents interact with each other and the environment is often the primary goal of the simulation. In conventional MAS, the agents go through a so-called sense-think-act cycle. First the agent gathers information about itself, other agents, and its environment. This is then used to determine a (pre-described) behaviour later to be translated into actions on itself, other agents, or its environment. In this paper we assume that over the life-time of the MAS simulation, agents go through many of these cycles, and this defines the progression of time as experienced by the agent.

Agents in a MAS simulation encapsulate a number of variables some of which are visible or public to, or shared with, other agents, while others remain private and available only to the agent itself. Values of all shared and private variables, at each step in the agent's time progression, determines the state of the agent at that time. The collective state of the shared variables can then be seen as similar to the MAS simulation's space-time memory [1, 10]. Shared variables offer a natural representation of the simulation context when interactions between agents are described as interactions on these variables. As such, we assume that agents alter the state of the MAS simulation by interacting with these shared variables in an event-based fashion.

Space-time memory in distributed computing is commonly made available across distributed computer platforms through a DSM system, often without exposing the exact way in which the memory is accessed or organised internally by the DSM. Access to the shared memory is provided through read and write operations performed as events, mirroring our assumptions about the MAS given above. The read- and write-events allow the reading of a value stored at a memory address (called an ID-query) and write a value to a certain memory-address (called an ID-write). Memory addressed are allocated by the DSM and are specific, with indexing over memory-addresses offered as a feature of the DSM. Reading of values over a range of memory-addresses that satisfy a condition (associative memory), is an extension of a single read-event and is called a range-query. Range-queries in a MAS are often used to retrieve aggregate information during the sense-cycle of the information, e.g., retrieving the location of all agents within viewing range [13].

In a MAS simulation, a DSMS system should provide simultaneous distributed access to the shared data of the simulation. And in this context, maintaining data consistency is an important part of the functionality of the DSM system. In distributed simulations, two main synchronisation mechanisms for maintaining data consistency can be generally recognised: conservative synchronisation and optimistic synchronisation. Conservative synchronisation disallows conflicting access to the data by predicting when conflicting access will occur and then apply strict access-rules (preemptive locking) to the data. Optimistic synchronisation allows free access to the data at all times but repairs data inconsistency afterwards through a roll-back mechanism [5].

In this paper the PDES-MAS framework is used as an implementation of a DSM system for MAS. PDES-MAS is a framework for the distributed simulation of MAS [9]. It implements a DSM structure where the shared state, i.e., the visible and publicly accessible attributes or variables of the agents, is represented by Shared-State Variables (SSV). SSVs are data-structures that store the time-stamped history of values of a particular variable over time [5]. Following the Parallel Distributed Event Simulation (PDES) paradigm, agents in the MAS are assigned to Logical Processes (LP), known as Agent Logical Processes (ALP), while SSVs are assigned to Communication Logical Processes (CLP). An ALP potentially models more than one agent, with multiple ALPs allowed concurrent access to the set of SSVs associated to the agents by connecting them to a tree-like network of CLPs. SSVs are then distributed among the CLPs in a scalable and balanced manner. Figure 1 shows a depiction of the PDES-MAS framework with 4 ALPs, 3 CLPs, and 4 SSVs.

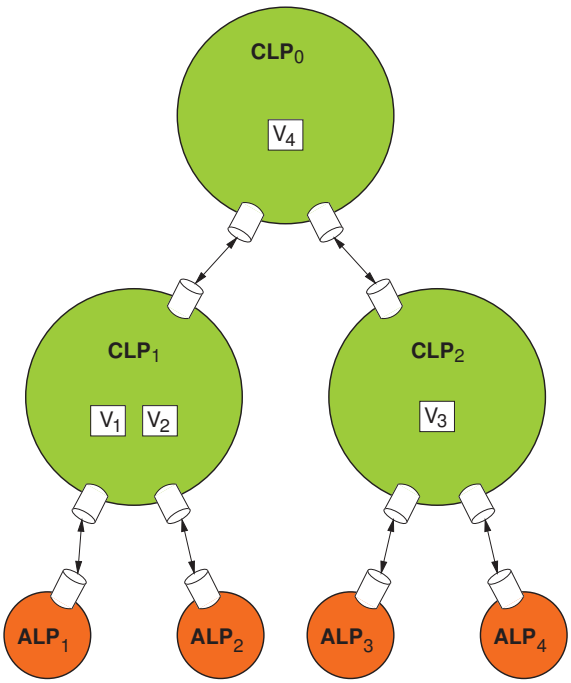

Figure 1: The PDES-MAS framework.

The PDES-MAS framework allows SSV access through time-stamped events along the lines described earlier for interactions within a MAS simulation, i.e., within the sense-cycle the PDES-MAS framework supports ID-query and Range-query events, while during the act-cycle the PDES-MAS framework supports write-events. ALPs are connected to the leaf CLP nodes in the tree-like structure, and an ALP issues SSV access events through its parent CLP. If the SSV is not assigned to the parent CLP, the access request is passed along the tree to the CLP where the SSV is located. The return information is then passed back along the same route in reverse, to the parent CLP and from there to the ALP. Control messages internal to the PDES-MAS framework are

conveyed through the tree-like structure in a similar way. Conceptually the ALP provides the access-node for the MAS simulation to the PDES-MAS framework.

Synchronising the events received from the ALP is the responsibility of the CLP. The PDES-MAS framework uses optimistic synchronisation, the specifics of which have been reported in [2, 4, 3, 5, 6, 7, 8]. In general, each SSV is associated with a list of Write Periods (WP), representing the values of the variables at different times throughout the time progression of the simulation. When a WP is invalidated by a straggler write, any agents associated to the ALPs that have read that WP subsequent to the straggler write will be asked to roll-back to the time-stamp before the straggler write. An agent that is rolled back is then supposed to resume its time-progression from that time, using the now consistent data. As such, the data consistency is repaired for that time-stamp.

The CLP tree-like structure in the PDES-MAS framework is reconfigured dynamically and automatically so as to reflect the interaction patterns of the agents as exhibited through the access patterns of the SSVs. SSVs that are accessed most frequently are pressured to move closer in the tree-like structure to the ALP handling that agent, i.e., towards the parent leaf CLP. The aim is to concurrently self-organise the SSVs in the tree so as to minimise the average number of hops required to access them, as well as to reduce the load imbalance between the CLPs. Reconfiguration of the CLP tree-like structure can be achieved by creating or deleting CLPs, moving ALPs to different parent CLPs, or by migrating SSVs between CLPs. In the PDES-MAS implementation used in this paper, the CLPs are configured in a fixed binary tree-structure with the leaf CLPs hosting a fixed and constant number of ALPs. Only SSVs are migrated through the tree to achieve redistribution [12].

3. THE MIDDLEWARE

The Middleware functions as the glue between the MAS simulation and the PDES-MAS framework and serves three overall goals: It provides a framework in which it is relatively straightforward to design and build a MAS; It translates interactions between the agent's shared state variables into events for the PDES-MAS framework to handle; And it provides a means for the agents in the MAS to handle return events, from the PDES-MAS framework, like roll-backs, to take effect in the MAS. Figure 2 depicts the overall architecture of the Middleware and we will be discussing further below.

3.1 Distributed Object Template

The Middleware supports the design and development of a MAS by providing an agent template. All agents in the MAS are to implement the so-called *Distributed Object Template*. The Middleware itself acts as the MAS counterpart of the PDES-MAS framework's ALP with several agents or distributed objects handled by each ALP. In order for the PDES-MAS framework's ALP to associate SSVs to these agents, a unique agent or distributed object identifier is maintained, so that for the set of all agent identifiers A, each agent has a unique identifier $a \in A$. In addition, the various variables associated with the agents in the MAS are defined and associated with a unique variable identifier, so that for the set of all variable identifiers V, each variable has a unique identifier $v \in V$. Each variable has associated with it a variable

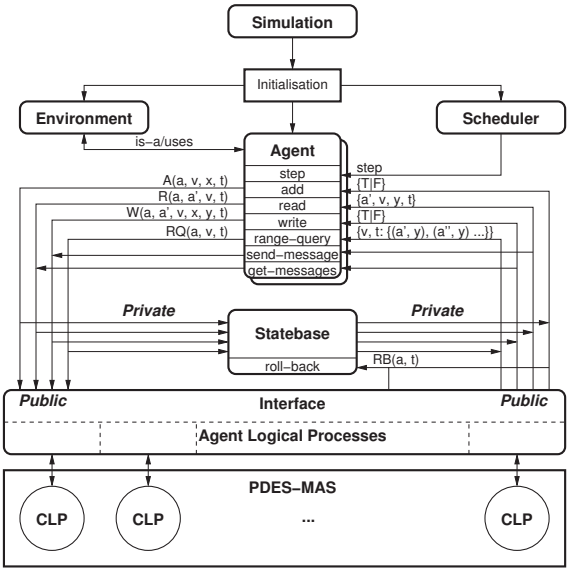

Figure 2: The Middleware architecture.

type so that for the set of variable types X, each variable v has a variable type $x_v \in X$. Time progression in the MAS is defined by a set of increasing timestamps T where $t_i \in T$ is the time-stamp at time i with $0 \leq i \leq n$ where n is the maximum time-stamp in T and also of the MAS.

3.2 Shared or Private Variables

Variables can be annotated as either public or shared, or private and not-shared: v^s or $v^{\neg s}$, with v^s equivalent to an SSV in the PDES-MAS framework. There is no restriction on whether a variable is shared or private over different types of agents. Agents can, for example, have a public or shared variable for one type of agent, while that same variable is private and not shared for another type of agent. All shared or public variables, or SSVs, are handled by the PDES-MAS framework, while all private variables are maintained internally by the Middleware. To enforce this, the agent template provided by the Middleware encapsulates all interactions with all variables. All interactions, like adding a variable, reading a variable, writing a variable, or performing a range-query over a variable, are encapsulated without exception. It is this that allows the Middleware to determine whether the variable is declared public and thus pass the interaction to the PDES-MAS framework, or whether the variable is private and thus handle the interaction in the Middleware itself.

The particular implementation of the PDES-MAS framework itself poses a couple of constraints on a number of interactions. Adding variables has to occur before the experiment with the MAS begins, i.e., during the initialisation of the MAS. This to allow assignment of the SSVs to the CLPs in the PDES-MAS framework. No variables may be added to agents in the MAS after the MAS experiment has started. In other words, the shared state space of the simulation is fixed from the beginning to the end of the simulation. As to the set of variable types (X) allowed, our particular implementation allows integers (INT),

doubles (DOUBLE), strings of characters (STRING), and a POINT-type where POINT is a tuple of two integers: $X = \{\text{INT}, \text{DOUBLE}, \text{STRING}, \text{POINT}\}$. The PDES-MAS framework can handle other variable-types as well, as its internal handling of SSVs, including WPs, is variable-type agnostic. But in generally this set of variable-types should suffice for most MAS. The POINT-type is commonly used to express the location of the agent in the MAS in a two dimensional environment. The STRING-type is used, and the agent template in the Middleware supports this explicitly, to provide a messaging system for the agents to use. To this end, the agent or distributed object template uses a special MESSAGES SSV as a message-box variable, and provides access methods for this variable so that messages received can be retrieved and send through the PDES-MAS framework.

3.3 Agent interactions with PDES-MAS

The Middleware allows the agent to interact with variables in four different ways: add, read, write, and range-query.

The *add* interaction allows an agent to add a variable to its set of variables during initialisation. Adding public variables is passed onto the PDES-MAS framework while private variables are handled in the Middleware itself. Where a is the identifier of the agent adding the variable, v the variable to be added, x the variable type of the variable to be added, and t the time-stamp at which the variable is added (constrained to $t = 0$ here), the add interaction can be described as $A(a, v, x, t) \rightarrow \{T|F\}$, with the interaction returning either true (T) if the interaction was successful, or false (F) otherwise ($\{T|F\}$ representing the boolean set).

The *read* interaction allows an agent to read a variable, either from its own set of variables, or from the set of public variables. The read interaction returns the value stored in either the PDES-MAS framework if the variable is public, or the value stored in the Middleware if the variable is not (private). Where a is the identifier of the agent reading the variable, a' the identifier of the agent in whose variable set the variable is located ($a = a'$ is possible), v the variable identifier, and t the time-stamp of the read interaction, and y the value stored by the variable. As such, the read interaction can be described as $R(a, a', v, t) \rightarrow \{a', v, y, t\}$.

The *write* interaction allows an agent to write a value to a variable, either to its own private variables ($v \in V^{\neg s}_{a=a'}$) or an agent's public variables ($v \in V^{s}_{a\neq a' \vee a=a'}$). Again, private variables are handled in the Middleware itself, public variables by the PDES-MAS framework. The write interaction returns true (T) if the interaction is successful and false (F) otherwise. Where a is the identifier of the agent writing the variable, a' the identifier of the agent in whose variable set the variable is located, v the variable identifier, y the value of the variable (of type $x \in X_v$) and t the time-stamp of the write interaction. The write interaction can then be described as $W(a, a', v, x, y, t) \rightarrow \{T|F\}$.

The *range-query* interaction allows an agent to discover the identifiers of public variables satisfying a predicate. The range-query interaction returns a list of all agent identifiers associated with a value of the queried variable. Where a is the identifier of the agent initiating the range-query interaction, v the identifier of the range-queried variable, and t the time-stamp of the range-query interaction, the range-query interaction can be described as $RQ(a, v, t) \rightarrow \{v, t : (a', y), (a'', y), \ldots\}$. A constraint over the values applicable for addition to the reset-set can be added, for example an upper and lower bound (not depicted).

3.4 Scheduler

The PDES-MAS framework provides access to SSVs using events that are timestamped. To support this, the Middleware provides a time-progression service to the MAS. There are two options for implementing this service of which the latter is currently implemented. The first option is to have each agent or distributed object in the MAS maintain its own time-progression with the Middleware then providing asynchronous access to the PDES-MAS framework. Although this corresponds closely to the asynchronous nature of the message passing design of the PDES-MAS framework, it would also require the Middleware to copy this design with all the required inter-agent synchronisation that that entails (each agent would have its separate thread to maintain and protect). Instead, the Middleware implements a synchronised although distributed solution through the inclusion of a *Scheduler*. The scheduler's task is then to maintain a list of all agents or distributed objects on a single ALP in the PDES-MAS framework and call upon all agents in that list to perform its sense-think-act cycle encapsulated in a *step* method, thus providing time-progression and synchronisation. An advantage of such a synchronised design is that there is a single entity that handles simulation control messages, like roll-back requests, about which more further on. It also means that such a design leads to two different time-progression mechanisms having to cooperate closely. On the one hand there is the progression of time according to the agent itself, to be used as a time-stamp for PDES-MAS framework events. On the other hand there is the time-progression according to the scheduler itself based on the times it calls the agent's sense-think-act cycle. The latter however is exclusively internal, with no explicit need to be known by the MAS. A further advantage to the centralised scheduler design is that the scheduler can be accessed to provide data about an agent's state for generating results or analysis of the MAS itself. In a design without a centralised scheduler this would have to be provided through polling the agents individually, or through some other service added specifically for this. With the centralised scheduler design, the scheduler polls the agents in the MAS for the user, with the user only required to provide a means of collecting the results.

3.5 Interface

Access from the Middleware to the PDES-MAS framework is provided by the *Interface*. The Interface can be seen as consisting of two parts: one connected to the Middleware, and another connected to the PDES-MAS framework. Encapsulating variable interactions in the agent or allows the Middleware to redirect the appropriate events to the part of the interface that is connected to the Middleware. That part of the interface is infrastructure agnostic, in that it has no, and needs no, knowledge of the underlying infrastructure the PDES-MAS framework. Methods in this part of the infrastructure reflect the interaction types as encapsulated in the agent template: adding variables, reading variables, writing variables, and performing range-queries. Information to use these methods is maintained in the Middleware itself: agent identifier, variable identifier, variable type, time-stamp, and so forth (see above). The part of the interface connected

to the PDES-MAS framework implements an ALP in the PDES-MAS framework and this requires it to be infrastructure aware. This part of the interface interacts with its parent CLP, using the message passing mechanism used internally in the PDES-MAS framework. Internally the interface needs to translate variable interaction events into messages to be passed along to the parent CLP in the PDES-MAS framework and wait for a response from the PDES-MAS framework. When the interface receives the response it is translated into an appropriate response for the agent to use.

For example: a read interaction in the sense-cycle of an agent is encapsulated in a get-method in the agent template in the Middleware. If the variable is public, this interaction is passed along to the interface together with information as to which agent the variable wants to read it, to which agent the variable belongs, which variable is to be read, and at which time-stamp (that of the agent itself). The interface turns this event into a ID query message, passes this to its parent CLP, and waits for a response-message. Upon receiving the response-message, the read value for the time-stamp in the PDES-MAS framework, stored in the response-message, is then turned into a return value for the get-method call by the agent and passed along to it. All other interactions go through a similar translate-to and translate-back process in the interface.

3.6 Roll-backs and Statebase

The example given above describes the process of a normal event. There is an exception to this process, and this occurs when the PDES-MAS framework issues a roll-back request instead of a response-message. Roll-back requests are issued when a straggler write has occurred. This is a write event that invalidates read (and range-query) events that happened after the time-stamp of the write event. PDES-MAS uses an optimistic synchronisation strategy in which events invalidated by a straggler write event need to be repaired, or, rolled-back. The term roll-back can be explained by the need for the agent to roll back in time to the time-stamp before the invalidating event was issued. From that time-stamp it has to continue with the now consistent SSV value. As such, when a straggler-write occurs, the PDES-MAS framework sends roll-back requests to all agents that have read the now altered variable.

Roll-back requests issued by the PDES-MAS framework and received by the Middleware can pertain to either the agent currently issuing events, or another agent altogether. Both cases have to be handled, and both are handled differently. In the case where the agent currently issuing events needs to be rolled back, the current sense-think-act cycle has to be interrupted. In the case where the agent is not currently issuing event, no interruption of the sense-think-act cycle is required. The interface of the Middleware is extended to handle both cases correctly, i.e., in both cases the roll-back request is translated correctly. Where a is the identifier of the agent to be rolled-back and t the time-stamp to which the agent has to be rolled back, a rollback request can be described as: $RB(a, t)$. Given that the PDES-MAS framework itself maintains data consistency for the SSVs, or public variables, the Middleware needs only to roll-back the private variables maintained there. In order to do so, the Middleware maintains a *statebase* for those variables similar to the Write-Period design as maintained by the PDES-MAS framework. Encapsulation of the variable interactions in the distributed object allows maintenance of the statebase to occur opaque to the designer and developer of the MAS. Note that only shared variables can have straggler writes, and as such, no roll-back requests will ever be issued from private variables maintained by the Middleware.

4. THE MWGRID USE-CASE

The Middleware is being used and experimented with in the context of the MWGrid project. The MWGrid project ([11]) is a collaborative project between the Institute of Archeology and Antiquity and the School of Computer Science at the University of Birmingham. It seeks to address the problems associated with early military logistics through agent-based modelling and distributed simulations. In medieval states the need to collect and distributed resources to maintain armies affected all aspects of the political organisation of the state. And at critical times, when armies failed, the results of a failure could prove disastrous to society. Despite this, even though military activity in terms of resource allocation and consumption is decisive in shaping pre-modern societies, military studies seldom progress past the study of existing texts to bear out the pragmatic consequences of military behaviour.

In this context, the MWGrid project explores the military-logistical context of the Battle of Manzikert in 1071, a key historic event in Byzantine history. The defeat of the Byzantine army by the Seljuk Turks, and the following civil war, resulted in the collapse of Byzantine power in central Anatolia. In the MWGrid project the Middleware and PDES-MAS framework combination was used to design and build a MAS for modelling the implications for pre-industrial societies of marching a large medieval army across the breadth of the Anatolian mainland of the Byzantine empire. Agents in the MAS represent all participants of the army from the emperor down to the individual soldier, all by extending the agent template provided by the Middleware. Agent and environment attributes and variables are based on detailed historical and geological analysis (see [11] for more details of the model). The MAS based on the Middleware and PDES-MAS framework combination allows running what-if scenarios with different configurations and, for example, army sizes. The result of these experiment are expected to have significant implications for the study of pre-industrial societies in methodological and theoretical terms, and will benefit academics with an interest in comparative military history, the cultural role of military organisation and the relationship of historical and modelled data.

Although experimentation and performance analysis is still ongoing, in the MWGrid project we found the Middleware to be an adaptable and efficient solution for designing and implementing the MWGrid MAS. Within the MWGrid project, we used the Middleware to design a series of ever more complex MAS, suggesting that the Middleware provides a solution not just for designing and implementing the MWGrid MAS, but for MAS in general. The Middleware hides a great deal of specific PDES-MAS framework or DSM dependencies, so that designing a MAS with the Middleware is generally concentrated on designing the agents themselves, providing the environment they live in, and defining the interactions they have with themselves or this environment.

The Middleware proved to be quite adaptable as it allowed us to design and develop a series of ever more complex MAS without the need for changes to the Middleware itself. In

addition, because of the modular design of the interface part of the Middleware, it should be relatively easy to adapt the Middleware to support other DSMs besides the PDES-MAS framework, and do so without having to adapt to majority of the Middleware itself, or the MAS using it. Although the latter was not required in the MWGrid project, the ease with which it was possible to extend the Middleware with a sequential substitute of a DSM shows its potential. There are ofcourse limitations, for example, the Middleware still expects that the underlying DSM uses an optimistic synchronisation methods using roll-backs.

In this respect it is worth mentioning that most of the Middleware, and the MWGrid MAS were developed in Java, while the interface was developed in as a JAVA-JNI interface to provide the connection with the C++ implementation of the PDES-MAS framework.

The Middleware also proved to be an efficient solution. Preliminary measurements showed that on average, time spend on the various functions of the Middleware was negligible compared to the time spend required by the various functions in the MAS, or by the PDES-MAS framework for that matter. The most costly function of the Middleware is the maintenance of the private variables but even there the Middleware showed itself to be very efficient indeed. A more indepth performance analysis of the Middleware is expected to be published shortly.

The specific implementation used in the Middleware, with a centralised scheduler calling step-functions, does however mean that a waiting period is incurred in the Interface. How the agents are distributed among the different nodes, and how the architecture is configured over the distributed platform, has a direct effect on the performance of the system, as is to be expected. Overloaded nodes handling too many agents in the Middleware, or handling too many SSVs in the PDES-MAS framework, have an adverse effect on the performance of the system as a whole and this is reflected in the wall-clock time needed to run the experiment. Careful design of the distributed platform architecture and the dynamic load-balancing done within the PDES-MAS framework do mitigate these adverse effects however.

5. CONCLUSION

As MAS are used increasingly often to solve larger and more complex problems, distributed simulation is emerging as the only viable approach to cope with the resultant increased scale and complexity of MAS simulations. One of the pivotal questions that requires answering is how to distribute the MAS over multiple computational platforms in an efficient and effective way. One answer to this question is to distribute the shared state of the agents of the MAS over the different computational platforms. Several ways of doing so have been proposed, and in this paper we use an implementation of the PDES-MAS framework. What remains is a means of designing and building a MAS using the PDES-MAS framework and in this paper we explore a solution to this problem called the Middleware.

The Middleware provides an agent template that can be extended to straightforwardly implement the agents in a MAS. In addition it is used to translate agent interactions, like add, read, write, and range-query interactions into events that can be issued directly to the PDES-MAS framework. It also provides services for maintaining time-progression for the agents in the MAS as well as handling specific features of the PDES-MAS framework, like roll-back requests. In general the Middleware provides these functions in a way that is both efficient and effective and it has already been used in the MWGrid Project at the University of Birmingham [11]. During the MWGrid project, the Middleware showed itself to be an adaptable and efficient solution for designing and implementing a variety of MAS. Experiments are currently being conducted to evaluate the Middleware and quantify the overhead it induces to the overall performance of the simulation. An in-depth performance analysis of the Middleware is expected to be published shortly.

6. REFERENCES

[1] K. Ghosh and R. M. Fujimoto. Parallel discrete event simulation using space-time memory. In *Proceedings of the International Conference on Parallel Processing, Volume III, Algorithms & Applications*, pages 201–208, 1991.

[2] M. Lees, B. Logan, C. Dan, T. Oguara, and G. Theodoropoulos. Decision-theoretic throttling for optimistic simulations of multi-agent systems. In A. Boukerche, S. J. Turner, D. Roberts, and G. Theodoropoulos, editors, *Proceedings of the Ninth IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2005)*, pages 171–178, Montreal, Quebec, Canada, Oct 2005. IEEE Press.

[3] M. Lees, B. Logan, C. Dan, T. Oguara, and G. Theodoropoulos. Analysing probabilistically constrained optimism. In E. Alba, S. J. Turner, D. Roberts, and S. J. Taylor, editors, *Proceedings of the 10th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2006)*, pages 201–208, Malaga, Spain, Oct 2006. IEEE Press.

[4] M. Lees, B. Logan, C. Dan, T. Oguara, and G. Theodoropoulos. Analysing the performance of optimistic synchronisation algorithms in simulations of multi-agent systems. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation (PADS06)*, pages 37–44, Washington, DC, USA, 2006. IEEE Computer Society.

[5] M. Lees, B. Logan, and G. Theodoropoulos. Adaptive optimistic synchronisation for multi-agent simulation. In D. Al-Dabass, editor, *Proceedings of the 17th European Simulation Multiconference (ESM 2003)*, pages 77–82, Delft, 2003. Society for Modelling and Simulation International and Arbeitsgemeinschaft Simulation, Society for Modelling and Simulation International.

[6] M. Lees, B. Logan, and G. Theodoropoulos. Time windows in multi-agent distributed simulation. In *Proceedings of the 5th EUROSIM Congress on Modelling and Simulation (EuroSim04)*, Paris, Sep 2004.

[7] M. Lees, B. Logan, and G. Theodoropoulos. Using access patterns to analyze the performance of optimistic synchronization algorithms in simulations of mas. *Transactions of the Society for Computer Simulation International*, 84:481–492, 2008.

[8] M. Lees, B. Logan, and G. Theodoropoulos. Analysing probabilistically constrained optimism. *Concurrency*

and Computation: Practice and Experience Journal, special issue on Distributed Simulation, Virtual Environments and Real Time Applications, 21:1467–1482, Aug 2009.

[9] B. Logan and G. Theodoropoulos. The distributed simulation of multi-agent systems. In *Proceedings of the IEEE*, volume 89, pages 174–186, Feb 2001.

[10] H. Mehl and S. Hammes. Shared variables in distributed simulation. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation (PADS'93)*, pages 68–75, 1993.

[11] P. Murgatroyd, V. Gaffney, B. Craenen, G. Theodoropoulos, V. Suryanarayanan, and J. Haldon. Logistics: A case of distributed agent-based simulation. In *Proceedings of the Distributed Simulation & Online Gaming Conference (DISIO 2010)*, Torremolinos, Spain, Mar 2010. ACM Digital Library.

[12] T. Oguara, D. Chen, G. Theodoropoulos, M. Lees, and B. Logan. An adaptive load management mechanism for distributed simulation of multi-agent systems. In *The 9-th IEEE International Symposium on Distributed Simulation and Real Time Applications (DSRT 2005)*, pages 179–186, Montreal, Oc. Canada, Oct 2005.

[13] V. Suryanarayanan, R. Minson, and G. Theodoropoulos. Synchronised range queries in distributed simulations of multi-agent systems. In *13th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2009)*, Singapore, Oct 2009.

[14] G. Theodoropoulos, R. Minson, R. Ewald, and M. Lees. Simulation engines for multi-agent systems. *Multi-Agent Systems: Simulation and Applications*, pages 77–108, 2007.

[15] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10:115–152, 1995.