Parallel & Distributed Simulation with DEUS

Michele Amoretti, Marco Picone, Stefano Bonelli, Francesco Zanichelli
Department of Information Engineering
Università degli Studi di Parma, Italy
michele.amoretti@unipr.it

ABSTRACT

This paper illustrates how parallel & distributed simulation capabilities have been introduced in DEUS, our tool for the analysis of complex systems. Its essential Java API provides basic interfaces and classes for modeling nodes, events and processes that characterize the structure and dynamics of any complex system. By supporting parallel & distributed simulations across multiple computing nodes, DEUS enables the analysis of large-scale complex systems. Its experimental validation has been obtained by means of the distributed simulation of a significant complex system, i.e. a large peer-to-peer network.

KEYWORDS: PDES tools, complex systems, peer-to-peer networks.

1. INTRODUCTION

Complex systems are dynamic systems composed of interconnected parts that as a whole exhibit one or more properties that could not be gathered from the properties of the individual parts. Examples of complex systems are found in nature, such as ant colonies, human economies, climate, nervous systems, cells and living things, including human beings, as well within modern energy and telecommunication infrastructures, ranging from networked embedded systems to large scale peer-to-peer architectures.

Quantitative approaches for analyzing the dynamics of complex systems have to consider a broad range of concepts, from analytical tools, statistical methods and computer simulations to distributed problem solving, learning and adaptation. Very often closed-form, analytical evaluation is not feasible, thus simulation remains the only viable evaluation methodology. Hence, performing a simulation means mimicking the occurrence of events over time, and recognizing their effects as represented by states of the modeled complex system. Future events occurrences induced by states must be scheduled (*i.e.* planned). In *continuous* simulation state changes occur continuously in time, while in *discrete* simulation the occurrence of an event is instantaneous and fixed to a selected point in time. Continuous simulation models can be converted into discrete models, which are more easily managed and thus most used. Depending on the characteristics of the system to be simulated, the reliability of the answer required, and many other factors, discrete simulation can be either *event driven* (in which time jumps from event to event), or *time driven* (in which time proceeds at a constant step) may be more appropriate. Continuous simulation refers to differential equation models, while discrete simulation refers to discrete time or discrete event models.

In this context, our research activity focuses on discrete event simulation [14] of large complex networked systems, that are characterized by events that are not guaranteed to occur at regular intervals, and by the lack of a bound on the time step (*i.e.* it should not be so small as to make the simulation run too long, nor so large as to make the number of events unmanageable). Examples are distributed computing systems based on the peer-to-peer paradigm, with randomly joining and leaving nodes, but also emergency rescue and crisis management scenarios, where rescuers do not arrive and leave at regular time intervals.

We are developing a general-purpose discrete event simulation environment, called DEUS [4]. With respect to our previous work [1], here we illustrate a new version of DEUS, supporting parallel & distributed simulation (PDES) based on a hybrid approach, that avoids violating the causality constraint among events scheduled on distributed queues. Indeed, we consider the latter as the most suitable strategy to maintain consistency among logical processes when communication among them are frequent, which is the case of distributed simulation of networked complex systems.

978-1-61284-383-4/11/$26.00 ©2011 IEEE

With respect to the sequential version, PDES-based DEUS allows to simulate larger complex systems, *e.g.* peer-to-peer networks with size increased by two orders of magnitude (from 10^4 to 10^6 nodes).

The paper is organized as follows. Section 2 illustrates the basic principles of our DEUS simulator. Section 3 describes how parallel & distributed simulation is effectively supported by the most recent release of DEUS. Section 4 shows the performance of the simulator with an event-based implementation of the Chord peer-to-peer overlay scheme. Section 5 discusses related work, and section 6 concludes the paper with a summary of the work done and a proposal for future work.

2. DEUS SIMULATION TOOL

Discrete event simulation (DES) works by maintaining a list of events sorted by their scheduled event times. Executing events results in new events being scheduled and inserted into the event list as well as events being deleted and removed from the event list. Such a fundamental principle is the basis of most DES tools, including our general-purpose simulation environment, called DEUS [1]. The essential Java API of DEUS, which can be downloaded from the project site [4], allows developers to implement (by subclassing):

- nodes (*i.e.* the parts which interact in a complex system, leading to emergent behaviors: humans, pets, cells, robots, intelligent agents, etc.)
- events (*e.g.* node births/deaths, interactions among nodes, interactions with the environment, logs, etc.)
- processes (either stochastic or deterministic, constraining the timeliness of events)

In DEUS, a node represents a system characterized by a set of possible states, whose transition functions may be implemented either in the source code of the events that can be associated to the node, or in the source code of the node itself.

The first solution is appropriate if we do not want to set in the node class definition all the possible node behaviors, but leads to events that must be specific for one or few kinds of nodes. For example, consider DEUS nodes representing computational Grid nodes, hosting computational and storage resources. We could implement different resource discovery events, each one with its own routing strategy. Such events take into account the structure of the simulated Grid node, while the latter is designed independently of the events that could affect it.

The second solution puts the whole specification of the node behavior on the node itself, and allows to implement events that potentially work on any kind of node. For example, consider two DEUS node classes Car and Boat, extending an abstract class Transport, with function startEngine. An event StartEngineEvent, either associated to a Car or to a Boat, would have a generic reference to a Transport object, and trigger startEngine on it.

3. PDES WITH DEUS

Parallel & distributed simulation (PDES) refers to the concurrent simulation of several model components, each one contributing to the simulation of a composite system model. Model simulators can be either geographically dispersed, or executed on each processor of a multiprocessor architecture. The objectives of PDES are manifold: increasing the speed, increasing the size of models, exploiting the greater graphics capability provided by specialized nodes, etc. Sequential simulators guarantee causality between events by generating a total ordering of events according to their scheduling times. Distributed simulators, however, are prone to causality violations. To avoid it, they must respect the causality constraint [6], stating that

> "a simulation of a model consisting of modular components which exclusively communicate through their modular input/output couplings obeys the causality constraint, if and only if each modular component simulator processes the events in nondecreasing timestamp order".

Several approaches to PDES have been developed in three decades of research. They are categorized as conservative, optimistic, or hybrid [5, 14]. Conservative solutions strictly avoid violating the causality constraint. Optimistic approaches temporarily violate it, but then detect and repair the violations. The hybrid solution is a compromise between the two other approaches. It weakens the conservative "block until safe-to-process" rule in a sense that if the time instant of the occurrence of an external event is in the time interval $[s, t]$, it allows progressing simulation until the forecasted next event instant $t' \in [s, t]$, but further progression only with controlled probability.

In the following we recall the conservative solution proposed by Chandy and Mistra [2]. Then we illustrate how it has been adapted in order to be implemented within DEUS,

becoming a hybrid strategy. Other approaches are under development, and will be illustrated in a future work.

3.1. Conservative Strategy

Suppose to have a simulation of a system composed by many nodes. The whole set of nodes is divided in subsets, and each subset is assigned to a *logical process (LP)* running on a different host. Thus, each LP is responsible for a group of nodes.

Sometimes one node being simulated on one host generates an event that is associated to another node being simulated on another host. Using the following notation for events:

$$e(sender, receiver, scheduled\_time) \quad (1)$$

we say that LP_1 generates $e(LP_1, LP_2, t_{12})$.

Nodes are instantiated and initialized during the simulation, by means of birth events. Each node n has an identifier $id(n)$. Each LP is responsible for a fraction of the identifier space. Node identifiers are randomly assigned to nodes, for which if a LP generates a node birth event, depending on the node identifier the event is inserted in the local queue or sent to the destination LP. If the list of available LPs is static and known in advance, this operation is trivial. If LPs can join and leave dynamically, it is necessary to manage the redistribution of the nodes. Here we consider the simplest case, *i.e.* we suppose the number of hosts and their responsibilities in respect to identifiers are fixed and known in advance by each LP.

Going back to distributed event scheduling, there are two possible situations: $t(LP_2) \leq t_{12}$ (which is good, e is put in the queue of LP_2), or $t(LP_2) > t_{12}$ (which is bad).

The general conservative solution states that, on the i-th logical process, next internal event $e(LP_i, LP_i, t_{ii})$ can be processed if and only if $t_{ii} < t_{ji} \forall j \neq i$, where t_{ji} is the scheduled time of last received event $e(LP_j, LP_i, t_{ji})$. Thus, a logical process blocks when it has not received events from all its influencing logical processes.

A deadlock may occur when there is no event present in one of the input queues of a logical process, for which it cannot consume events in the other queues, resulting blocked. The usual scheme to avoid deadlock is by using *null messages* to announce the absence of outputs for a certain period in the future [14]. When a logical process has an event for another logical process, say at time 10, it also sends a null-message to other logical processes announcing that it has advanced to time 10 and will not have an output prior to 10. Thus, other logical processes can process all the events received prior to 10.

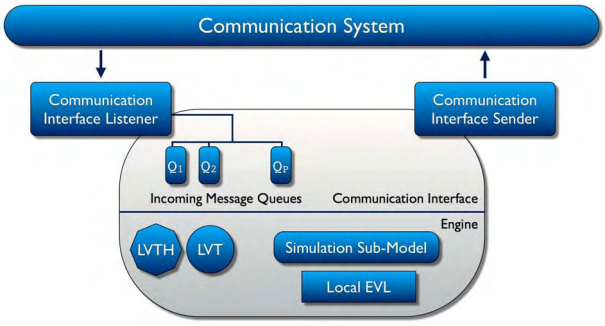

Figure 1. Internal Architecture of a Logical Process. Local EVL is the Queue of Events - Generated by the LP Itself, or by External Ones.

As almost every parallel & distributed simulation system, DEUS cannot adopt this approach, because when LP_i produces an event $e(LP_i, LP_j, t_{ij})$ it does not mean that the *virtual time* (VT) at LP_i is t_{ij}; in general it is $t_i \leq t_{ij}$.

3.2. Hybrid Strategy

Being DEUS a general-purpose simulation tool, it allows the analysis of systems where complexity arises from the number of nodes and from the huge communication exchange among them, as well as of systems composed by few partially interconnected elements exhibiting a complex internal behavior. Peer-to-peer networks are an example of the former kind in which each node potentially interacts with any other node and the simulation load is due to the network size and to the frequent data exchange. A representative example of the latter kind is given by a hardware system where a few statically connected modules cope with a high number of internal events while limited inter-module communication takes place. In the case of a P2P network, the simulation needs to allow each LP (whose architecture is illustrated in figure 1) to communicate with any other, thus yielding to a completely connected graph. For the other type of systems, specific communication links among LPs (representing the hardware modules) should be defined, thus improving the independence of each LP and providing a better exploitation of the system parallelism.

In order to put less constraints on the designer and to make simulation independent from the number of available LPs, the current choice is to maintain a fully connected graph among LPs. This allows the simulation of both kinds of aforementioned systems, although performance improvements cannot be achieved for partially connected complex

systems. Additionally, this design choice has been beneficial with respect to deadlock prevention. With a completely connected configuration deadlock prevention is guaranteed and derives from a structural property of the simulator in which a specific albeit sometimes constraining limitation has been introduced. Each communication between LPs in a completely connected graph can give rise to other communications in arbitrary future points of time and in unknown directions, thus any communication between LPs is seen as a point of synchronization. To guarantee that constraints on temporal causality are met, no LP can advance beyond the point in time of the next envisioned communication, even though, at execution time, it will not be influenced by any event message. In this way the simulation progress is favored since it does not need to take into account the actual interconnections among system components which are forcedly considered as independent.

Even load distribution over an arbitrary number of LPs is obtained by dividing the the total number of DEUS nodes and of initial events equally among the available LPs. Being the majority of events usually related to a specific DEUS node it is assumed that runtime generated events will presumably be distributed in the same way as nodes. Communications between nodes result into the creation of events, each associated to the destination node. In the case of a node executed on a remote LP, event scheduling cause an event message to be sent to the destination LP. When the received message is processed, the event is recreated and scheduled in the local event queue.

Initially, our plan was to tackle the design of the distributed version of DEUS following Chandy-Misra approach [2], which is based on *null messages* as a mean for synchronizing logical processes. Nevertheless, having a completely connected LP graph, we had to introduce a number of changes. The resulting parallel & distributed simulation approach is based on a barrier algorithm, whose main loop includes the following steps:

1. Lookahead computation/estimation: each LP examines its event queue looking for the first occurrence of an external communication; the virtual time of the related event is considered as the safe Local Virtual Time Horizon (LVTH), by which other LPs can process their events without violating temporal causality constraints;
2. Communication of the LVTH: as envisioned by the null message mechanism, each LP transmits its computed LVTH value to all other LPs.
3. Agreement on a common LVTH: after the second step each LP is aware of the time horizon computed by each other LP for which it assumes no communication will take place. Thus, the worst case horizon, *i.e.* the closest one, is obtained for the whole system[1].
4. Processing of events until the common LVTH: each LP retrieves and executes events from its queue up to the point of time corresponding to the LVTH. At this exact time, the LP (or more LPs if they all computed the same lookahead in the first phase) which obtained the shortest horizon will surely have processed events related to external communications.
5. End of the communication exchange: either in the case a LP has sent events to other LPs or otherwise, it is required to send a notification message to every other LP in order to guarantee that in the current iteration no other communication will be received.
6. Message queue check/inspection: each LP examines its inbound message queue, inserts any received event and waits for (if necessary) any notification of end of communication exchange (sent at previous phase).

After each loop iteration, message queues have been possibly modified and are examined to compute the new LVTH. With respect to the original Chandy-Misra protocol, it has been necessary to impose a progress based on constant time intervals, to guarantee the independence of load distribution from the specific simulation. Such kind of solution, called Conservative Time Windows has been proposed also in [10] and [11]. Thus, our solution can be considered to be hybrid, in between Chandy-Misra and Conservative Time Windows approaches.

In Chandy-Misra's strategy, both event messages and null messages contribute to define, thanks to their timestamp, the virtual time (VT) of the sender logical process, which is the guarantee for not receiving communications before that VT. In DEUS, as already stated, when LP_i generates $e(LP_i, LP_j, t_{ij})$, supposing $VT_i = t_i$, in general it means that the VT at LP_i is $t_i \leq t_{ij}$. Thus, the message that is sent at step 2 of the above algorithm does not include the information $VT_i = t_i$, but the worst-case estimate of next communication triggered by LP_i.

We recall the, if the execution of an event requires the interaction of the associated node with other nodes to retrieve information about their current status, new "communication" events must be scheduled. Even better, the best practice is to collect neighborhood information in advance, by scheduling the appropriate events. From a general viewpoint, it

[1] This mechanism based on a centralized barrier does not scale well. We plan to improve the efficiency of the algorithm by using a dissemination barrier, instead.

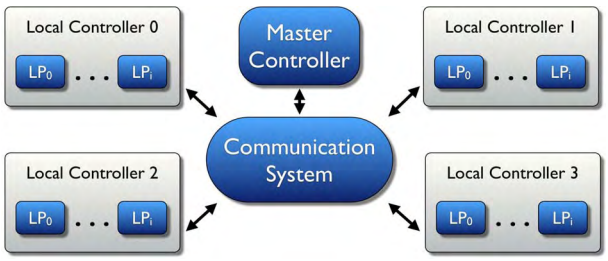

Figure 2. DEUS Components that Realize Distributed Simulations.

can be observed that the rarest and more foreseeable are the communications, the longest are the execution bursts.

The worst-case estimate of next communication can be very tight, if the problem does not allow to know with certainty that next communication will be performed in the far future. In general, the notion of *lookahead* in distributed discrete event simulation is the characteristic that a LP predicts future messages on the basis of messages it has already received. This lookahead can be the result of the incorporation of more problem-specific system knowledge into the model specifications, as well as due to the properties of the service time distributions. Simulation performance is directly related to the lookahead estimation.

Finally, to support distributed simulation in DEUS we had to provide it with a communication infrastructure that allow distributed instances of the simulation tool to exchange messages and to be coordinated by a centralized controller. Moreover, we introduced a mechanism for which the logs created by each LP are collected, at the end of each execution, and merged, to be processed by the statistical tools that were already available within DEUS. The resulting architecture is illustrated in figure 2. The *Master Controller (MC)* is the front-end of the simulation tool, allowing to set the configuration file describing the simulation, the *IP:port* of machines running *Local Controllers (LCs)*, as well as the total number of LPs that will be automatically distributed and assigned to LCs. The MC is also responsible for collecting log files from LCs at the end of the simulation process.

4. EXPERIMENTAL EVALUATION

As a benchmark, we implemented the distributed simulation of a Chord ring, *i.e.* a well-known peer-to-peer network based on the decentralized structured model (DSM). It is demonstrated that, with high probability, the number of nodes that must be contacted to find resource in an N-node network is $O(\log N)$ [13].

Our distributed simulation includes the construction of a large Chord ring and the propagation of several queries. The Chord ring is split in m equal slices, each one assigned to a different logical process (LPs). Message propagations over the Chord ring (represented by communications events among peers) may involve several LPs, although last hops are usually concentrated in the same slice.

The Master Controller (MC) has been executed on a virtual Windows Server machine (the MC is involved just for few seconds at the beginning and at the end of the simulations). Local Controllers (LCs) have been executed on a cluster consisting of four Linux machines, each one being equipped with two Intel Xeon E5504 2.00GHz (quadcore processor) and 16GB of RAM. Each LC ran on one node of the cluster, and created up to 8 LPs - one per core.

We measured the elapsed time, speedup (*i.e.* the ratio between the elapsed times of the sequential and parallel solutions) and efficiency (*i.e.* the ratio between the speedup and the number of LPs) versus the number of logical processes (from 1 to 32), for a Chord ring made of $2.5 \cdot 10^5$ nodes (figure 3), and for one made of 10^6 nodes (figure 4). In the first case, parallelization is not very convenient: the maximum speedup is 3.2, obtained with 8 LPs (very far from the theoretical limit, *i.e.* m, the number of LPs). The reason of these bad results is that, by increasing the number of LPs, the Chord ring (which is relatively small) becomes too much fragmented, for which message propagations involve too many LPs. Indeed, parallelization gets more convenient in the case of the Chord ring with 1 million nodes: the speedup matches the theoretical limit when $m = 2$ and $m = 4$, and is acceptable also with $m = 8$ (in this case, $S_m = 6.2$, that is not so far from 8).

5. RELATED WORK

CD++ [15] is a modelling environment that allows to define and execute modesl specified with the DEVS formalism (not to be confused with our DEUS tool) [14]. In CD++ is a simulation environment developed in C++, that directly manages DEVS models. CD++ is a sound tool and has also a GUI for the configuration and execution of the models. There is also a parallel implementation of the tool, called PCD++, based on optimistic approaches like Time Warp [8, 9]. Unfortunately, CD++ is not suitable for simulating large-scale complex systems with dynamically evolving structure, such as peer-to-peer networks, because it does not allow to include network construction and modification within simulated processes - the structure of the system

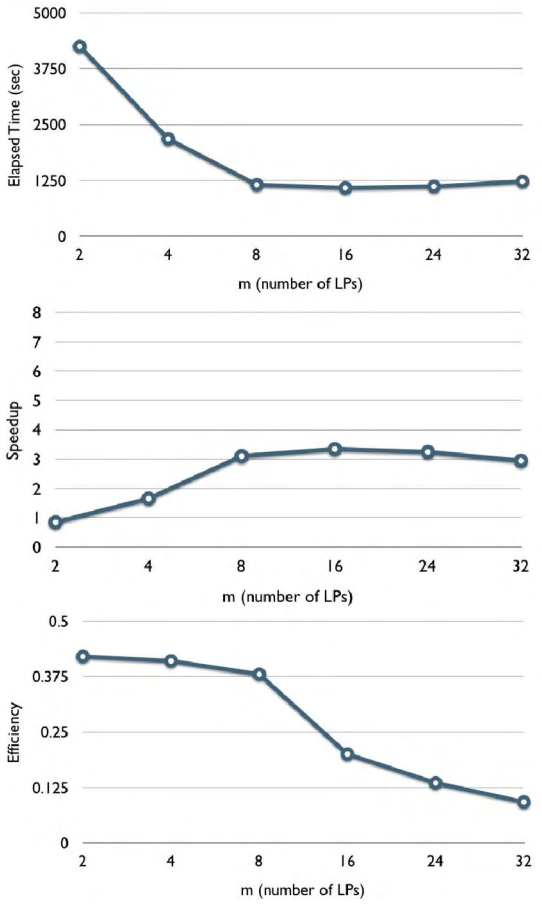

Figure 3. Elapsed Time, Speedup and Efficiency Versus the Number of Logical Processes, for a Simulated Chord Ring Made of $2.5 \cdot 10^5$ Nodes.

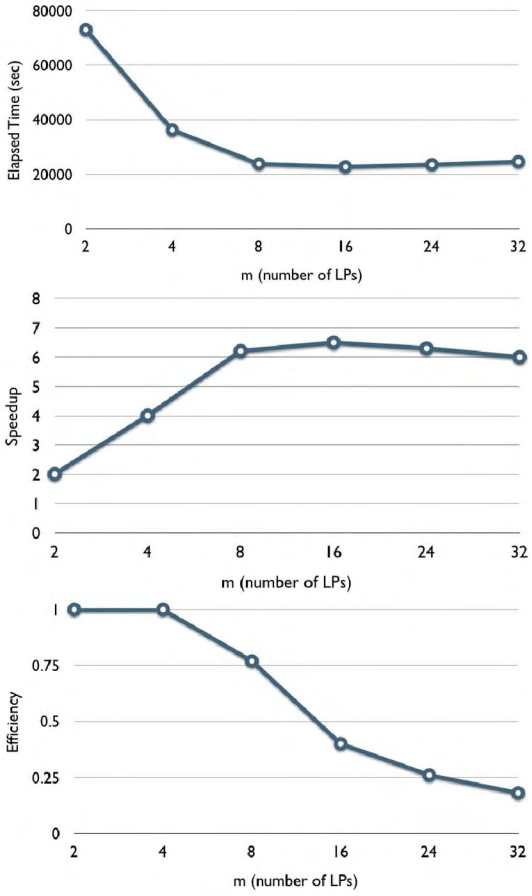

Figure 4. Elapsed Time, Speedup and Efficiency Versus the Number of Logical Processes, for a Simulated Chord Ring Made of 10^6 Nodes.

must be pre-defined before the actual simulation starts.

In [7], the authors present a simple method called micro-synchronization to exploit the parallelism inside each logical process (LP). Different from the previous work, such as lookahead accumulation and local time warp, they keep the traditional usage of lookahead among LPs unchanged, and however, they impose the relaxed sequential event scheduling inside each LP, which can indirectly improve the lookahead. The experimental evaluation of the proposed method shows that it can improve the performance of conservative parallel simulation of computer networks to some extent.

Cheon *et al.* propose to combine the parallel DEVS formalism [14] and the peer-to-peer paradigm, introducing a customized new DEVS protocol for distributed simulation in which logical processes solve the synchronization problem by themselves, without involving a coordinator [3]. The idea is highly appealing, but other than the developed prototype, there is no evidence of its application on complex and large-scale models.

More recently, another implementation of the DEVS formalism called DEVS/RMI system has been presented in [16], as a natively distributed simulation system based on standard implementation of DEVS. The DEVS/RMI system is also built to support auto-adaptive and reconfiguration of simulations during run-time. Because Java RMI supports the synchronization of local objects with remote ones, no additional simulation time management needs to be added when distributing the simulators to remote nodes. Several tests have proven that our distribution protocol based on XML is more simple, efficient and light of any Java RMI -based solution.

In [12], Seo *et al.* introduce DEVS/Grid, a distributed simulation framework allowing DEVS M&S activities over Grid computing infrastructures. The solution relies on existing

Grid management framework, such as Globus, and provides a number of interesting functionalities such as cost-based hierarchical model partitioning, dynamic coupling restructuring, automatic model deployment, remote simulation activation, self-communication setup, M&S name directory service. In future developments of parallel & distributed DEUS we will take into account DEVS/Grid facilities.

6. CONCLUSIONS

In this paper we have illustrated the implementation of a hybrid strategy for distributed simulations into our DEUS open source tool. We discussed the internal organization of distributed DEUS and the importance of lookahead estimate. The effectiveness of such enhancement has been illustrated by using DEUS to simulate a Chord ring with a large number of nodes.

As future work, we plan to introduce an automatic load balancer, as well as different distributed simulation strategies, such as optimistic and probabilistic ones, in order to confirm the general-purpose nature of DEUS. Moreover, we plan to perform new tests, considering other challenging problems related to the study of peer-to-peer networks (node churn, node mobility, etc.).

REFERENCES

[1] M. Amoretti, M. Agosti, F. Zanichelli, "DEUS: a Discrete Event Universal Simulator", 2nd ICST/ACM International Conference on Simulation Tools and Techniques (SIMUTools 2009), Roma, Italy, March 2009.

[2] K.M. Chandy, J. Mistra, "Distributed Simulation: A Case Study in Simulation and Design of Distributed Systems", *IEEE Trans. on Software Engineering*, Vol.5, No.5, pp.440-452, 1979.

[3] S. Cheon, C. Seo, S. Park, B. P. Zeigler, "Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Network System", *Military, Government, and Aerospace Simulation*, 2004.

[4] DSG, "DEUS: a simple tool for complex simulations", DEUS project [website], April 16, 2011, Available: http://code.google.com/p/deus/

[5] A. Ferscha, G. Chiola, "Self-Adaptive Logical Processes: the Probabilistic Distributed Simulation Protocol", 27th Annual Simulation Symposium, La Jolla, California, April 1994.

[6] R. M. Fujimoto, "Parallel Discrete Event Simulation", *Comm. ACM*, Vol.33, No.7, pp.30-53, 1990.

[7] S. Lin, X. Cheng, J. Lv, "Micro-synchronization in Conservative Parallel Network Simulations", 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation, Lake Placid, NY, USA, June 2009.

[8] Q. Liu, G. Wainer, "Lightweight Time Warp - A Novel Protocol for Parallel Optimistic Simulation of Large-Scale DEVS and Cell-DEVS Models", 12th IEEE International Symposium on Distributed Simulation and Real Time Applications (IEEE DS-RT 2008), Vancouver, BC, Canada, October 2008.

[9] Q. Liu, G. Wainer, "A Performance Evaluation of the Lightweight Time Warp Protocol in Optimistic Parallel Simulation of DEVS-based Environmental Models", 23rd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS'09), Lake Placid, NY, USA, June 2008.

[10] B.D. Lubachevsky, "Bounded lag distributed discrete event simulation", SCS Multiconference on Distributed Simulation, San Diego, California, USA, February 1988.

[11] D.M. Nicol, "Performance bounds on parallel self-initiating discrete event simulations", *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No.1, pp.24-50, January 1991.

[12] C. Seo, S. Park, B. Kim, S. Cheon, B.P. Zeigler, "Implementation of Distributed High-performance DEVS Simulation Framework in the Grid Computing Environment", High Performance Computing Symposium, Arlington, VA, USA, April 2004.

[13] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications", *IEEE/ACM Transactions on Networking*, Vol.11, No.1, 2003.

[14] B. P. Zeigler, H. Praehofer, T. G. Kim, THEORY OF MODELING AND SIMULATION, 2nd Edition, Academic Press, 2000.

[15] G. Wainer, "CD++: a toolkit to develop DEVS models", *Software - Practice and Experience*, Vol.32, No.13, pp.1-46, November 2002.

[16] M. Zhang, B.P. Zeigler, P. Hammonds, "DEVS/RMI-An Auto-Adaptive and Reconfigurable Distributed Simulation Environment for Engineering Studies", *International Test & Evaluation Association (ITEA) Journal of Test and Evaluation*, Vol.27, No.1, pp.49-60, March/April 2006.