NET CENTRIC MODELLING AND SIMULATION USING ACTORDEVS

Franco Cicirelli[(a)], Angelo Furfaro[(b)], Andrea Giordano[(c)], Libero Nigro[(d)]

[(a) (b) (c) (d)] Laboratorio di Ingegneria del Software (www.lis.deis.unical.it)
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria
87036 Rende (CS) – Italy

[(a)]f.cicirelli@deis.unical.it, [(b)]a.furfaro@deis.unical.it, [(c)]agiordano@deis.unical.it, [(d)]l.nigro@unical.it

ABSTRACT
The goal of the DEVS-World project is the development of a net-centric modelling and simulation (NCMS) infrastructure having the net as *the* computer, thus favouring different levels of interoperability among research groups operating world wide. This paper proposes an architecture based on web services for NCMS using ActorDEVS. ActorDEVS is a lean and efficient agent-based framework in Java supporting modelling of Parallel DEVS systems under both centralized and distributed simulation. ActorDEVS supports custom control engines. The paper discusses some architectural scenarios for wrapping ActorDEVS in the DEVS-World infrastructure, opening to interoperability with other DEVS or (possibly) non-DEVS systems. The proposal clearly separates model and simulation concerns. An entire model is partitioned among a number of simulation nodes with web services, in a case, which act as the transport layer for inter-node message exchanges. A global coordinator with a minimal interface of operations governs the "in-the-large" simulation aspects.

Keywords: M&S using the Internet, agent-based DEVS, web services, interoperability

1. INTRODUCTION

The DEVS-World project (DEVS-World 2007) aims at developing a world-wide standard platform for modelling and simulation (M&S), promoting collaborative research and experimentation in the engineering, i.e. design, evaluation, implementation, deployment and execution of complex, scalable, dynamic structure systems (Hu *et al.* 2005) belonging to diverse and significant problem domains like biology and bioinformatics, environment systems, traffic simulation etc.

The project has its strength in the use of DEVS (Zeigler *et al.* 2000) as the unifying M&S formalism and an exploitation of nowadays software technologies and middleware such as agents (Agha 1986, Wooldridge 2002, Cicirelli *et al.* 2007a) and services (Papazoglou and Georgakopulos 2003, Cicirelli *et al.* 2007c), which are a key for software interoperability. The main goal is enabling the exchange of both models and experiments among researchers and developers operating in academic or industry labs, thus favouring cooperation.

In this paper the ActorDEVS (Cicirelli *et al.* 2006, Cicirelli *et al.* 2007b, Cicirelli *et al.* 2008) framework is put under the perspective of DEVS-World in order to identify possible extensions and cooperation scenarios. ActorDEVS (see Fig. 1) is a lean and efficient agent-based framework in Java supporting modelling of Parallel DEVS systems under both centralized and distributed simulation. The approach clearly separates modelling from simulation concerns.

Both simulation and real-time execution modes are supported for model continuity which rests on the possibility of changing the control engine and ultimately the time notion regulating the evolution of the application. The approach is *control-centric*, in the sense that it allows customizing the control machine (see Fig. 1) which offers basic scheduling and dispatching message services to actor components.

Key factors underlying ActorDEVS are the adoption of actors (Agha 1986, Cicirelli *et al.* 2007d) as programming in-the-small building blocks, and of theatres (Cicirelli *et al.* 2007a) as programming in-the-large execution loci (see Fig. 1). Adopted actors are thread-less reactive objects which encapsulate an internal data state (which include *acquaintances*, i.e. known actors which can be contacted by messages), have a behaviour patterned as a finite state machine, and communicate to one another by asynchronous message passing. Actors can migrate dynamically from a theatre to another for reconfiguration purposes.

ActorDEVS is supported by a minimal API in Java. Typed input/output ports are mapped on to actor messages. Configuration operations correspond to updating receiver information in output ports, also during the runtime. More in general, changing actor's acquaintance network, a concept which is often referred to as link mobility, is a natural way to achieve model structure dynamism (Cicirelli *et al.* 2007a, Cicirelli *et al.* 2007d, Cicirelli *et al.* 2008). Good execution performance is ensured by having a DEVS model is flattened from the point of view of the simulation engine.

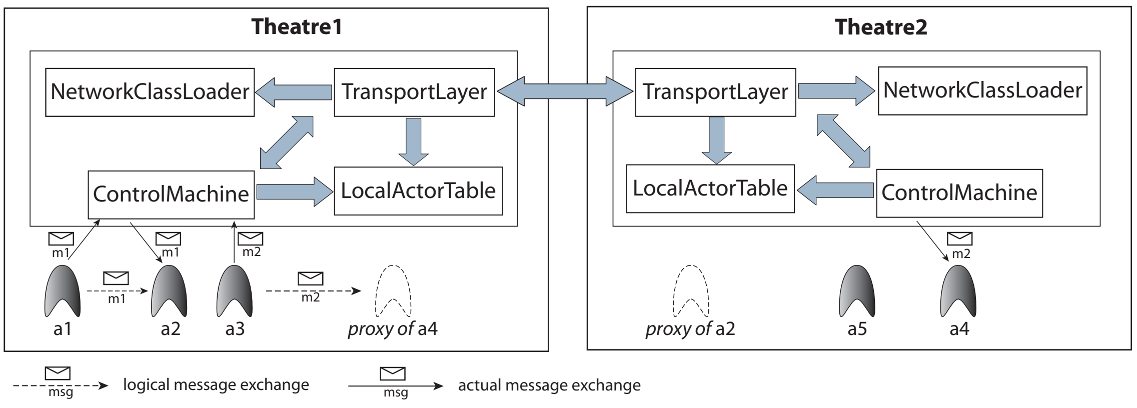

Figure 1: Actor/Theatre architecture for ActorDEVS

The paper is structured as follows. Subsequent section introduces the DEVS-World project and its objectives and key features. After that, a description is provided of how Theatre/ActorDEVS architecture can be extended for supporting NCMS. Last section discusses some issues relevant to a pragmatic use of the resultant architecture. Finally, future and on-going work is summarized.

2. AN OVERVIEW OF DEVS-WORLD

Novel in the DEVS-World project is the definition of a development methodology for supporting world-scale distributed *open* systems of systems M&S (DEVS-World 2007). Openness is a fundamental property which expands along different directions with different levels of integration and interoperability.

A first level of integration is relevant to model interoperability. Many different implementations of DEVS simulators currently exist, and usually each of them uses a built-in modelling language often tied to a specific programming language like Java or C++. To cope with this problem, specific conversion tools capable of translating a DEVS model from a language to another can be realized. A more general solution would be that of adopting emerging DEVS standard language such as DEVSML (DEVS-World 2007).

Another direction of integration concerns interoperability at architectural level. In (DEVS-World 2007) but also in (Mittal *et al.* 2008) the proposed world-wide architecture is aimed at harmonizing heterogeneous models based on special-case DEVS tools, programming languages and engines, through the use of Web Services and SOAP dependent messages and other DEVS concepts (ports, simulators, coordinator etc.). Web Services are viewed as a world-wide *glue* enabling interoperation through DEVS/SOA mechanisms, with WSDL used for web services interface specification.

Besides standardization of models and simulation infrastructure, the definition of a standard simulation protocol (Xiaolin and Zeigler 2008) is mandatory. The protocol (see Fig. 2) describes how a DEVS model should be simulated and how service/simulation engines should coordinate each other. Such a protocol opens also to a scenario in which both DEVS and non-DEVS simulators may (possibly) participate in a simulation.

CoreSimulatorInterface (see Fig. 2) is the common interface to simulators. The term "core" means "essential" in that as long as a simulator implements this interface, it can participate in a simulation driven by a DEVS coordinator. In the case of DEVS-simulators, the *CoupledSimulatorInterface* is considered. This interface extends the core interface by providing other functionalities e.g. for adding/removing couplings among DEVS models.

CoordinatorInterface must be implemented by the coordinator. The coordinator is in charge of synchronizing the activities of the various simulators guiding them through the simulation control cycle. Basic phases of the simulation cycle are shown in Table 1.

In handling simulation of hierarchical coupled models, a coordinator orchestrates a set of controlled simulators within it and, at the same time, can participate with peers in a coupled model above it. To allow such downward/upward facing interfaces, the *CoupledCoordinatorInterface* is introduced which extends both the *CoordinatorInterface* and the *CoupledSimulatorInterface*.

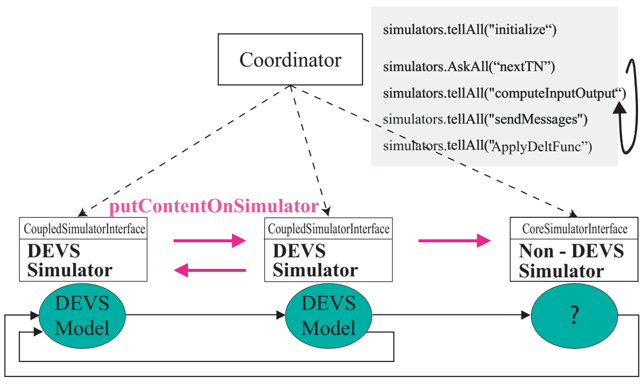
Coordinator
simulators.tellAll("initialize")
simulators.AskAll("nextTN")
simulators.tellAll("computeInputOutput")
simulators.tellAll("sendMessages")
simulators.tellAll("ApplyDeltFunc")
putContentOnSimulator
CoupledSimulatorInterface
DEVS Simulator
CoupledSimulatorInterface
DEVS Simulator
CoreSimulatorInterface
Non - DEVS Simulator
DEVS Model
DEVS Model
?

Figure 2: Simulation protocol in a federation of DEVS and non-DEVS simulators

Table 1: Simulation cycle phases

Step	Description
nextTN	the coordinator requests that each simulator sends its time of next event and takes the minimum of the returned values to obtain the global time of next event
computeInputOutput	each simulator applies its *computeInputOutput* method to produce/gather an output that consists of a collection of *Contents* (i.e. port/value pairs)
sendMessages	each simulator partitions its output into messages intended for recipient simulators and sends these messages to these recipient simulators. Sending a message implies to call the recipient's *putContentOnSimulator* for any target simulator
applyDeltFunc	each simulator executes its *ApplyDeltFunc* method which computes the combined effect of the received messages and internal scheduling on its state. A side effect is in producing the time horizon gives back at the nextTN

3. WRAPPING ACTORDEVS IN DEVS-WORLD

This section highlights a service-based approach extending the Theatre/ActorDEVS architecture in order to meet requirements of DEVS-World project. Provided extensions support architectural interoperability among heterogeneous DEVS simulators. The approach adopts previously described DEVS simulation protocol. At the moment, interoperability at modelling language level is not addressed. Each DEVS model is assumed to be implemented as a Java class complying with the ActorDEVS API (Cicirelli *et al.* 2008).

A *Coordinator* is introduced in order to coordinate the evolution of the overall simulation and it is in charge of implementing the DEVS simulation cycle (see Table 1). A *Configurator* makes it possible to configure the whole simulation system and start execution. An UML class diagram of system components is reported in Fig. 3.

The *Theatre* component and the *Configurator* are not exclusive of DEVS simulations, they are common to all actor-based applications. The *Coordinator* (see Fig. 4), instead, is tightly related to DEVS-World prospective.

A *DEVSControlMachine* has been purposely developed in order to work in pair with the coordinator and be compliant with the DEVS simulation protocol. This control machine implements a *CoupledSimulatorInterface*–like (see Fig. 2) and behaves as a DEVS simulator.

With respect to the approach proposed in (Xiaolin and Zeigler 2008) the *Coordinator* is only concerned with the execution of the DEVS simulation cycle. In particular it does not manage coupling information among DEVS models. Such information is directly handled at simulator level. In addition, being in a net-centric context, the *Coordinator* must wait until all outgoing messages, i.e. inter-simulator messages, are received by recipient simulators before proceeding to the applyDeltFunc phase (see Table 1). This is ensured by `Chek` messages (see Fig. 4) sent by simulators to the coordinator. Toward this, the `setCoordinator` method was added to *CoreSimulator* (see Fig. 3). `Chek` messages are actually generated at the end of sendMessages phase and after external messages are received.

CoupledSimulator interface (Fig. 3), which does not introduce further methods, extends both *CoreSimulator* and *Coupled* interfaces. This is to guarantee a clear separation of concerns among configuration (i.e. coupling management addressed by the *Coupled* interface) and simulation aspects (simulation protocol management addressed by the *CoreSimulator* interface).

It is worthy noting, finally, that a *DEVSControlMachine* is in charge of handling the simulation needs of all the models allocated to the same theatre. In other words, each theatre has one simulator instead of having one simulator for every distinct atomic model.

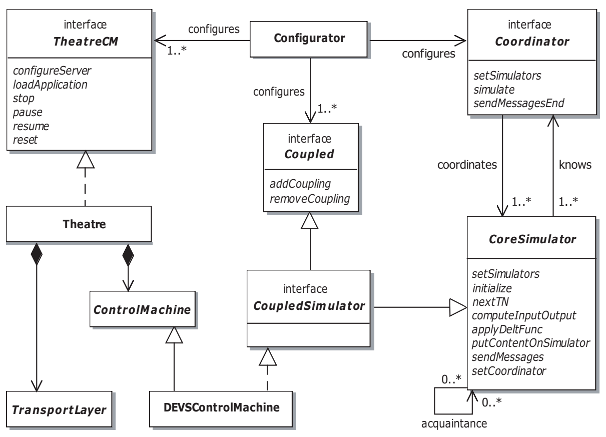

Figure 3: Class diagram of system components

```
public interface Coordinator{
   void setSimulators(SimulatorInfo[] si)throws Exception;
   void simulate(long simulationTime)throws Exception;
   void sendMessagesEnd(Check check)throws Exception;
}
```

Figure 4: Coordinator interface

In order to support the NCMS vision, a whole Theatre/ActorDEVS system, which can span from a single atomic model to a complex coupled model, is made usable through Web Services. Each system component is made available as a Web Service by means of specific objects called Wrappers. Client-side interactions are instead mediated by means of specific Proxy objects. It is worthy of note that in a service oriented architecture the roles of client and provider are not strictly defined, being possible for a same node to act as client or provider on the basis of the required/offered functionalities.

Wrappers and Proxies are transparently used. As a consequence, would e.g. Java RMI be used in place of Web-Services based protocols, only Wrappers and Proxies would be accordingly changed. Fig. 5 shows the architecture of a resultant Theatre/ActorDEVS system.

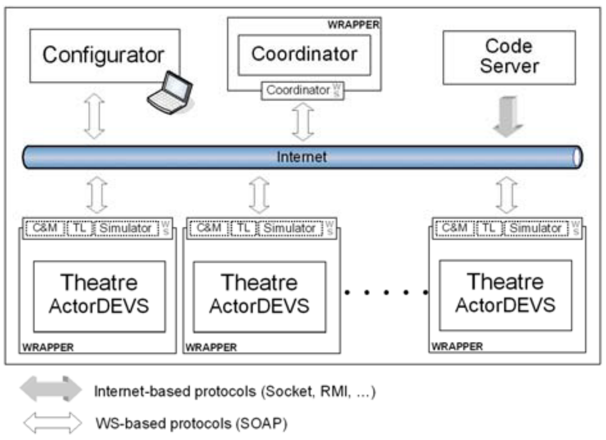

Figure 5: Architecture of a Theatre/ActorDEVS system

A *Code Server* is shared among theatres and it is used as a remote Java-class repository from which download the actor-based application to execute, i.e. in this case the DEVS models to simulate. Configuring and starting a simulation consists of four steps. The first step is devoted to setting-up the Theatre nodes by specifying the control machine, the transport layer to use and the code server IP address.

This is accomplished by exploiting the Configuration and Management Web Service (see the C&M-WS in Fig. 5). After the control machine is instantiated its functionality is made available as a Web Service which is automatically published (see the Simulator-WS in Fig. 5). The *DEVSControlMachine* oversees message exchange with other simulators. As a consequence, the transport layer (see the TL-WS in Fig. 5) in this scenario is used only to manage inter-theatre control messages.

The second step consists in assigning to each Theatre the DEVS model(s) to simulate. A single model may correspond to an atomic or to a coupled DEVS component. The Java class name of each model requires to be specified along with the parameters possibly required by its constructor. This step is carried out by exploiting the C&M-WS and completes when models get assigned to target theatres, i.e. downloaded from the code server and instantiated.

The third step consists in establishing the necessary bindings among coordinator and simulator services (i.e. acquaintance relationships). In particular, a *CoordinatorInfo* object is provided to each simulator and a list of all *SimulatorInfo* objects, relevant to simulators involved in the federation, is furnished to each simulator and to the coordinator. An info object contains the name of the service and the relevant service endpoint address which is necessary to contact and use it. As stated above, each simulator has to know the coordinator in order to communicate information about the state of the current sendMessages phase (see Table 1).

The fourth step consists in defining couplings among deployed models in order to build the entire simulation model. This is achieved by invoking the method *addCoupling* onto simulators. Coupling information mainly contains a couple of names, identifying the two ports to be connected. The first name is relevant to an output port of a DEVS component local to the simulator. The second name is relevant to an input port of a DEVS component which can be either local to the simulator or residing on a remote simulator. In the latter case, the name of the remote simulator is provided along with coupling information.

A naming policy is required to distinguish ports belonging to different instances of the same model. In particular, full name of a port is assumed to be specified in the form *modelInstanceName.portName*.

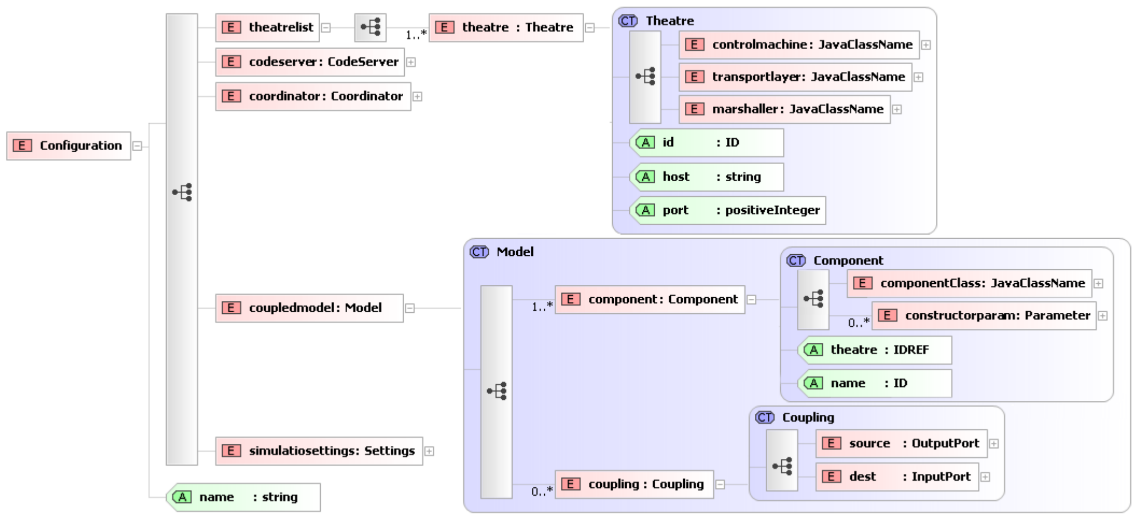

Figure 6: XML schema of the configuration files

At runtime, remote couplings get actualized by means of the so called *RelayPort* objects. Making a remote coupling implies linking an output port of a DEVS component to a relay port which, in turn, is logically connected to a remote input port. All of this makes the DEVS component unaware of network partitioning.

All data needed during configuration steps are contained in an XML file whose schema is reported in Fig. 6. In the current prototype system implementation, the `Settings` type is used only to contain the simulation time info. The `CodeServer` and `Coordinator` types contain information required to contact the relevant components on the web (e.g. service name, host, port). Other types are self-explanatory.

At configuration end, the Configurator may launch the simulation by calling the simulate method on the Coordinator which in turn triggers into execution the simulation control loop.

4. VARIABLE STRUCTURE SYSTEM EXAMPLE

The achieved implementation of WS-based Theatre/ActorDEVS architecture was tested by modelling and simulation of a variable structure system based on server relocation (Cicirelli *et al.* 2008). The modelled system consists of a collection (closed pipeline) of interconnected node components (see Fig. 7).

Each node receives from its environment a stream of jobs, stores them in a buffer (of unbounded size) and ultimately processes them using a number of server components. A system is assumed to work with a fixed number of servers. Servers cannot be dynamically generated because they model physical computing resources. However, a high loaded node can ask for a server to its neighbours. A dispatcher component in a node is in charge of handling the server relocation issues. Main difference between the model as handled in (Cicirelli *et al.* 2008) and here, consists in the achievement of structure dynamism.

In (Cicirelli *et al.* 2008), server components migrate from a node to another as mobile agents. In the scenario of this paper, though, servers do not migrate but port objects are created/destroyed dynamically in order to contact servers.

Asking for a server may return a server port through which a dispatcher can submit a job to a server allocated on a different node. As a consequence, server relocation is achieved by changing the number of servers a node can contact to process its jobs. Different strategies of server relocation can be considered (see later).

Fig. 7 depicts a three node system, together with input/output ports and connectors. Each node can direct useful statistical data to an external *Statistics* (*transducer*) component connected to the *StatOut* output port. When used, the *OverloadGenerator* can inject jobs randomly to any node.

Fig. 8 shows the internal structure of a node. Inter-node ports serve to send/receive an ask to/from a neighbour for a server (*ask-OUT?*, *askIN*), to send/receive a server to/from a neighbour (*moveIN?*, *moveOUT?*, *moveIN*), or to send/receive back a no longer useful server (*sendBackOUT?*, *sendBackIN*). Fig. 7 shows *delegate* connections (represented by using dashed lines) within a coupled node. The shadowed *TimerToken* component in Fig. 8 is required only by some relocation protocols.

A high loaded node, that is a node with a pending job but without idle servers, asks for a server port to its neighbours. When the *Dispatcher* of a node receives a request for a server, it honours the request with a server port if at least one idle server is available. Otherwise the request is ignored. If no server ports are obtained, a node asks again for a server port after a certain time delay. Three particular strategies (Cicirelli *et al.* 2008) were considered about the way a node can handle external utilizable servers.

On-demand strategy - A node which achieves an external server, views it as an own server. Therefore, the protocol freely distributes server ports among nodes on a on-demand basis. It can be anticipated that this strategy makes it possible for nodes to behave in a selfish-way, possibly leading to an unbalanced distribution of server use.

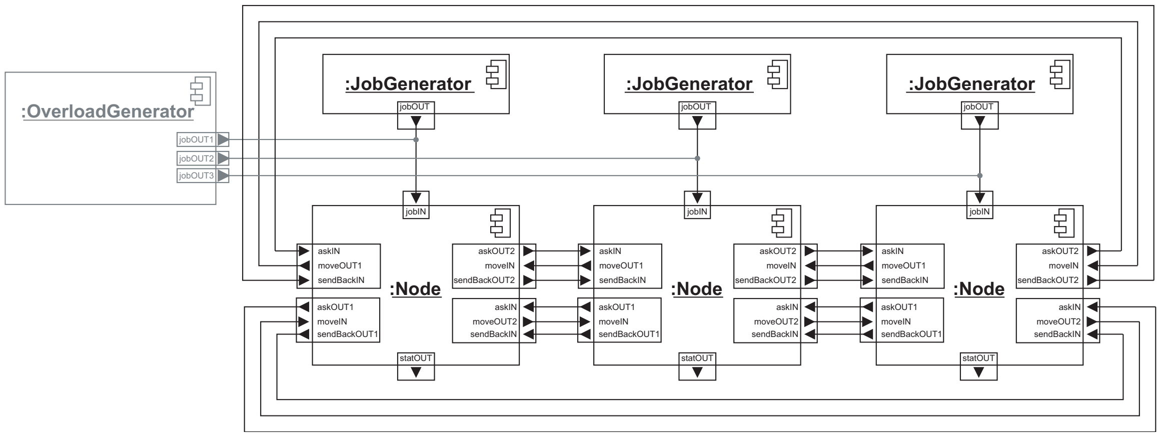

Figure 7: A ring of three nodes

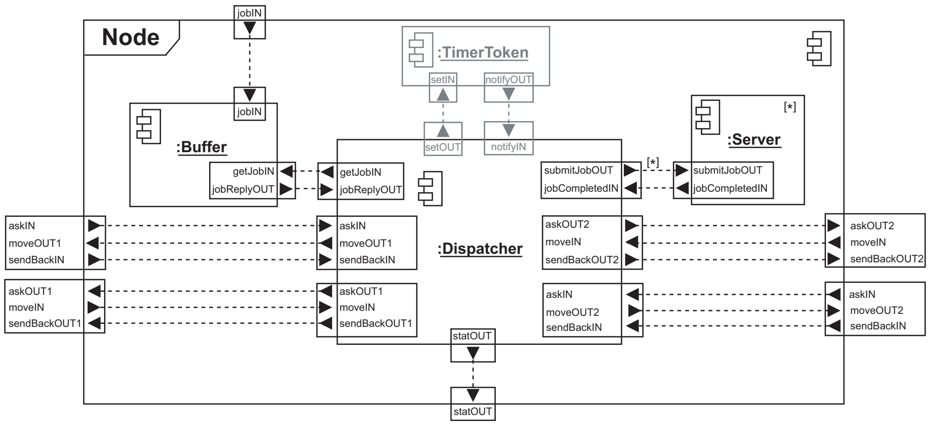

Figure 8: Internal structure of a node

Debt strategy - A debt concept for server allocation is introduced. A node which receives a server port from a neighbour, annotates the identification of the furnishing node. As soon as the *Dispatcher* of a debtor node has no pending job but has at least one idle server, it tries to exhaust its debits by anticipating restitution of some server ports to its creditor nodes. Intuitively, the protocol attempts to avoid non uniform utilization of servers.

Token passing strategy - One server port is used as a token which circulates upon the closed pipeline. A node receiving the token-server can use it if has a pending job but has no available local server. Otherwise, or after token usage, the token is forwarded to the next node in the ring. The strategy tries to anticipate a server request. A node which receives the token as well as server ports coming from neighbours, uses the token and sends back the other server ports.

5. CONFIGURATION, DEPLOYMENT AND SIMULATION

Some simulation experiments concerning the server relocation model described in the previous section were carried out by using two Theatre/ActorDEVS systems allocated on two Win platforms.

Another Win platform was used to host the Coordinator, the Code Server and the Configurator. The experiments were directed to study the effects of overloads starting from an equilibrium situation. Simulation parameters which, under either On-Demand or Debt strategy, ensure the buffers size or equivalently the mean delay time of jobs is definitely constant and of a low value are as follows.

The job interarrival time is in the interval [2,4], the job size (which indicates the time needed to process the job) belongs to the interval [8,15]. The time delay a *Node* waits between two consecutive asks for a server was set to 1 time unit. The number of servers initially allocated to each node is 4. Starting from the equilibrium, the *OverloadGenerator* (see Fig. 7) is capable of injecting each generated job to a randomly chosen node.

To respond to the overload, one additional server was introduced, whose management ultimately depends on the adopted strategy(ies).

For instance, under On-demand or Debt strategies the extra server is initially assigned to a given node. In the Token passing strategy, instead, the extra server (its port) circulates in the pipeline ring. In this case, to avoid Zeno behaviours, the token which reaches the node where it was last used, is forced to wait one time unit before starting the next round.

The job mean delay time (that is the time which elapses between the instant in time a job is received by Buffer and the subsequent time the job gets assigned to a server) was measured by the Statistics components. The investigated strategies for responding to overload were: Debt & Token, On-demand & Token, On-demand alone.

The DEVS models relevant to *Node*, *JobGenerator*, *OverloadGenerator* and *Statistics* were deployed to the Code Server. A number of *Node*s, varying from one to five, along with the relevant instances of *JobGenerator*s were assigned to each Theatre. The *OverloadGenerator* and the *Statistics* were allocated on a single Theatre. The simulation time limit was set to $t_{END}=10^5$.

Different system configurations were actualized by specifying different configuration files. An excerpt of such a file is reported in Fig. 9. The configuration is relevant to a relocation system model made up of two *Node*s allocated to two theatres. Only the Debt strategy is considered.

Coupling information, common to all the configuration files, is used to build up the overall simulation model. In particular:

- each *JobGenerator* was coupled with the relevant *Node*
- each *Node* was coupled with its neighbors in the closed pipeline
- the *OverloadGenerator* was coupled with all the *Node*s
- each *Node* was coupled with the *Statistics*.

```
<?xml version="1.0" encoding="utf-8"?>
<Configuration name="RelocationServers"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="./TheatreDEVS.xsd">
<theatrelist>
 <theatre id="PERSEUS8000" host="perseus" port="8000">
  <controlmachine name="theatre.DEVSControlMachine"/>
  <transportlayer name="theatre.transport.WSTransport"/>
  <marshaller name="theatre.marshaler.ByteArrayMrshlr"/>
 </theatre>
 <theatre id="HYDRA8000" host="hydra" port="8000">
  ...
 </theatre>
</theatrelist>
<codeserver url="http://orion:8989"/>
<coordinator name="Coordinator" host="orion" port="8080"/>
<coupledmodel>
 <component name="Node1" theatre="PERSEUS8000">
  <componentClass name="relocation.Node"/>
  <!-- number of servers -->
  <constructorparam type="java.lang.Long" value="4"/>
  <!-- token disabled -->
  <constructorparam type="java.lang.Boolean" value="false"/>
  <!-- debt enabled -->
  <constructorparam type="java.lang.Boolean" value="true" />
 </component>
 <component name="Node2" theatre="HYDRA8000">
  <componentClass name="relocation.Node"/>
  ...
 </component>
 <component name="OverloadGenerator" theatre="PERSEUS8000">
  <componentClass name="relocation.OverloadGenerator" />
 </component>
 ...
 <coupling>
  <source theatre="PERSEUS8000" port="Node1.sendBackOut2"/>
  <dest theatre="HYDRA8000" port="Node2.sendBackIn1"/>
 </coupling>
 <coupling>
  <source theatre="PERSEUS8000" port="Node1.askOut2"/>
  <dest theatre="HYDRA8000" port="Node2.askIn"/>
 </coupling>
 <coupling>
 ...
</coupledmodel>
<simulatiosettings>
  <simulationtime>100000</simulationtime>
</simulatiosettings>
</Configuration>
```

Figure 9: An excerpt of a configuration file

Coupling information dictates system topology at configuration time. At runtime, on the basis of the adopted strategy, a *Node* may dynamically change the servers it actually contacts without resorting to the add/remove coupling mechanism.

Simulation experiments (see Fig. 10) indicate that the combination of Debt & Token strategies minimizes the job mean delay time when compared to the other strategies.

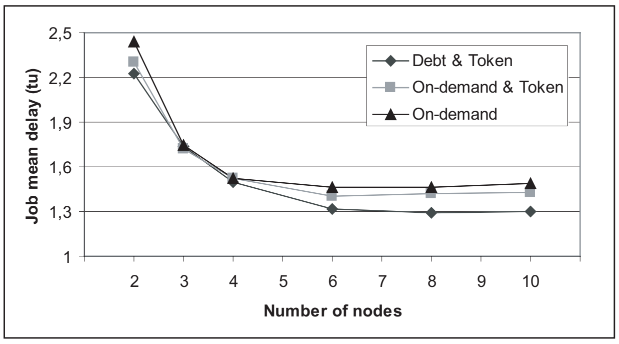

Figure 10: Job mean delay time vs. number of nodes

6. CONCLUSIONS

A prototype version of the Theatre/ActorDEVS architecture based on Web Services was realized and tested. The implementation relies on Java technology. In particular, the SOAP engine Axis (Axis website) is used for managing WS related aspects. The following are some points which deserve some discussion within the community of DEVS-World.

- The DEVS simulator protocol appears “too much synchronous” for a networked context. Many interactions among the simulation-protocol participants are required for each simulation step independently from the complexity of the simulated model. A systematic exploitation of a kind of “lookahead” could alleviate the problem. By exploiting lookahead the coordinator could give a granted time to each simulator allowing a more independent evolution of local simulation.
- Another (obvious) issue concerns simulation performance achievable by the use of WSs. This is not only tied to the use of verbose XML for SOAP messaging but mainly to the management of network connections. Simulation experiments confirmed that network resources (connections) of operating system may be wasted considerably during simulation and need in general careful control.

On-going work is directed at:

- improving the Configurator component by providing a friendly GUI for visual system configuration, model composition, deployment and simulation control
- replacing Axis by other Web Service infrastructure e.g. related to latest J2EE
- introducing a model repository service, enabling model reuse and sharing
- adopting standard formalisms like DEVSML for supporting DEVS modelling
- favouring model and experiments interchange by developing translation tools allowing model transformation from a high-level implementation-independent formulation into the terms of a specific DEVS setting (e.g. ActorDEVS and Java) and vice versa
- experimenting with Theatre/ActorDEVS architecture in an heterogeneous environment where diverse DEVS simulators have to cooperate
- developing tools for visual modelling.

REFERENCES

Agha, G., 1986. *Actors: A model for concurrent computation in distributed systems*. Cambridge, MIT Press.

Axis website. Available from: http://ws.apache.org/axis/index.html. [Accessed May 2008].

Cicirelli, F., Furfaro, A., Giordano, A., Nigro, L., 2007a. An agent infrastructure for distributed simulations over HLA and a case study using Unmanned Aerial Vehicles. *Proceedings of 40th Annual Simulation Symposium*, IEEE Computer Society Press, pp. 231-238, March, Norfolk (VA).

Cicirelli, F., Furfaro, A., and Nigro, L., 2006. A DEVS M&S framework based on Java and actors. *Proceedings of 2nd European Modelling and Simulation Symposium (EMSS 2006)*, pp. 337-342.

Cicirelli, F., Furfaro, A., and Nigro, L., 2007b. Conflict management in PDEVS: an experience in modelling and simulation of time Petri nets. *Proceedings of Summer Computer Simulation Conference (SCSC'07)*, pp. 349-356.

Cicirelli, F., Furfaro, A., and Nigro, L., 2007c. Integration and interoperability between Jini services and Web Services. *Proceedings of IEEE Int. Conf. on Services Computing (SCC'07)*, pp. 278-285, July.

Cicirelli, F., Furfaro, A., and Nigro, L., 2008. Actor-based Simulation of PDEVS Systems over HLA. *Proceedings of 41st Annual Simulation Symposium (ANSS'08),* pp. 229-236, April, Ottawa, Canada.

Cicirelli, F., Furfaro, A., Nigro, L., and Pupo, F., 2007d. A component-based architecture for modelling and simulation of adaptive complex systems. *Proceedings of 21st European Conference on Modelling and Simulation (ECMS'07d)*, 4-6 June, Prague.

DEVS World, 2007. DEVS_WORLD: A platform for developing advanced discrete-event simulation at worldwide scale. Internal document.

Hu, X., Zeigler, B.P., and Mittal, S., 2005. Variable structure in DEVS component-based modelling and simulation. *Simulation*, 81(2), 91-102.

Hu, X., and Zeigler, B.P, 2004. Model continuity to support software development for distributed robotic systems: A team formation example. *J. of Intelligent and Robotic Systems*, 39(1), 71-87.

Mittal, S., Zeigler, B.P., Martin, J.L.R., Sahin, F., and Jamshidi, M., 2008. Modeling and simulation for systems of systems engineering. In: *System of Systems – Innovations for the 21st Century*, Wiley (in press).

Papazoglou, M.P., and Georgakopulos, D., 2003. Service Oriented Computing. *Communications of the ACM*, 46(10), 25-28.

Xiaolin, H., and Zeigler, B.P., 2008. A Proposed DEVS Standard: Model and Simulator Interfaces, Simulator Protocol, Internal document.

Yu, Y.H., and Wainer, G., 2007. eCD++: an engine for executing DEVS models in embedded platforms. *Proceedings of SCS Summer Simulation Multiconference*, pp. 323-330.

Zeigler, B.P., Praehofer, H., and Kim, T.G., 2000. *Theory of modeling and simulation*. 2nd edition, New York, NY, Academic Press.

Wooldridge, M., 2002. *An introduction to multi-agent systems*. John Wiley & Sons, Ltd.