Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

RMIS: MIDDLEWARE FOR TRANSPARENT OBJECT-ORIENTED MODELING IN MULTI-SIMULATOR SYSTEMS

Niels A. Lang

Dept. of Decision and Information Sciences
Burg. Oudlaan 50, PO Box 1738
RSM Erasmus University Rotterdam
3000DR, Rotterdam, THE NETHERLANDS

Peter H.M. Jacobs
Alexander Verbraeck

Systems Engineering Group
Faculty of Technology, Policy and Management
Delft University of Technology
Jaffalaan 5, 2628BX, Delft, THE NETHERLANDS

ABSTRACT

A system of middleware services is proposed to realize transparent Object Oriented (OO) modeling in multi-simulator systems. Shortcomings of Remote Procedure Call schemes in a multi-simulator setting are identified. The use of simulation time synchronization services is suggested instead.

The resulting middleware system supports both asynchronous and synchronous interaction. Remote Method Invocation Scheduling (RMIS) is introduced as the main enabling mechanism. Special naming services are introduced to generate required middleware objects. The middleware system is fully transparent to model objects, which may thus remain deployment neutral. Current results and directions for further research are presented.

1 INTRODUCTION

Object Oriented (OO) methods are widely adopted for software development. This is for example demonstrated by the emphasis placed on OO features of major software platforms like Microsoft's .NET and Sun Microsystem's Java. The advantages offered by OO methods include enhanced reusability, scalability and maintainability (Martin and Odell 1998, Booch 1991).

The main principles by which OO realizes these advantages include encapsulation and design independence (Martin and Odell 1998). The first principle advocates a strict separation of (external) interface and (internal) implementation. As a result, the same interface may be implemented by a wide variety of components, which realizes a high level of maintainability. The second principle advocates the development of technology independent, or deployment neutral, OO components. This allows easy component reuse in heterogenous deployment settings.

The relevance of such principles can be observed in today's increasing attention for web services (Papazoglou and Geogakopoulos 2003) and grid computing (Foster et al. 2001). Both technologies intend to harness the computational potential of heterogenous, distributed environments by developing modular, deployment neutral components.

It is clear that the typical benefits of OO software development are desirable for simulation modeling as well. In fact, it is perhaps no coincidence that the first OO language SIMULA (Dahl and Nygaard 1966) was intended for simulation. Besides realizing reusable, maintainable model components, OO simulation modeling may also open up the potential of distributed computing environments.

In Jacobs et al. (2002), we presented the Distributed Simulation Object Library (DSOL) as an architecture for OO simulation modeling. DSOL was designed to enable reusable, scalable modeling in a distributed environment.

For DSOL, Discrete Event System Specification (DEVS, Zeigler et al. (2000)) was proposed as the simulation paradigm for scalability reasons. In combination with OO models, the OO-DEVS paradigm results. In a OO-DEVS simulation system, the model consists of *objects* and, following OO practice, all interactions take the form of *method invocations*. Such method invocations are either *scheduled*, in which case they are delegated to a simulator, or called *directly*. DSOL fully supports simulation in a distributed setting, however, it was limited to single-simulator systems. In a DEVS setting, this implies that the simulation process is executed as a single, sequential process (Zeigler et al. 2000).

In this paper, we explore the application of DSOL (i.e. the OO-DEVS paradigm) for multi-simulator systems. Multi-simulator simulation may be beneficial for a number of reasons (Fujimoto 1999):

- Models and simulators may be deliberately dispersed at a number of geographical locations. For example, different parts of a supply chain model

may be developed and controlled by different actors (Zeigler et al. 2000, Kuhl et al. 1999).
- For certain domains (e.g. logistics) the computational resources required within components (e.g. simulation of a number of individual production facilities) may outweigh those required for inter-component interaction (e.g. a model describing flows between production facilities). Such decoupled models are naturally deployed as a number of independent, parallel simulation processes, only requiring modest interaction.
- Multi-simulator systems may fully employ the potential for parallel computing offered by grid computing. Application of distributed simulations on the grid are therefore actively researched (Fitzgibbons et al. 2004, Iskra et al. 2002).
- Currently, developments in hardware appear to head in the direction of multi-threaded and multi-core processors. While originally intended for the server market, this development may eventually reach the 'ordinary' desktop as well (Spracklen and Abraham 2005). Multi-simulator systems may fully employ the potential for parallel computing offered by such single-box hardware.

We found that OO interaction (i.e. method invocations) in a multi-simulator OO-DEVS setting is impeded by the need for simulator synchronization. We also found that current tools for distributed simulation do not fully support OO-style interactions (i.e. method invocation). We therefore set out to develop a system of middleware services to realize OO interaction in distributed, multi-simulator systems. In line with the objectives of OO modeling, we furthermore require that this middleware preserves the deployment neutrality of OO models. In other words, the developed middleware mechanisms should be fully transparent to the model objects involved.

The paper proceeds as follows. First, the problem of multi-simulator OO interaction is presented in more detail. After that, relevant types of middleware services are presented. Remote Method Invocation Scheduling (RMIS) is introduced as a main enabling middleware mechanism. Subsequently, the use of RMIS to realize OO interaction is demonstrated. Specifically, the application to asynchronous and synchronous interaction is presented. The preservation of deployment neutrality is illustrated. Finally, the paper presents conclusions and directions for further research.

2 OBJECT ORIENTED MODELING IN MULTI-SIMULATOR SYSTEMS

To obtain a clearer view on the issues regarding OO interaction in multi-simulator systems, we will first compare it with single simulator systems. After that, previous work and requirements are discussed.

2.1 Single Simulator OO Models

In a single-simulator setting, all model objects can be considered time synchronized, as they all use the same simulator to discover simulation time. This also holds true in case the model is in fact distributed over multiple runtime environments. This situation is illustrated in Figure 1, which illustrates the mechanisms and functions involved in a single simulator distributed OO model. The legend for this and subsequent figures is provided in Figure 2.

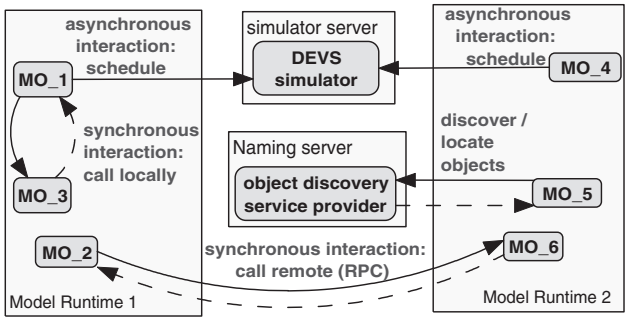


Figure 1: Remote Interaction in a Single Simulator Distributed OO Model

Figure 1 shows a model consisting of several model objects (e.g. MO_1, MO_2). Furthermore it presents two service types facilitating the object model: a DEVS simulator and a provider of object discovery services. The latter allows the construction of the model as a loosely coupled object system, in which the actual objects are only resolved during runtime. The first facilitates time scheduled asynchronous object communication.

Asynchronous object communication is defined in Booch et al. (1999) as *an object invoking an operation on another object where it does not expect immediate result*. In a simulation context, this implies that a result value is either never delivered, or delivered at a possibly later moment in simulation time. Delivery takes place by passing the result value as an argument in a new call on the initiating object.

Analogously, *synchronous* object communication can be defined as *an object invoking an operation on another object where it DOES expect immediate result*. In a simulation context, this implies that invocation and the return of the result value are to happen at the same simulation time. In fact, the initiating object is not allowed to perform any other tasks before the return value has been received, since otherwise parallel state modifying processes (in both calling and called object) could violate the deterministic state change sequence required in a single DEVS simulator setting. In a single simulator setting, synchronous object communication can simply be implemented by direct

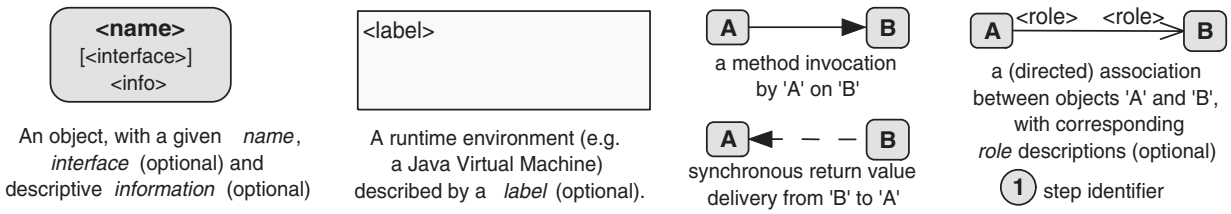


Figure 2: Conceptual Diagram Notation

method invocation. This invocation can either be local (i.e. between objects within a single runtime environment) or remote (i.e. between objects residing in different runtime environments). For the latter, several Remote Procedure Call (RPC) mechanisms are available, such as Java's Remote Method Invocation (RMI) (Arnold et al. 2000), CORBA (Object Management Group 1995) or the Simple Object Access Protocol (SOAP) (Box et al. 2000). As indicated before, since all objects use the same simulator to determine simulation time, direct method invocation, whether local or remote, will not cause causality violations.

However, although RPC has been originally developed as a paradigm for simple and transparent remote communication (Menascé 2005, Birrell and Nelson 1984, Birrell 1992), we find that RPC frameworks used in practice often fail to provide this transparency. For example, Java's Remote Method Invocation (RMI) framework enforces the use of special exception declarations in all RMI enabled interfaces. As this renders RMI enabled interface incompatible with non-RMI interfaces (as in Java the exception declarations are part of the interface signature), we argue that this violates the principle of deployment neutral model objects. In order to arrive at truly deployment neutral model objects, middleware should hide the details of the RPC mechanisms employed for the model objects involved.

2.2 Multi-simulator OO Models

Figure 3 presents the situation of an OO model deployed in a distributed manner using multiple simulators.

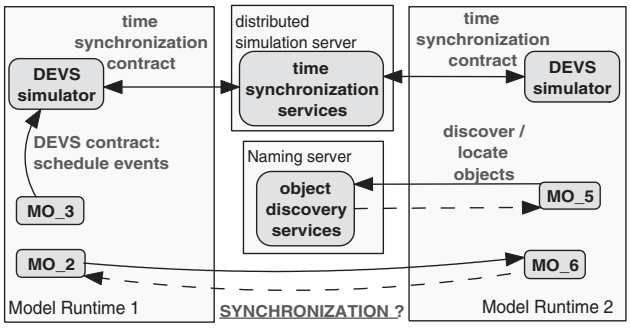


Figure 3: Unsafe Remote Interaction in a Multi-simulator Distributed OO Model

First, we note that, in contrast with the single simulator situation, all model objects have access to a local simulator. Interactions between these objects and their local simulator are in potential faster than in the single simulator situation, where model objects always need an RPC connection to contact their simulator.

Secondly, time synchronization services were added. The need for such services to prevent causality violations is well described in the distributed simulation literature (Fujimoto 1999).

Finally, however, we note that direct, simple RPC interaction between distributed model objects in a multi-simulator setting will violate time synchronization requirements. This is illustrated in Figure 3 by the interaction between MO_2 and MO_6. At the moment of invocation, MO_2 can observe its own simulation time. However, it has no guarantee whatsoever that MO_6 will observe the same simulation time. This can only be guaranteed when proper simulation time synchronization has taken place between the two simulators before the remote invocation. However, such functionality is not provided by standard RPC mechanisms.

2.3 Towards Middleware Supporting Transparent Multi-simulator OO Modeling

In the previous section we demonstrated that in distributed settings, there is a lack of support for transparent remote interaction between different Object Oriented simulation models. In the single simulator setting, we found that the direct use of common RPC mechanisms impacts the object model, since RPC specific interface elements are required. In the multi-simulator setting, we moreover found that the the direct use of RPC will violate time synchronization requirements. We conclude that to realize transparent Object-Oriented modeling in multi-simulator settings, the following functionality is required:

- Supporting remote versions of OO interaction primitives: scheduled asynchronous and synchronous method invocations. As the use of remote interactions must not violate the validity of the multi-simulator system, the remote protocol must ensure proper inter-model time synchronization.
- To allow model reuse in a variety of deployment settings, the details of the remote interaction func-

tionality should be hidden from the model objects involved. Specific deployment details should be part of the experimental frame (which is by definition experiment specific) rather than the model.

The fields of distributed simulation and software development have developed a substantive body of knowledge on distributed simulation and OO respectively. However, to our knowledge no conclusive work has been presented on this paper's subject, which requires integrated results from both fields. We summarize some related efforts in the following paragraphs.

2.3.1 The High Level Architecture

The High Level Architecture (HLA) (Kuhl et al. 1999) is generally regarded as the main reference architecture for distributed simulation (Boer and Verbraeck 2002). An evolved version has recently been adopted as IEEE standard 1516 (IEEE 2000). Main services provided by HLA control the management of federates, simulation time, objects and data distribution.

Although the HLA includes support for objects, HLA objects are not fully Object Oriented (Kuhl et al. 1999, p. 29). The most important omission, in the context of multi-simulator OO modeling, is the lack of support for object *methods*. This prevents the definition of a model in terms of *services*, transparently implemented by multiple flavors of objects.

2.3.2 IDSim

The Interoperable Distributed Simulation (IDSIM, Fitzgibbons et al. (2004)) framework was developed as a domain-independent, extensible framework for distributed simulation. The framework, built upon the Open Grid Services Infrastructure (OGSI), allows automatic and controlled deployment of distributed simulation models on arbitrary grid configurations, thus realizing a high level of model deployment neutrality. Main services provided by IDSim control the management of federates, simulation time and data distribution.

As we understand, IDSim's current support for remote model interaction is based on user-customizable *events*. Other than HLA, IDSim supports both push-type and pull-type delivery methods. However, method invocation style interactions appear yet to be unsupported. As IDSim is explicitly designed to support extensions, we expect that the mechanisms proposed in this paper could easily be incorporated in the IDSim framework.

2.3.3 Distributed Simulation Objects

In 1997, Heim already proposed a framework for distributed OO simulation (Heim 1997) which closely resembles the perspective adopted in this paper. Recognizing the need for model reuse, maintenance and integration, he departs from the notion of a model as an *association of distributed model-objects*. He suggested the Common Object Request Broker Architecture (CORBA) as the remote inter-object infrastructure. The proposed framework features interface generators, able to generate remote interaction helper objects for simple model objects, thus preserving the deployment neutrality of the original models. Finally, the framework introduced the use of an open directory protocol (LDAP) to standardize remote model object discovery.

However, issues regarding inter-model coordination (i.e. synchronization) were not resolved. It is specifically on these issues that this paper intends to make a contribution.

2.3.4 Remote Procedure Calls

Extensive literature is available on the concept of RPC. A hidden assumption in RPC literature is that remote objects are allowed to interact at any arbitrary moment in time. However, as illustrated before, this assumption is not true in a multi-simulator setting.

Although earlier research results provide important partial solutions for realizing transparent multi-simulator OO modeling, we did not find a framework meeting all the earlier mentioned requirements. We therefore set out to develop such a framework and an accompanying proof of concept to realize a deployment neutral implementation. In the next sections, the proposed middleware architecture is presented. First, the architecture's components and functions are introduced. After that, the application of the architecture for asynchronous interaction, deployment independent referencing and synchronous interaction are provided. Finally, first results for a proof of concept are demonstrated. The paper is concluded by discussing the current results and future research.

3 MIDDLEWARE ENABLING MULTI-SIMULATOR OO MODELS

This section presents the components which constitute the proposed middleware for multi-simulator OO models. They cover three functions already introduced in Figure 3: simulation, object discovery and time-synchronization. Finally, a new middleware component (Remote Method Invocation Scheduling) is introduced, which enables time synchronized object interaction following the requirements stated in section 2.3.

3.1 Discrete Event System Specification

We consider Discrete Event System Specification (DEVS) to be a useful simulation formalism to simulate object systems for a number of reasons. First, as demonstrated by Zeigler (Zeigler et al. 2000), DEVS is a fundamental formalism able to embed other ones. As such, DEVS is a fundamental formalism which may serve as the basis for multi-formalism simulation systems. Second, we argued in Jacobs et al. (2002) that, compared to thread-based OO simulation systems, DEVS based simulations scale further and offer more potential for distributed deployment.

Conceptually, a DEVS simulator executes events sequentially in strict ascending time order. The simulator exposes event (un)scheduling services to the model. During the event execution phase, the model may generate and schedule new events, or unschedule previously scheduled events. We argued in Jacobs et al. (2002) that in a pure OO model, the only relevant events take the form of *scheduled method invocations*. Such an event specifies at a minimum the target object, the method to be invoked, the simulation time at which invocation should take place and the invocation arguments. In our view, only the ability to *schedule* method invocations distinguishes simulation model objects from 'ordinary' software objects.

As an implementation for the OO-DEVS formalism, we used the open source Java-based Distributed Simulation Object Library (DSOL, Jacobs et al. (2002), Lang et al. (2003)). Since in an OO-DEVS system all scheduled inter-object interaction is managed by a simulator, this simulator is the natural object to extend with custom event processing functionality.

3.2 Dynamic Object Discovery Services

Naming systems are a fundamental part of distributed system development (Silva et al. 1998). In our view, the use of naming systems also offers significant benefits in the development of OO models, such as the possibility for object referencing by name and to provide a natural layer to perform object transformations in a deployment neutral way. In the context of distributed OO modeling, a naming system preferably has the following properties. First, we like it to be applicable for arbitrary name spaces, e.g. to support different URL schemes. Second, we favor an open standard, allowing a wide variety of implementations. For example, specific implementations may need to be distributed, or in-memory, or perform specific middleware transformations. Finally, the naming system should support update notifications, as monitoring the lifetime of model objects is a basic need for simulation modeling. In the proof of concept presented in this paper we applied the Java Naming and Directory Interface (JNDI) framework. As a set of high-level interfaces supporting naming systems of different capabilities, JNDI meets all our requirements. Default JNDI providers are available to access file, RMI, LDAP and CORBA based namespaces.

3.3 Time Management Services

In order to realize correct multi-simulator behavior, simulators need to synchronize event execution to prevent causality violations. In this work, we applied the High Level Architecture (HLA, (Kuhl et al. 1999) now also IEEE 1516 (IEEE 2000)) to realize this synchronization. In terms of the HLA, the simulators participate as *federates* in a *federation*. Federates make use of HLA's services via a single instance of the RunTime Infrastructure (RTI). In the context of this work, we only employed a subset of HLA services, namely the ones supporting initialization, interaction relay and conservative synchronization (see Fujimoto (1999)). As HLA's object model is not fully object oriented, we decided not to use HLA's object management services. Instead, objects may be discovered using the naming services introduced previously. In the following, we briefly discuss our use of initialization and interaction services.

During the initialization phase, a special management federate supervises proper initialization of the multi-simulator system, using HLA's synchronization services.

During execution, simulators use *interactions* to communicate simulation time-sensitive events. In HLA, an interaction is a pre-specified message of a given type with a given number of named attributes. An interaction type may extend another interaction type. As we intend to use interactions for time-sensitive information, we use HLA's capability to send the interactions in a *reliable* fashion, in Time Stamp Order (TSO). As we use conservative synchronization, HLA guarantees that an interaction thus sent by a federate with timestamp T will be delivered to receiving federates at a time $t \leq T$, with t denoting the simulation time at the receiving federate. In other words, such interactions will never be received in a federate's past.

In the next section, we introduce how this time-stamped interaction mechanism can be used to implement time-safe remote method invocations.

3.4 Remote Method Invocation Scheduling

As indicated previously, we regard the main event in an OO simulation model to be the scheduled method invocation. However, when attempting to schedule a method invocation on a remote object, one encounters the following challenges. First, the remote object resides by definition in a *remote* runtime. For 'ordinary' objects, i.e. ones not prepared for a remote protocol such as RMI or CORBA, remote references cannot be obtained. Second, method invocations on the remote object are managed by a remote simulator. The management of the invocation event should therefore

be transferred to the remote simulator. In a conservative synchronization setting, this must at least have been performed before the remote simulator has passed the event's invocation time.

We propose to combine naming and HLA services to meet both challenges. To solve the first challenge, the remote target object should be identified by a reference (i.e. an object providing a name), rather than by a direct handle. The reference should be resolvable by the naming system in use.

To meet the second challenge, we introduced a special HLA interaction type: **InteractionRoot.DSOLEvtInteraction**, with attributes *evt* and *targetFED*. The *evt* attribute can hold a network streamable version of the method invocation event. The *targetFED* attribute identifies the destination federate. Under a conservative synchronization policy, a method invocation event wrapped as an interaction will be transported *in time* by the RTI. Using this interaction, a simulator can *relay* a simulation event to another simulator, thus delegating its execution.

The originating simulator will discover the need for relaying the event by inspecting the event's target property. If the target's name identifies a context other than the local one, the event is automatically relayed to the simulator managing that context. Using the local context, the receiving simulator is able to resolve the target's reference into a handle.

This basic mechanism, which we will hereafter refer to as Remote Method Invocation Scheduling (RMIS) provides the backbone middleware mechanism for realizing transparent inter-model object interaction in multi-simulator settings. We note that RMIS could also be implemented on simulator synchronization and messaging services other than HLA.

In the next sections, the application of RMIS for realizing transparent remote object interaction is discussed. Asynchronous interaction, transparent referencing and synchronous interaction are discussed respectively.

4 TIME-SYNCED ASYNCHRONOUS REMOTE OBJECT INTERACTION

Supporting time synchronized asynchronous remote object interaction proved to be a rather direct application of RMIS. In this section, the steps involved in a successful asynchronous remote interaction are introduced. However, for illustrative purposes we will first introduce a simple example model. This model will also be referred to in subsequent sections.

4.1 An Example Model: Distributed Workers

Imagine an organization with several workers, distributed over several facilities. These workers are encouraged to perform load balancing: when they receive many jobs, they may delegate the job processing to other workers, possibly at remote facilities. Figure 4 presents a highly simplified model for analyzing this situation.

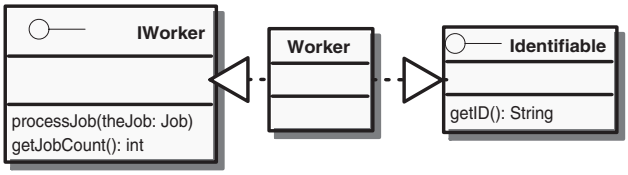


Figure 4: A Simple Distributed Model

It is expected that most job processing and delegating will take place within facilities. It is therefore decided to model a number of facilities in parallel, using one simulator per facility. Facilities are numbered fac_1, fac_2, etc. Workers can push jobs to others by invoking *processJob*. To perform load balancing, the number of jobs already allocated can be returned by calling *getJobCount*. In addition, the *Identifiable* interface allows middleware to discover the name under which an object is bound in a context.

4.2 Asynchronous Interaction Using RMIS

Figure 5 illustrates the steps involved in scheduling a job transfer by *worker_1* in facility 1 to the remote object *worker_2*. In terms of the model presented in Figure 4 this implies scheduling the method *processJob*, with a Job object as argument, on the remote worker.

The complete process involves several phases. In each phase, different middleware functions are used.

In the first phase (step 1), the remote object *worker_2* is resolved by the naming system, which hides the details of remote reference resolution. In the second phase (step 2), the method invocation is scheduled on the local simulator: a standard OO model mechanism.

During the third phase (steps 3 and 4), the simulator notes the need to relay the method invocation event. It then prepares and sends an interaction encapsulating the event.

In the fourth phase (steps 5, 6 and 7), the remote simulator processes the received interaction. The encapsulated event is examined and its target (*worker_2*) resolved. Then, the event is scheduled on its local event list. During the fifth phase (steps 8 and 9) the HLA managed time advancement process (a sequence of time requests and grants) causes the remote simulator to reach the time of the remotely scheduled event. Finally, this event is executed as usual (step 10).

We note that in this example, *worker_1* performs the same steps as it would do when scheduling on a local object: it resolves the target object using the naming system and subsequently schedules the desired method invocation using the local simulator. The details of the remote interaction are transparently performed by middleware services

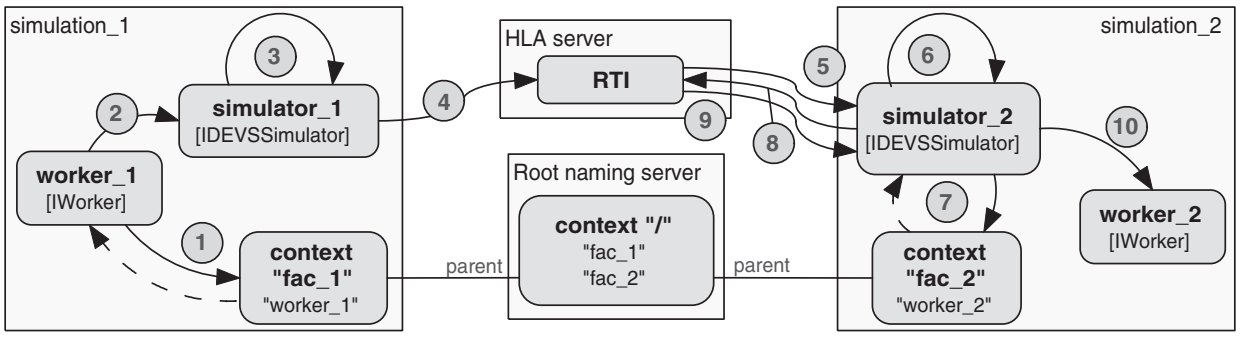


Figure 5: Asynchronous Interaction Using RMIS

in the simulator, the naming system and the HLA. RMIS thus enables asynchronous remote object interaction while preserving deployment neutrality of the model objects.

5 REFERENCE GENERATION

In the previous example, the context returned a *reference* rather than a handle. In that example, where no methods were *directly* invoked on the reference, this posed no problem. However, in general the requesting model object (here: *worker_1* expects to retrieve an object implementing a known model interface from the naming system (here: an object implementing *IWorker*). Deployment neutrality demands that this expectation is not violated when retrieved objects happen to be situated in a remote context.

We therefore propose middleware to generate references dynamically, such that the generated references expose the expected model interfaces. This is effectively an application of the OO design pattern known as the bridge pattern (Lang et al. 2001). In the remainder of this section we briefly discuss the process of reference generation and the required content of generated references.

5.1 Middleware for Reference Generation

As indicated previously, a naming system provides a natural layer to conduct object transformation for middleware purposes. We therefore implemented the reference generation functionality as part of a custom naming system.

The reference generating process is managed by a custom context implementation, the so-called HLA context. First, we note that only objects retrieved from *remote* context need to be replaced by references. For that purpose, only remote requests (i.e. originating from the remote root context) are handled by an HLA context.

The moment an HLA context receives a lookup request, it first resolves the requested object via its source context. Then, it inspects whether or not the result should be replaced by a reference. In the current implementation, this decision is based on the interfaces implemented by the result value.

If the result is to be replaced by a reference, the reference class is determined. In this implementation, the reference class equals the original class with suffix **HLAWrapper**. If no such class is found, the HLA context generates the class. Using introspection, a Java source file is generated which includes the necessary interfaces and methods. This source is subsequently compiled.

After locating the proper reference class, a new instance is initialized using the result value's name in the namespace. Finally, the reference is returned.

5.2 The Content of Transparent References

The contents of a reference are determined by several requirements. First, it should be recognizable as a reference. Hence, it must implement an interface identifying it as such. Second, it should expose the expected model interfaces and implement them properly (i.e. using RMIS). Finally, the reference is to be streamed over the network. In Java this implies the need to identify it as Serializable.

Figure 6 demonstrates an example implementation for the wrapper of an instance of the class *Worker*.

First, we developed a generic wrapper (of class *ReferenceWrapper*) which can execute methods by using the RMIS mechanism described previously. This service is exposed as a protected method *callMethod*. It also takes care of eventual wrapping of the invoked method's arguments. The generic wrapper also implements the interface *IHLAObjectReference* which in our implementation identifies it as a remote reference to simulators.

The specific reference class (*WorkerHLAWrapper*) extends the generic wrapper class. Within this specific class, methods of the exposed model interfaces are implemented to delegate invocation to the generic wrapper's method *callMethod*. To a receiving object, the resulting reference thus implements the expected *IWorker* interface. Direct method invocations will be translated into the proper RMIS calls.

We conclude that as long as model objects only depend on other model object's *interfaces*, this reference generating middleware will fully hide the details of remote interaction

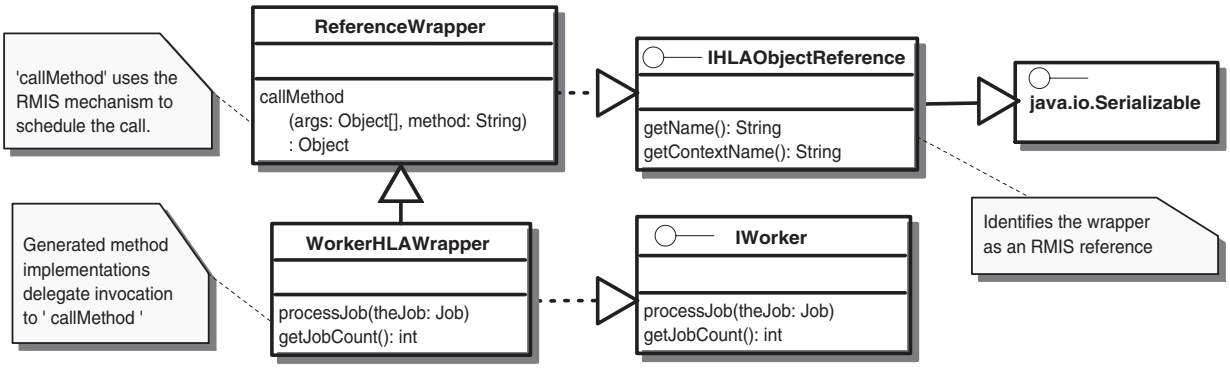


Figure 6: An RMIS Wrapper for the IWorker Model Interface

for the model. As such, it preserves the model's deployment neutrality.

6 TIME-SYNCED SYNCHRONOUS REMOTE OBJECT INTERACTION

The complement of asynchronous, scheduled interaction is synchronous interaction, by direct method invocation. Compared to scheduled interaction, synchronous interaction introduces the following additional challenges. First, it implies that methods are directly called on objects as resolved by the naming system. Second, it implies that a remotely generated result value needs to be returned to the calling object. Moreover, between invocation and reception of the result value, the calling object is to be blocked.

The first challenge is already solved by the use of transparent references (see section 5). To meet the second challenge, we refined the RMIS mechanism in a number of ways. First, the generated wrapper object now not only invokes the RMIS mechanism, but also registers on a remote channel and blocks operations. Secondly, a refined simulation event object not only invokes a method on execution, but also relays the return value on a remote channel. As a result, the waiting wrapper object will eventually be notified of the result value, unblock operations and return the value to the invoking model object. An example of the resulting synchronous interaction process is illustrated in Figure 7.

Compared to asynchronous interaction (see section 4), several differences are introduced. First, the calling model object directly invokes a method on the object as resolved by the naming system (step 1). This implies that execution is to take place instantaneous, at *simulator_1*'s current simulation time. To prevent past event scheduling, *simulator_2* is then by consequence not allowed to run ahead of *simulator_1*. We therefore conclude that under a conservative synchronization protocol, support for synchronous interaction implies the use of zero lookahead.

Second, to enable the relay of the remotely generated result value, the wrapper sets up a remote channel (step 2). In this implementation, the management of remote channels is performed by a *ReturnSessionManager*. Currently, this object is implemented as a Java RMI server, but any other RPC implementation could have been used instead.

Finally, a special event class (**HLAEvent**) was introduced, which allows events to relay return values over the remote channel (steps 7 and 8). As the class implements common event interfaces, no simulator changes are needed.

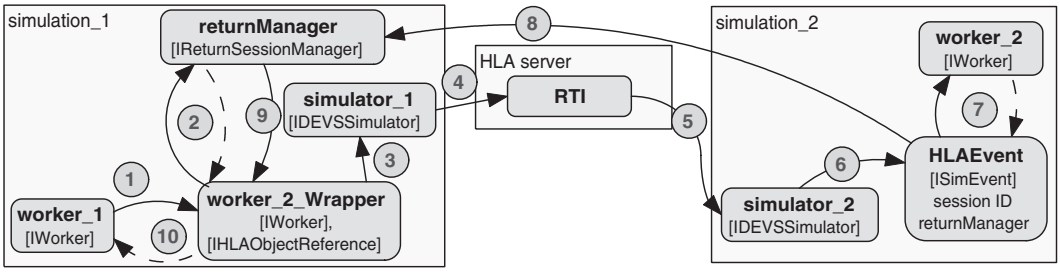
As the calling model object remains completely unaware of the middleware processes performed, we conclude that the mechanism introduced above realizes remote synchronous interaction while preserving deployment neutrality.

7 CONCLUSIONS

We presented a system of middleware services to enable Object Oriented interaction in a distributed, multi-simulator setting. Asynchronous and synchronous method invocation were identified as the main required interaction types. In addition, we identified the requirement of deployment neutrality, to preserve the modularity and extendability advantages of OO models.

The middleware service developed employ functionality provided by DEVS simulators, the HLA and naming services. Remote Method Invocation Scheduling (RMIS) was developed as the main mechanism for OO-style synchronized remote interaction. We demonstrated how RMIS enabled (a)synchronous remote, synchronized interaction in a deployment neutral manner. As a result, the middleware services developed can be used to deploy simple, non-remote OO models in a multi-simulator, distributed setting with minimal effort.

Several issues remain for further research. First, the performance of the services proposed needs to be analyzed. Issues potentially limiting performance are the use of dynamic code generation (to generate wrappers) and the implication of zero lookahead in case of synchronous interaction. Design directions to improve the latter include the use of optimistic simulator synchronization and the application of smart remote event prediction.



1. **Worker_1** has retrieved a wrapped instance of **worker_2** and invokes the method **getJobCount()** . The wrapper internally delegates the call to the generic method **callMethod(Object[ ] args, String name)** .
2. The wrapper registers itself as a return value listener with the **returnManager** , who returns a session id.
3. The wrapper constructs an **HLAEvent**, initialized with the session id. After performing RMIS, the wrapper blocks the simulator thread.
4, 5. The broadcast interaction is handled using RMIS. As a result, the event is scheduled at **simulator_2**.
6. The simulator executes the **HLAEvent**.
7. The **HLAEvent** invokes the correct method on **worker_2** . In addition, the method's return value (here: an **int**) is retrieved.
8. The **returnManager** is notified of the return value of the call identified by the session id.
9. The **returnManager** notifies **worker_2_Wrapper** of the received return value.
10. The **worker_2_Wrapper** unblocks the simulation thread and returns the received return value.
From the perspective of **worker_1** , an ordinary synchronous method invocation has just been completed.

Figure 7: Synchronous Interaction Using RMIS

Finally, we note that the presented implementation is Java based. However, we feel the concepts presented here could easily be translated into a platform neutral implementation using, for example, CORBA.

REFERENCES

Arnold, K., J. Gosling, and D. Holmes. 2000. *The java (tm) programming language (3rd edition)*. Boston, MA: Addison-Wesley Professional.

Birrell, A. 1992. Session: An assessment of the remote procedure call mechanism. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, 1–3.

Birrell, A., and B. Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems* 2 (1): 39–59.

Boer, C., and A. Verbraeck. 2002. Connecting high level distributed simulation architectures: An approach for a famas-hla bridge. In *Proceedings of the 14th European Simulation Symposium, The Society for Computer Simulation International SCS-European Publishing House*, 398–405.

Booch, G. 1991. *Object oriented design*. Benjamin/Cummings.

Booch, G., J. Rumbaugh, and I. Jacobson. 1999. *The unified modeling language user guide*. Indianapolis, IN: Addison-Wesley.

Box, D., D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. 2000. Simple object access protocol 1.1. Technical report, W3C.

Dahl, O.-J., and K. Nygaard. 1966. Simula - an algol-based simulation language. *Communications of the ACM* 9 (9): 671 – 678.

Fitzgibbons, J., R. Fujimoto, D. Fellig, S. Kleban, and A. Scholand. 2004. Idsim: An extensible framework for interoperable distributed simulation. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*.

Foster, I., C. Kesselman, and S. Tuecke. 2001. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications* 15 (3): 200–222.

Fujimoto, R. 1999. *Parallel and distributed simulation systems*. Wiley.

Heim, J. 1997. Integrating distributed simulation objects. In *Proceedings of the 1997 Winter Simulation Conference*, ed. S. Andradóttir, K. Healy, D. Withers, and B. Nelson.

IEEE 2000. IEEE standard 1516-2000 IEEE standard for modeling and simulation (m&s) high level architecture (hla) – framework and rules.

Iskra, K., R. Belleman, G. van Albada, J. Santoso, P. Sloot, H. Bal, H. Spoelder, and M. Bubak. 2002. The polder computing environment, a system for interactive distributed simulation. *Concurrency and Computation: Practice and experience* 14:1313–1335.

Jacobs, P., N. Lang, and A. Verbraeck. 2002. D-SOL; a distributed java based discrete event simulation architecture. In *Proceedings of the 2002 Winter Simulation Conference*, ed. E. Yücesan, C.-H. Chen, J. Snowdon, and J. Charnes, 793 – 800.

Kuhl, F., R. Weatherly, J. Dahmann, and A. Jones. 1999. *Creating computer simulation systems: An introduction to the high level architecture*. New Jersey: Prentice-Hall.

Lang, J., B. Bogovich, S. Barry, B. Durkin, M. Katchmar, J. Kelly, J. McCollum, and J. Potts. 2001. Object-oriented programming and design patterns. *ACM SIGCSE Bulletin* 33 (4): 68–70.

Lang, N., P. Jacobs, and A. Verbraeck. 2003. Distributed, open simulation model development with DSOL services. In *Simulation in Industry. Proceedings 15th European Simulation Symposium*, ed. A. Verbraeck and V. Hlupic, 210–218.

Martin, J., and J. Odell. 1998. *Object-oriented methods: A foundation uml edition*. Prentice Hall.

Menascé, D. 2005. Mom vs. rpc: Communication models for distributed applications. *IEEE Internet Computing* 9 (2): 90–93.

Object Management Group. 1995. *The common object request broker: Architecture and specification*.

Papazoglou, M., and D. Geogakopoulos. 2003. Service oriented computing. *Communications of the ACM* 46 (10): 25–28.

Silva, A., P. Sousa, and M. Antunes. 1998. Naming: design pattern and framework. In *Proceedings COMPSAC '98 - 22nd International Computer Software and Applications Conference, August 19-21, 1998, Vienna, Austria*, 316–323: IEEE Computer Society.

Spracklen, L., and S. Abraham. 2005. Chip multithreading: Opportunities and challenges. In *Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)*.

Zeigler, B. P., H. Praehofer, and T. Kim. 2000. *Theory of modeling and simulation. integrating discrete event and continuous complex dynamic systems*. 2nd ed. Academic Press.

AUTHOR BIOGRAPHIES

NIELS A. LANG is a PhD. candidate at the Rotterdam School of Management. His research focuses on the application of simulation to support logistic system design. Specifically, he investigates the possibilities of integrating operational and economic logistic models. He is co-developer of the DSOL project. His e-mail address is <nlang@rsm.nl>.

PETER H.M. JACOBS is a PhD. student at Delft University of Technology. His research focuses on the design of simulation and decision support services for the web-enabled era. His working experience within the iForce Ready Center, Sun Microsystems (Menlo Park, CA), and engineering education at Delft University of Technology founded his interest for this research. His e-mail address is <p.h.m.jacobs@tbm.tudelft.nl>.

ALEXANDER VERBRAECK is an associate professor in the Systems Engineering Group of the Faculty of Technology, Policy and Management of Delft University of Technology, and a part-time full professor in supply chain management at the R.H. Smith School of Business of the University of Maryland. He is a specialist in discrete event simulation for real-time control of complex transportation systems and for modeling business systems. His current research focus is on development of open and generic libraries of object oriented simulation building blocks in Java. Contact information: <a.verbraeck@tbm.tudelft.nl>.