A Layered Architecture for the Model-driven Development of Distributed Simulators

Conference Paper · January 2008

DOI: 10.1145/1416222.1416291 · Source: DBLP

Daniele Gianni
-
53 PUBLICATIONS **527** CITATIONS
SEE PROFILE

Andrea D'Ambrogio
University of Rome Tor Vergata
169 PUBLICATIONS **1,682** CITATIONS
SEE PROFILE

Giuseppe Iazeolla
University of Rome Tor Vergata
180 PUBLICATIONS **750** CITATIONS
SEE PROFILE

A Layered Architecture for the Model-driven Development of Distributed Simulators

Daniele Gianni
Dept. of Electrical and Electronic Engineering
Imperial College London
Exhibition Road
SW7 2AZ, London, UK
gianni@imperial.ac.uk

Andrea D'Ambrogio
Dept. of Computer Science
University of Rome TorVergata
Via del Politecnico, 1
I-00133 Rome, Italy
dambro@info.uniroma2.it

Giuseppe Iazeolla
Dept. of Computer Science
University of Rome TorVergata
Via del Politecnico, 1
I-00133 Rome, Italy
iazeolla@info.uniroma2.it

ABSTRACT

The development of a distributed simulator requires knowledge and skills that might not be available or suitable in particular situations. Bringing model-driven approaches to the development of distributed simulators contributes to reduce both the need for specific skills and the development effort. To support this innovative development methodology, we introduce a layered simulation architecture named SimArch that allows to define simulation models that can be transparently transformed into a simulation programs ready to be executed in a distributed (as well as local) fashion.

SimArch defines layers of services at increasing levels of abstraction on the top of the distributed environment, thus allowing developers to build distributed simulators without explicit knowledge about the execution environment (local/distributed) and the specific distributed simulation infrastructure (e.g., HLA).

In order to show the effectiveness of the proposed approach, SimArch has been provided with an Extended Queueing Network (EQN) simulation modeling language, which has been applied to the development of an example distributed simulator in the computer network domain.

Categories and Subject Descriptors

D.2.11 Software Architecture, D.2.13 Software Reusability, D.2.10 Design, D.3.2 Language Classification, I.6.5 Model Development, I.6.7 Simulation Support Systems, I.6.8 Discrete Event, I.6.8 Distributed, I.6.2 Simulation Language.

General Terms

Design, Experimentation, Languages.

First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems, (SIMUTOOLS'08), March 3–7, 2008, Marseille, France.
Copyright 2008 ACM 1-58113-000-0/00/0004…$5.00.

Keywords

Simulation Languages, Simulation Framework, Distributed Simulation, HLA, Queueing Network, Computer Network Simulation.

1. Introduction

Distributed simulation brings well-known advantages with respect to local simulation, mainly in terms of resource availability [1]. The availability of more memory and computational power can indeed allow the efficient execution of simulation programs that could not be run on a single machine. On the other hand, the implementation of a distributed simulator is far from being easy and effortless, and very often distributed simulators are to be developed by appropriately adapting existing simulation components that have been developed for local execution [2].

Although much attention has been devoted to minimize the development effort by addressing interoperability and reusability issues, the ease (i.e. the adaptability) with which a local simulator developer can approach distributed simulation has not been considered. In order to address such, we introduce a layered architecture named *SimArch* that exploits simulation approaches based on the precepts recently introduced in the *model-driven development* field [3].

Model-driven development is simply the notion that it is possible to build an abstract model of a system that we can then transform into more refined models and eventually into the system implementation. In the proposed approach, model-driven development is applied to distributed simulation by use of SimArch, which allows to develop a simulator by simply defining a simulation model that can be effortlessly transformed into a simulation program ready to be executed in a distributed (as well as local) fashion.

SimArch defines a set of layers at increasing levels of abstraction on top of a distributed computing infrastructure (HLA in our case [4]) to allow the simulation developer to deal with simulation component logic issues only, without explicitly focusing on specific (distributed or local) execution environment and simulation infrastructure. In such a way, SimArch brings several advantages in terms of simulation component reusability and effort reduction.

In order to show the effectiveness of the proposed approach, SimArch has been provided with an Extended Queueing Network

(EQN) [5] simulation modeling language. Such language has been built by developing various software libraries that implement the services provided by the different layers of SimArch [6] [7] [8].

The paper is organized as follows. Section 2 briefly illustrates the related work, while Section 3 gives the description of the proposed architecture. Section 4 describes the EQN language built on top of SimArch layers, and Section 5 illustrates its application to the development of an example distributed simulator in the computer network domain.

The example application has been carried out by first defining the EQN model of a so-called catenet, and then showing how the EQN simulation modeling language built on top of SimArch can be effectively used to develop both a local version and a distributed version of the catenet simulator, with a twofold motivation: comparing the effort or developing the local and the distributed simulators and providing confidence about the effectiveness of the proposed approach.

2. Related work

The simulation community has provided several contributions that aim at increasing the abstraction level of the simulation primitives in order to achieve effort savings and reusability when building distributed simulators. However, the adaptability, i.e., the ease with which a new distributed simulator can be produced from existing ones, has not been thoroughly addressed.

Some related research efforts can be found in [9] [10] [11] [12], which describe PDNS, DisSimJava, DEVS/HLA, and OSA, respectively.

PDNS is an extension of the popular ns-2 tool for running network simulation in a distributed fashion. SimArch differs from PDNS in two main aspects. The first is that SimArch is application domain independent, and therefore it does not deal with computer networks specifically. The second is the uniform representation of the communication among entities, independently from their type (local-local, local-remote, or remote-local), while PDNS explicitly introduces the ghost node abstraction to locally represent the connections to remote entities.

The DisSimJava work presented in [10] is based on a view similar to SimArch, from which it differs for three major aspects: first, it is only a prototype with the implementation of communication entities but without synchronization; second, it does not identify a layered architecture that allows simulation components to be run on different implementations; third, it does not provide HLA-compliant implementations.

DEVS/HLA [11] and the associated definition of a simulation middleware [13] can also be considered contributions similar to SimArch. The basic difference is that SimArch approaches the easier problem of enabling developers familiar with local simulation to easily switch to distributed simulation, with a more pragmatic solution inspired by the current trend of semi-formal modeling languages, like UML. Moreover, SimArch clearly defines the communication interfaces between the layers and, at the same time, allows the composition of any set of implementations of such layers. In the end, SimArch could also be used to provide a DEVS implementation [14].

Finally, OSA [12] has a broader scope compared to SimArch. OSA aims at providing a general framework to cover the activities involved in the definition, development, validation and execution of a discrete event simulators. SimArch, differently, focuses on a restricted sub-set of the above activities, but in a deeper way. SimArch partitions the provided simulation services into four independent layers, whose implementation can be effortlessly changed to accommodate different requirements or future needs.

In addition, SimArch is currently provided with jEQN [15], a very flexible Domain-Specific Language (DSL) [16] for the description of EQN models.

3. SimArch

Distributed simulator systems can be considered as composed of several layers, each introducing a more abstract set of simulation services on top of the underlying layer, the bottom one being the distributed computing infrastructure.

There are three main advantages for adopting this layering approach. The first is that the simulation model is decoupled from the specific execution environment, and then it can be reused across several simulation platforms. The second is that layers' implementations can be easily switched to accommodate custom deployment requirements (i.e. local or distributed deployment, performance optimization for given simulation workload, etc.). Finally, the third advantage is that simulation developers deal only with high level simulation services, and therefore can focus on the model description rather than being involved in the typical intricacies of distributed simulators implementation.

The proposed layered simulation architectures is denoted as SimArch and is composed of four layers [13] [8], each dealing with a specific distributed simulation issue.

Figure 1 illustrates the architecture layers, whose detailed description is given in [8].

In SimArch, the Layer 4 (top layer) is the layer of the simulation model definition through the invocation of the simulation language primitives.

The primitives' implementation, i.e., the components' simulation logic and the model configuration services, are provided by Layer 3; while Layer 2 deals with the simulation components synchronization and communication, transparently for local and distributed environments. The distributed version of this layer uses in turn Layer 1 to achieve global time synchronization and provide communication with the remote simulation components.

Finally, Layer 1 provides a DES (discrete-event simulation) abstraction on top of the distributed computing infrastructure conventionally identified by Layer 0. Such bottom layer does not belong to the architecture and therefore the service interfaces between layers 1 and 0 is not defined. In the case of a HLA-based implementation of layer 1, such interfaces correspond to subsets of the *RTIAmbassador* and *FederateAmbassador* interfaces for the communication between layers 1 and 0 and between layers 0 and 1, respectively.

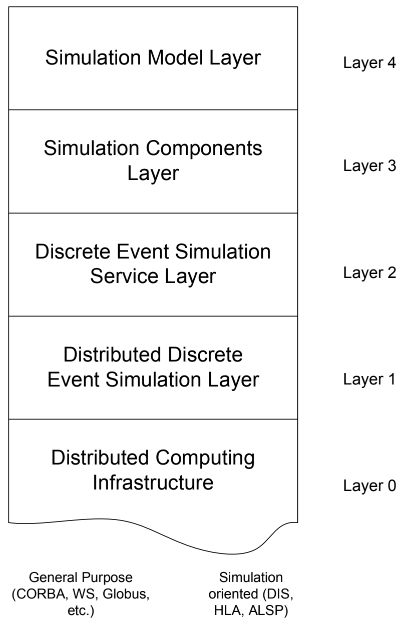

Simulation Model Layer
Layer 4
Simulation Components Layer
Layer 3
Discrete Event Simulation Service Layer
Layer 2
Distributed Discrete Event Simulation Layer
Layer 1
Distributed Computing Infrastructure
Layer 0
General Purpose (CORBA, WS, Globus, etc.)
Simulation oriented (DIS, HLA, ALSP)

Figure 1 SimArch layers

3.1 Data interfaces

The data interfaces define the access methods to the data exchanged between the layers. Specifying abstract data structures instead of concrete data structures decreases the level of layer coupling, and therefore allows layers' implementation to be effortlessly switched.

SimArch defines the following data interfaces:

- *ComponentLevelEntity*: interface for local simulation entities (including their hierarchical composition);
- *Event*: interface for events scheduled between layers;
- *GeneralEntity*: a base interface for local and remote entities;
- *InputPort*: interface for received events;
- *Link*: interface for connecting input and output ports;
- *Name*: interface for the declaration of names in SimArch;
- *OutputPort*: interface for sent events;
- *Port*: interface for the common access to port internal data;
- *RemoteEntity*: interface for remote simulation entities;
- *Time*: interface for the representation of simulation time;

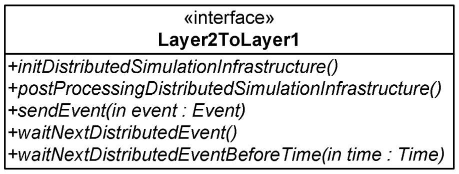
Each of the above defined interface is of immediate understanding and therefore not discussed further here. The reader is sent to [7] [8] for additional details.

3.2 Layers' service interfaces

The service interfaces defines communication path between adjacent layers, in both directions. Layers 1 and 2 communicate through interfaces *Layer2ToLayer1* (Figure 2) and *Layer1ToLayer2* (Figure 3); similarly Layer 2 and 3 communicates through interfaces *Layer3ToLayer2* (Figure 4) and *Layer2ToLayer3* (Figure 7). At Layer 4, simulator developers can describe their simulators by assembling Layer 3's domain-specific components through *Layer3UserInterface* (Figure 5), which is to be considered as *Layer4ToLayer3* SimArch interface.

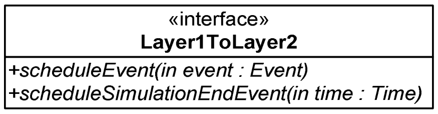
Layer2ToLayer1 (Figure 2) interface allows communication from Layer 2 to Layer 1. The interface is defined through five services. The *initDistributedSimulationInfrastructure* and *postProcessingDistributedSimulationInfrastructure* services are general placeholders to allow the system set-up and the initial state recover after the simulation execution, respectively. The remaining three services provide DES abstraction in the distributed system. *sendEvent* delivers the given event to the recipient at the specified time; *waitNextDistributedEvent* blocks the invoking thread until a distributed event is received, and *waitNextDistributedEventBeforeTime* blocks the invoking thread until either a distributed event is received or the specified time is reached by the distributed simulation.

Layer1ToLayer2 (Figure 3) interface allows Layer 1 to transparently schedule distributed events into the local system. This functionality, however, is split into two services, *scheduleEvent* and *scheduleSimulationEndEvent*, to further decouple Layers 1 and 2.



```
«interface»
Layer2ToLayer1
+initDistributedSimulationInfrastructure()
+postProcessingDistributedSimulationInfrastructure()
+sendEvent(in event : Event)
+waitNextDistributedEvent()
+waitNextDistributedEventBeforeTime(in time : Time)
```

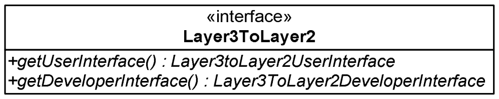
Figure 2 Layer2ToLayer1 interface [8]



```
«interface»
Layer1ToLayer2
+scheduleEvent(in event : Event)
+scheduleSimulationEndEvent(in time : Time)
```

Figure 3 Layer1ToLayer2 interface [8]

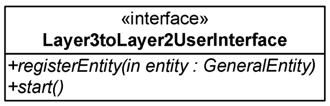
The communication from Layer 3 to Layer 2 takes place through the interface *Layer3ToLayer2* (Figure 4), which is in turn composed of a user-oriented service set (*Layer3ToLayer2UserInterface* - Figure 5) and a developer-oriented service set (*Layer3ToLayer2DeveloperInterface* - Figure 6).



```
«interface»
Layer3ToLayer2
+getUserInterface() : Layer3toLayer2UserInterface
+getDeveloperInterface() : Layer3ToLayer2DeveloperInterface
```

Figure 4 Layer3ToLayer2 interface [8]

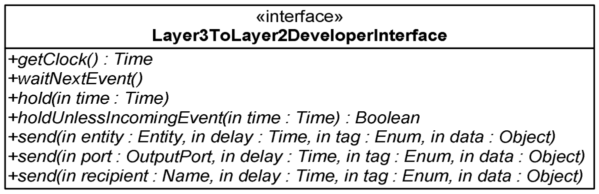
Layer3ToLayer2UserInterface groups the services that enable the simulation language user (simulator developer at Layer 4) to produce the simulator. The interface provides two basic configuration and simulation management services: *registerEntity*, to add a simulation component to the simulator; and *start*, to make the execution container start the simulation execution.



```
«interface»
Layer3toLayer2UserInterface
+registerEntity(in entity : GeneralEntity)
+start()
```

Figure 5 Layer3ToLayer2 user interface [8]

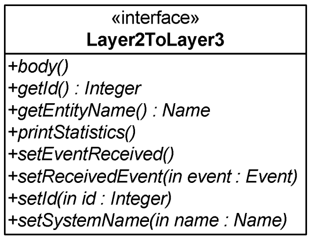
Layer3ToLayer2DeveloperInterface provides DES services to build the simulation logic of each component. The services are of four types: *waitNextEvent*, to block the component execution until an event is received; *hold*, to block the component execution for t simulated seconds; *holdUnlessIncomingEvent*, to block the component execution for *t* simulated seconds or until an event is received, whichever happens first; and, finally, a *send* method to deliver events at a given simulated time.



```
«interface»
Layer3ToLayer2DeveloperInterface
+getClock() : Time
+waitNextEvent()
+hold(in time : Time)
+holdUnlessIncomingEvent(in time : Time) : Boolean
+send(in entity : Entity, in delay : Time, in tag : Enum, in data : Object)
+send(in port : OutputPort, in delay : Time, in tag : Enum, in data : Object)
+send(in recipient : Name, in delay : Time, in tag : Enum, in data : Object)
```

Figure 6 Layer3ToLayer2 developer interface [8]

Layer2ToLayer3 interface defined the Layer 2 access to Layer 3 simulation components (Figure 7).



```
«interface»
Layer2ToLayer3
+body()
+getId() : Integer
+getEntityName() : Name
+printStatistics()
+setEventReceived()
+setReceivedEvent(in event : Event)
+setId(in id : Integer)
+setSystemName(in name : Name)
```

Figure 7 Layer2ToLayer3 interface [8]

The *body* method encapsulates the simulation logic of the component, and therefore is invoked when the component container is started. This method is made up of *Layer3ToLayer2* service calls (i.e. *hold*, *waitEvent*, etc.) plus operations on the component state variables (e.g. the length of a queue, the state busy/free of a service center, etc.). The other services are of immediate understanding and therefore not discusses further here.

3.3 Layers' implementation

To provide evidence of SimArch usefulness, we designed and developed a Java library for each of the layers (1 through 3), and produced an example simulator at Layer 4, as described in Section 5. The three libraries have been denoted as DDESoverHLA (at Layer 1), SimJ (at Layer 2)and jEQN (at Layer 3).

DDESoverHLA implements *Layer2ToLayer1* interface on top of HLA and CORBA-HLA [17].

SimJ provides a framework for the local development of simulation components that can be equivalently deployed in local and distributed simulations.

Finally, *jEQN* defines a DSL [16] for the definition of EQN simulators [15]. A DSL can be seen as a programming language designed for a specific kind of task, in contrast to a general-purpose programming language, such as C or Java.

The combination of SimArch and the three libraries allows the portability of locally developed simulation components onto HLA-based infrastructures and, at the same time, brings effort savings in the development of HLA-based simulators. Such savings are due to the fact that the simulator developers are only concerned with the essential services related to the simulation logic, while the synchronization, the interaction between the local and the distributed environment, and the distributed communication are transparently provided by SimArch behind the *Layer3ToLayer2DeveloperInterface*, as illustrated in [6] [7] [8].

The next section gives a brief overview of the jEQN language that has been used to simulate the example system illustrated in Section 5.

4. jEQN brief overview

As above mentioned, jEQN is a DSL for the definition of EQN models and for the implementation of EQN simulators on top of SimArch. jEQN is currently implemented as a Java library though its design features allow to make use of any other object oriented language [15].

The language is defined by Layer 4's services (i.e. *Layer3ToLayer2UserInterface*'s services) plus the syntax of the proper jEQN simulation components. This in turn includes the components' name, assigned according to the EQN standard taxonomy (e.g., user sources, waiting systems, service centers, routers and special nodes), and support components for their parameterization (e.g., the policy framework) and for the data structures (e.g., User, Queue, etc.).

jEQN has been designed to encapsulate the simulation logic only within simulation components and to leave undefined the parameters that do not directly affect it. In such a way, each simulation component exhibits a high degree of cohesion and it can be easily reused across the several values the parameters might assume. The parameters are designed and developed as support components, in order to decouple the jEQN simulation components from the parameters themselves and to allow a flexible configuration of such components.

jEQN defines the following types of support components:

cat 1. parameters used to take decisions;

cat 2. parameters that provide a sequence of values;

cat 3. parameters that provide storage support.

Parameters like the routing policy (decision: where to route a given user), the termination condition (decision: assess whether or not the source has to terminate), the enqueueing policy (decision: where to insert a given user), etc., belong to the first category. For these parameters, jEQN defines a framework and a taxonomy within which each policy has to be classified according to the use and the type of the following four parameters [15]:

- I: the type of implicit input;
- S: the type of the policy state data;
- T: the type of the explicit input;
- D: the type of the decision.

Where implicit input is the input taken at policy instantiation time, the policy state is the internal state of the policy and the explicit input is the data for which the decision has to be taken (the meaning of the type of decision being straightforward).

The second category includes parameters like interarrival times, user generators, etc.

Finally, the third category defines parameters for the components that provide storage support for the simulation, such as queues.

The definition of an EQN model is carried out by identifying: the EQN components (e.g.: user sources, waiting systems, etc.), their properties (e.g.: enqueueing policy, routing policy, user interarrival rate, etc.), and their connections.

The development of a jEQN simulator, either local or distributed, follows the same process, and therefore it contributes to bring model-driven development into simulation systems.

Further jEQN modeling details are available in [15].

5. Distributed computer network simulation with jEQN

Figure 8 illustrates the example system that has been simulated by use of jEQN. The system is a so called *catenet*[1], composed of:

(a) two separate LANs, the first (*LAN1*) a token ring that connects Host A, and the second (*LAN2*) an Ethernet that connects Host B

(b) two gateways *GW1* and *GW2*, that connect LAN1 and LAN2 to the WAN, respectively,

(c) the *WAN*, an X.25 packet switched network.

The objective of the simulation study is to evaluate the *end-to-end delay* between *Host A* and *Host B*, the former in LAN A and the latter in LAN B. Host A acts as a client and Host B acts as a server.

We developed both a local version of the simulator and a distributed one, in order to:

- validating the results (i.e., the end-to-end delay) obtained from both the local simulator and the distributed simulator through direct comparison with already available data (see [18]) and
- comparing the effort needed for building the distributed simulator with respect to the one required for developing the local simulator.

5.1 Model definition

It is assumed the interaction between the client (Host A) and the server (Host B) is based on a message exchange that is carried out as a packet flow over the various components of catenet. Such a packet flow involves several technologies (token ring, X.25, ethernet) and thus several mechanisms to deal with heterogeneity, namely:

(m1) protocol conversion, from the transport layer protocol TCP, to the network layer protocol IP, to the data-link layer and physical layer protocols (and vice versa), in either direction from HA to HB, with the IP to X.25 (and vice versa) protocol conversion at the gateway level,

(m2) packet fragmentation and re-assembly at many protocol conversion interfaces,

(m3) window-type flow control procedure operated at transport layer by protocol TCP for a fixed window size of value C (for the sake of simplicity no varying window sizes are considered, nor the use of congestion-avoidance algorithms).

Figure 9 gives a more detailed view of the packet flow in the system and puts into evidence the work performed by the network components in order to deal with mechanisms m1, m2 and m3, when transferring data from Host A to Host B (a symmetrical flow holds when transferring data from Host B to Host A).

Packets are originated in TCP format and enter LAN1 in IP format. From LAN1 they exit in LLC/MAC802.5 format to enter GW1 that fragments them into X.25 format to be accepted by the WAN. Vice versa for the GW2, where X.25 packets are re-assembled into LLC/MAC802.3 format for LAN2 and from there into the IP format and then in the TCP format. The token pool in Figure 9 is introduced to represent the window-type flow control procedure implemented by TCP between the Host A exit and the Host B entrance. For a window size C, up to C consecutive TCP packets can GET a token from the pool and enter the network layers section of Host A. Non-admitted packets are queued in front of this section. On the other hand, admitted packets RELEASE their token at Host B entrance, thus allowing another packet to enter. When data transfer takes place in the opposite direction (from Host B to Host A), the GET node with its queue takes the place of the RELEASE node, and vice versa. Further details about the definition of the model of the catenet system can be found in [18].

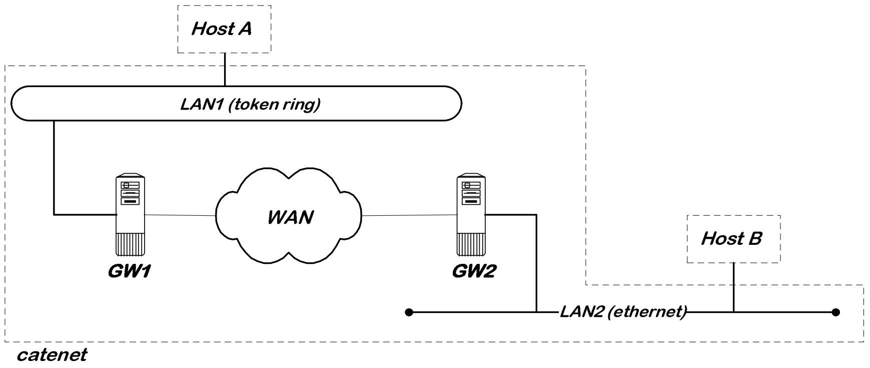


Figure 8. General view of the catenet system

[1] The term “catenet” is an obsolete term used to indicate a system of packet-switched communication networks interconnected via gateways [21]. It is here used for the sake of brevity to denote a computer network speaking the Internet protocol.

5.2 Design and implementation details

The implementation of the local version of the simulator consists of the following three steps:

1. Istantiate the local Layer 2 container (or simulation engine) to execute the jEQN components;
2. Define the jEQN components of the model by:
    a. Defining the jEQN entities
    b. Defining the connections among entities
3. Start the execution container.

It is worth noting that jEQN components are automatically registered with the underlying Layer2 container that execute them.

Step 1 and step 3 can be simply completed by use of SimArch implementations and SimArch services. Step 2, the core activity for the development of the simulator, is carried out by use of a procedure that can be partially automated to obtain jEQN components from the detailed definition of the EQN model. To this purpose, a further detailed view of the simulated model is first given.

For the sake of brevity, we report here only the details of the token ring (LAN1) element of the overall EQN model. A similar reasoning also applies to the remaining elements, whose details can be found in [18].

Figure 10 shows the details of the LAN1 EQN model, which is based on the typical behavior of local area networks of token ring type. The queueing policies at each queue, including token allocation queues, are of FCFS (First Come First Served) type, while the service centers are of non-preemptive type with Gaussian service times. The users (packets) flow into the main loop (from *allocTok* node to the *routing* node) until the number of the frames they are composed has been processed by the token ring components.

The implementation of the simulator is carried out by defining a jEQN component for each corresponding element in Figure 10. The jEQN thus consists of six waiting systems, three service centers, one router, two allocate nodes, one release node, one destroy node, two set nodes, and finally two pools of tokens.

The configuration of such components is carried out by applying the jEQN parametrization procedure that identifies, for each component, appropriate parameters within the three jEQN categories of parameters described in Section 4.

As an example, each queue in the model is defined through an enqueueing policy (cat 1 parameter) and a storage structure (cat 3 parameter). The enqueueing policy can be defined through the jEQN policy framework by first classifying the policy according to the jEQN policy taxonomy and then identifying the parameters [15]. The FCFS enqueueing policy can be modelled as a policy that depends on the implicit input parameter (the queue), which has no internal state, no explicit input, and whose returned decision data is the index within the queue. This policy is easily implemented by returning as "decision" the length of the queue.

As regards the storage parameter, the model implicitly considers all the queues as infinite single FCFS queues. In jEQN, this is can be done by first choosing a concrete data structure implementing the Java List interface, and then setting up both the storage parameter of such structure and the enqueueing policy. jEQN modularity and flexibility allows the introduction of more complex storage structures (e.g., multiqueue with different enqueueing policies and capacity, etc.), with no additional costs.

The service centers are of non-preemptive type, according to the model definition, and their parameters are limited to the type of service, i.e. to the sequence of numbers that simulate the introduced delay. The Gaussian property of service centers is set by use of the jRand framework [19], by passing a Gaussian pseudo-random generator at instantiation time.

The router is allocated with the proper routing policy. This policy is once again defined by use of the jEQN policy framework. It depends on the parameter T, explicit data, which is of type User,

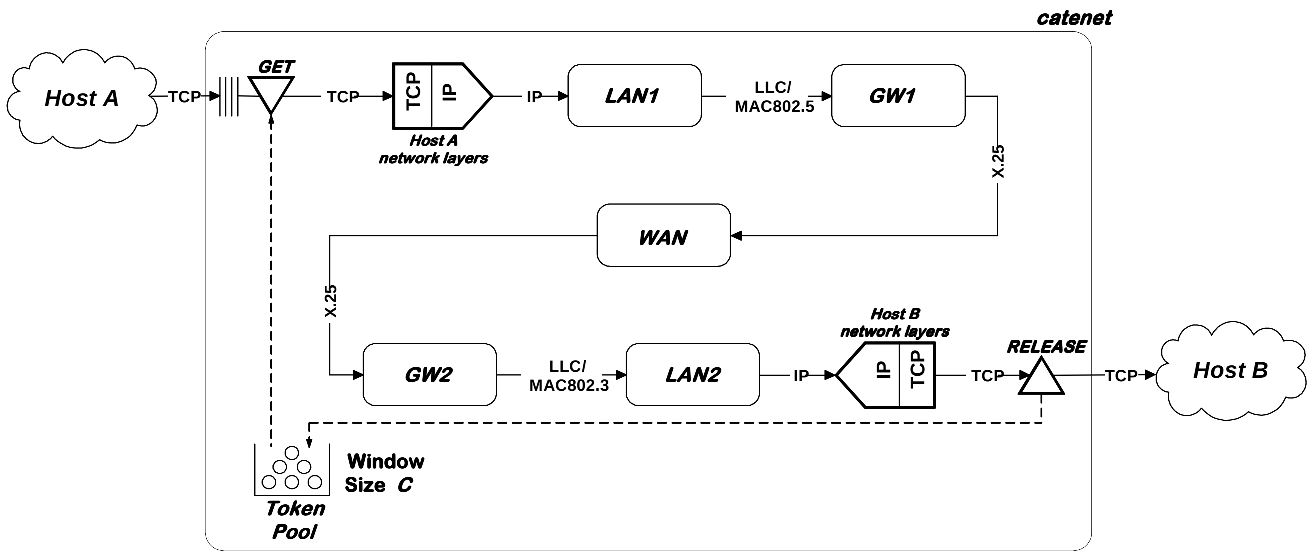


Figure 9 Detailed view of the catenet packet dataflow

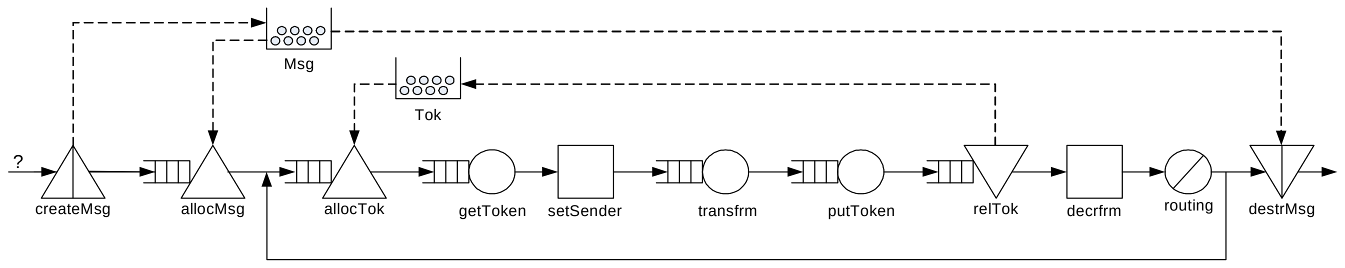


Figure 10 EQN details of the token ring LAN1 submodel

and returns a decision of type Integer, which indicates the port id through which the processed User has to be forwarded. The implementation of this policy can be easily obtained by testing whether the number of frame is equal to zero or not. In the former case, the port id returned is the id of the port connected to the *destrMsg* node, whereas in the latter case is returned the id of the port connected to the *allocTok* node.

The remaining jEQN components, corresponding to nodes for allocating and releasing tokens, do not need to be parametrized.

The simulation model definition is completed by connecting the components as in the model specification. To this purpose, the jEQN implementation of the Link SimArch interface can be used to instantiate links and to register components' ports as in [6]. For example, the *allocTok* waiting system has its user output port connected to the *getToken* service center, which is in turn connected to the following *setSender* special node, and so on for the remaining components.

The implementation of the distributed simulator is easily obtained from the corresponding local version. This further remarks how the abstraction layers introduced by SimArch actually allow the model-driven development of distributed simulators, as they are almost not aware of the distributed execution environment.

In general, the implementation of a SimArch distributed simulator follows four steps, for each (distributed) submodel:

1. Instantiate Layer1 and Layer2 implementations;
2. Define the submodel by:
   a. Defining the entities:
      i. Local simulation entities as in the local simulator;
      ii. Remote simulation entities by declaring remote references through the proper SimArch interface;
   b. Defining the connections:
      i. Local connections among local entities as in the local simulator;
      ii. Remote connections (outcoming connections) from local entities to remote entities.
3. Activate the Layer2 container.

In our example scenario, we decided to partition the model in Figure 8 into three distributed submodels (Submodel 0, Submodel 1 and Submodel 2), as shown in Figure 11.

Submodel 0 consists of the HostA and LAN1 entities and their local connections, as in the local simulator, plus a remote reference to GW1 and a remote connection between the LAN1 entity and GW1.

Similarly, Submodel 1 consists of the GW1, WAN and GW2 entities and their local connections as in the local simulator, plus two remote references to the LAN1 and LAN2 entities with the related remote connections to GW1 and GW2, respectively.

Finally, Submodel 2 consists of the HostB and LAN2 entities and their local connections, as in the local simulator, plus a remote reference to GW2 with the related remote connection.

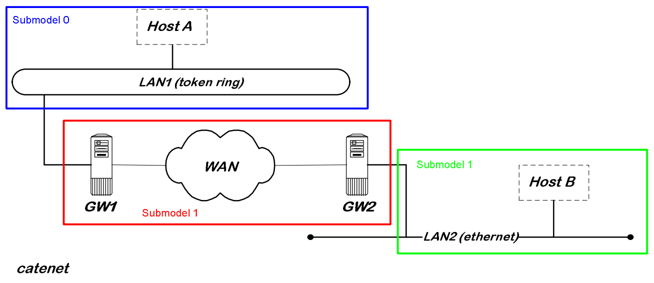


Figure 11 Model partitioning for the distributed simulator

5.3 Simulator validation

The validation is carried out through comparison of exact values and global trend of the end-to-end delay. The exact values are to be directly compared with data available from other studies [18] and from the analytical solution; while the global trend can be confirmed by the behavior of the real system. The end-to-end delay is measured at increasing value of the arrival rate λ (packets/sec required by the source Host A) for different values of the TCP window size (4, 7 and 12 packets). The statistics are collected through the batch method, with initial bias removed, in 5000 simulated seconds. The local simulator results are shown in Figure 12.

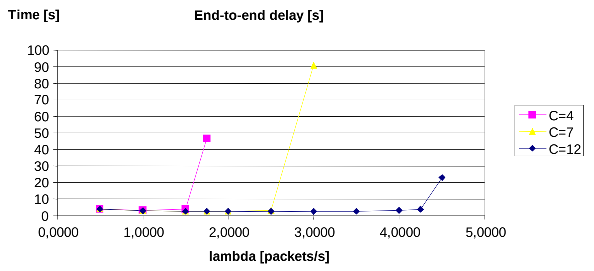


Figure 12 End-to-end delay obtained from the local simulator

Direct comparison between the above results with previous analyses confirms that the experimental data are reasonably within the 10^{-2} sec acceptance interval of theoretical results in [18].

At global level, the simulation results confirm the expected system behavior. In fact, for values of λ in the interval (0;1] the windows size (C) is expected not to have influence on the end-to-end delay; while for λ > 1 the windows size increase is expected to shift the saturation λ towards higher packets/sec values. Indeed, it is easy to be convinced that, by increasing the windows size, the packet sender (i.e., Host A) results less dependent on the delays experimented by single packets.

The distributed simulator validation has been carried out in a similar way. The simulator has been partitioned into three simulators, one for each submodel in Figure 11. Experiments were carried out both in LAN and WAN environments, the latter between Rome (Italy) and Atlanta (USA). Being the system completely simulation-time synchronized, the results of the LAN and WAN execution have not shown any difference.

The results obtained from the distributed simulator are illustrated in Figure 13.

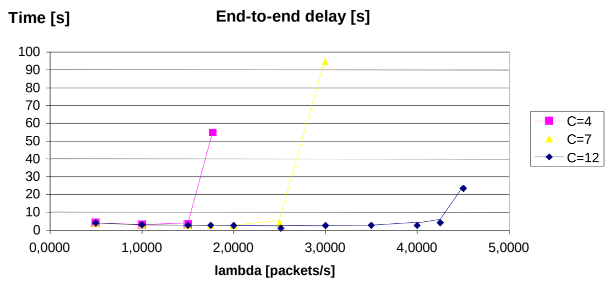


Figure 13 End-to-end delay obtained from distributed simulator

Comparing them with values obtained from previous analysis and from local simulator results, the distributed simulator is itself validated. The small differences in some points comes from the introduction of finite delays in the communication between simulation entities running on different simulators, which have been included to avoid non-reproducibility of zero lookahead simulations with the current HLA implementation [20]. This indeed leads to the simulation of a slightly different EQN model whose end-to-end delay is generally higher and with a lower saturation point for a given window size.

To further validate the distributed simulator, we adapted the local simulator model by introducing fixed delay centers to model the effect of the lookahead introduced in the distributed simulation. As expected, the results obtained by the updated local simulator are identical to the ones obtained from the distributed one.

6. Conclusions

Building a distributed simulator requires a non negligible effort and specialized skills compared to the development of a conventional local simulator. This, however, can be mitigated with the introduction of a set of layered abstract simulation services on top of the distributed environment. In such a way, it is possible to apply a model-driven approach to the development of distributed simulators. In fact, the simulation logic can be specified by use of few basic simulation services, which enables simulator developers to abstract from the details of the underlying (local or distributed) execution environment.

In this paper we have introduced SimArch, a layered architecture that brings model-driven development into the simulation field.

In order to show the effectiveness of SimArch, a set of software libraries that implement the different layers of the architecture have been developed and several simulators produced. In particular, SimArch has been provided with an Extended Queueing Network (EQN) simulation modeling language, which has been then applied to the development of an example distributed simulator in the computer network domain.

The example simulator, which has been validated through direct comparison with the analytical solution and with results obtained from previous studies, has shown that, by use of SimArch, a distributed simulator can easily be built from the definition of the simulation model, either from scratch or by slightly modifying a local version of the same simulator.

7. Acknowledgments

This work has been partially supported by funds from the CERTIA Research Center of the University of Roma TorVergata.

8. References

[1] R. Fujimoto, Parallel and Distributed Simulation Systems, Wiley (2000).

[2] A. Verbraeck, “Component-based distributed simulations: the way forward?”, *Proceedings of rht 18th Workshop on Parallel and Distributed Simulation (PADS04)*, 16-19 May, 2004, pp 141-148.

[3] S.J. Mellor, A.N. Clark, T. Futagami, “Model-driven development”, *IEEE Software*, 20 (5), 2003, 14-18;

[4] IEEE, Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – framework and rules, Technical Report 1516, IEEE (2000).

[5] G. Bolch, S. Greiner, H. de Meer and K. Trivedi, Queueing Networks and Markov Chains, Wiley (1998).

[6] A. D'Ambrogio, D. Gianni and G. Iazeolla, “SimJ: A Framework to Develop Distributed Simulators”, *Proceedings of the 2006 Summer Computer Simulation Conference*, Calgary, Canada, Aug, 2006, pp. 149 – 156.

[7] A. D'Ambrogio, D. Gianni and G. Iazeolla, “jEQN: a Java-based Language for the Distributed Simulation of Queueing Networks”, LNCS vol. 4263/2006, *Proceedings of the 21st International Symposium on Computer and Information Sciences (ISCIS'06)*, Istanbul, Turkey, Nov, 2006, pp 854 – 865.

[8] D. Gianni and A. D’Ambrogio, “A Language to Enable the Distributed Simulation of Extended Queueing Networks”, *Journal of Computer*, Vol. 2, N. 4, Academy Publisher, July, 2007, pp 76 – 86.

[9] G.F. Riley, M.H. Ammar, R.M. Fujimoto, A. Park, K. Perumalla, and D. Xu, “A federated approach to distributed network simulation”, *ACM Transaction on Modeling and Computer Simulation (TOMACS)*, Vol. 14 N. 2, April 2004.

[10] E.H. Page, R.L. Moose and S.P. Griffin, “Web-Based Simulation in SimJava using Remote Method Invocation”, *Proceedings of the 1997 Winter Simulation Conference*, Atlanta, GA, pp 468-474, December 1997.

[11] B.P. Ziegler, G. Ball, H. Cho, J.S. Lee, and H. Sarjoughian, “Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions”, *Proceedings of the 1999 Simulation Interoperability Workshop (SIW99)*.

[12] O. Dalle, “The OSA Project: an Example of Component Based Software Engineering Techniques Applied to Simulation”, *The 2007 Summer Computer Simulation Conference (SCSC’07)*, San Diego, USA, July 15–18, 2007.

[13] B.P. Zeigler, H.S. Sarjoughian, S. Park, J.S. Lee, Y.K. Cho, J.J. Nutaro, “Devs Modeling And Simulation: A New Layer Of Middleware”, *The Third Annual International Workshop on Active Middleware Services*, 2001, p. 21.

[14] B.P. Zeigler, T.G. Kim, H. Praehofer, Theory of Modeling and Simulation, Academic Press (2000).

[15] D. Gianni and A. D’Ambrogio, “A Domain-Specific Language for the Description of Extended Queueing Networks Model”, to appear in the *Proceedings of the IASTED International Conference on Software Engineering (SE08)*, February, Innsbruck, Austria, 2008.

[16] Marjan Mernik, Jan Heering, and Anthony M. Sloane, “When and how to develop domain-specific languages”, *ACM Computing Surveys*, 37(4):316–344, 2005.

[17] A. D’Ambrogio and D. Gianni, “Using CORBA to Enhance HLA Interoperability in Distributed and Web-Based Simulation”, *Proceedings of the 19th International Symposium on Computer and Information Sciences (ISCIS'04)*, Antalya, Turkey, Nov, 2004, pp 696 - 705.

[18] A. D’Ambrogio and G. Iazeolla, “Steps towards the Automatic Production of Performance Models of Web-Applications”, Computer Networks, n. 41, pp 29-39, Elsevier Science, 2003

[19] D. Gianni, “jRand: A Flexible Framework for Number Sequences”, Technical Report RI.20.2003, University of Rome TorVergata, 2003

[20] R. Fujimoto, “Zero Lookahead and Repeatability in the High Level Architecture”, *Proceedings of the 1997 Spring Simulation Interoperability Workshop*, Orlando, FL, USA, March, 2007

[21] L. Pouzin, “A Proposal for Interconnecting Packet Switching Networks”, *Proceedings of EUROCOMP*, Brunel University, May 1974, pp. 1023-36.