A GENERAL PROGRAMMING MODEL FOR DISCRETE-TIME DISTRIBUTED SIMULATION

João Duarte

Department of Computer Science and Engineering,
University of West Bohemia in Pilsen,
Czech Republic

duarte@kiv.zcu.cz

Abstract

This article presents a general programming model for discrete-time distributed simulation. The model uses a shared memory interface that can be used by simulation applications. It allows for a description of the whole distributed simulation to be written in XML and compiled, generating code that can be used directly by the simulation application, requiring only small modifications to parallelize a single computer simulation program into a distributed one. The model is language and system independent and does not restrict to a single scheme of communication and model time synchronization. It makes use of the CORBA framework to achieve inter-operability. The model also enables to incorporate already existing simulation programs as well as real-time programs and computer systems. A review of time-stepped algorithms for distributed simulations and their possible applications, and another review about distributed barrier algorithms are included in the article. A time-stepped implementation of the model was developed, and a case study included, which uses a queuing networks example. The performance and precision of the developed model were evaluated through tests, using the provided case study, and the results were satisfactory, showing that by using the proposed model is possible to achieve high parallelization performance without loss in precision.

Keywords: Distributed simulation, time-stepped, discrete-time, CORBA

1 Introduction

In a previous paper by the same author [1] the need of a general programming model for discrete-time distributed simulation was discussed. A survey of frameworks for distributed computing showed how the Common Object Request Broker Architecture (CORBA) could be the best solution for the problems of inter-operability between different languages and operating systems. A discrete time-stepped simulation model was developed and proved by implementation. It was shown to be possible to guarantee synchronization between different simulation instances using a simple prototype based on CORBA remote communication.

In [2], the notion of using CORBA as a framework for generic discrete-event distributed simulation is firstly introduced. It is claimed that it provides a location-transparent and language-independent mechanism for generic and remote simulations, and proven by a simple prototype. As stated by the author, the goal is to show the potentialities of CORBA as a possible framework for discrete-event simulation. A simple prototype was developed by the author to illustrate how it could be done.

In [3] is proposed a web-based network simulation framework using CORBA technology. They attempt to provide a flexible, extensible, platform and language-independent simulation environment, suitable for large-scale deployment over the World Wide Web. But it is mainly focused on network specific simulations.

In [4], is presented a CORBA based Time-Warp simulator, however, it is specific for the DEVS methodology and it only supports Time-Warp as a model time synchronization algorithm.

The High Level Architecture (HLA) [5] is a widely accepted framework for promoting interoperability and reusability in the simulation. But it also shows some drawbacks that reduce its expected capabilities when used for the development of distributed simulation applications. The following shortcomings can be identified [6,7]:

1. The responsibility for interoperability between federates (instances of a distributed simulation) in different languages is placed on the RTI (Run Time Infrastructure, the core of HLA) implementers.
2. The federates are tied to specific implementations of the RTI.
3. Different RTI implementations do not interoperate.

This adds a burden to implementing an HLA federation in multiple languages since, in contrast to CORBA imposing absolutely no overhead whatsoever for cross-language compatibility.

D'Ambrogio and Gianni developed a possible solution for the problem, by using CORBA to add further interoperability to HLA [7]. In that solution it was built an *HLA-CORBA Proxy* that is used by the simulation instances to make their HLA calls. The proxies then use CORBA to communicate with a *CORBA-HLA Server* that is placed in the RTI. For each language/system a Proxy must be used. However, the authors recognize that using a CORBA infrastructure to vehicle the HLA requests and responses introduces additional overhead and forbids the use of some communication features, as multicasting and message caching. It is also recognized that the use of CORBA is not CPU-intensive and that further improvements could be achieved by reducing the level of interoperability.

In this article is proposed new a model for discrete-time distributed simulation. The model is language independent and does not restrict to a single scheme of communication or model time synchronization. It enables to incorporate existing simulation programs without significant changes as well as the use real-time programs and computer systems. It also allows the easy exchange of synchronization algorithm in use.

2 A shared memory computation model

Generally, two major models for data exchange can be considered, message passing or shared memory. In shared memory models an application programmer can access variables using ordinary *read* and *write* commands. In message-passing models the programmer needs to keep in mind the architecture of the distributed system, to who and when he is sending the data. Usually shared memory models perform less well than message-passing, because unless the system where they are running uses physical shared memory, in the end the data is sent using messages. Nonetheless, a shared memory model is usually considered to be a more general and easier to use paradigm than a message-passing one [8].

Other advantage of the shared memory model is that in distributed simulations, message-passing models, which are typically mapped to event-triggered simulation algorithms, require extra mechanisms to be integrated in the simulation applications [9,10]. On the other hand, shared memory models need no great change in the simulation methodology. It requires only the definition of which data records are "global", enabling in consequence to use existing model, including real-world programs. It also allows integrating real-time nodes, either physical or virtual. The only modification that must be made is to assure that the simulation will run at a faster or equal speed of the real-time node.

Because of these advantages, it was decided that the data exchange model should be in abstraction, a shared memory model, i.e. a description of shared data, common for all considered communication sub-models, and a set of rules specifying how any change of a single shared data item is communicated among distributed simulation processes.

From a more practical point of view, the data exchange can be seen as passing between different processes as it would be passed locally, using functions like *write* and *read* instead of functions like *send* and *receive*, as in message passing models. In this way the middleware implements shared memory that contains a data-coded image of the state of the whole distributed application.

Because the model is distributed it makes no sense for all the processes to have access to the data from all the other processes, since it could increase tremendously the communication overhead. It was decided that each process should "subscribe" to a set of state shared data items, and only the subscribed set is sent to him, instead of the whole image.

3 A data-coded state of the data exchange model

The created model is composed of several (*n*) logical processes (LP, also known as simulation instances or simulation nodes), in which an LP is assumed to be a conventional run of a simulation application. Each LP_i externalizes its state-like data using one or more data records, say X_i. These records are "visible" from any other LPs. Every LP uses some (global) data records owned by other LPs, say X_3, X_4.

All the set of shared records has n members, i.e. $[X_1, X_n]$. Every record has only and only one owner LP who is responsible to update it. Other LPs can read it. All the names of global data records are known throughout the system, allowing the programmer to access easily records from other LPs.

An abstract description of the general data-exchange model can be defined. It should include a definition of data types, a definition of the shared memory structure, a definition of the distributed simulation application, and a set of simulation control parameters.

Each record has a type, say T_j. The record type can be a primitive type or a composite one. The syntax of CORBA IDL can be used for the definitions of types, including primitive and composite types, such as *structs*, *unions*, and *enums.* Each type, primitive or complex is assigned a type name, within the simulation. In Eq. (1) it is possible to see an example of data type definitions. The primitive types in use are indicated, and the possible composite types are defined.

$$\begin{aligned}&\text{primitive\_types }\{\mathit{short},\mathit{long},...\}\\&\text{composite\_type } T\{..\};\end{aligned}\qquad(1)$$

The definition of the global shared memory structure assigns to each simulation instance X_i a type T_j, as shown in Eq. (2), which is an example of global shared memory structure definition using three types and three state variables, including the composite type *T* created before.

$$\mathit{long}\ X_1;\mathit{short}\ X_2;T\ X_3;\qquad(2)$$

The definition of the distributed simulation application contains information about the different logical processes contained in the global simulation, and the data that they import and export, since the model is subscription-based. In Eq. (3) is shown an example. Each part (process) of the simulation (*simulation_application_part_i*) defines the data that exports (i.e. publishes) and that data that imports (i.e. subscribes).

$$\begin{aligned}&\text{simulation\_application\_part\_1}\{\\&\text{imported\_data}\{X_1,X_2\},\\&\text{exported\_data}\{X_3,X_4\}\};\\&\text{simulation\_application\_part\_n}\{...\};\end{aligned}\qquad(3)$$

The last part of the abstract description is the definition of the control parameters. It contains information about the distributed simulation itself, as well as the kind of synchronization algorithm to use, frequency of updates, as it is possible to observe in the example Eq. (4).

$$\begin{aligned}&\text{synchronization\_algorithm} = \text{time\_stepped};\\&\text{update\_frequency} = 0.2;\end{aligned}\qquad(4)$$

4 A multi-layered approach

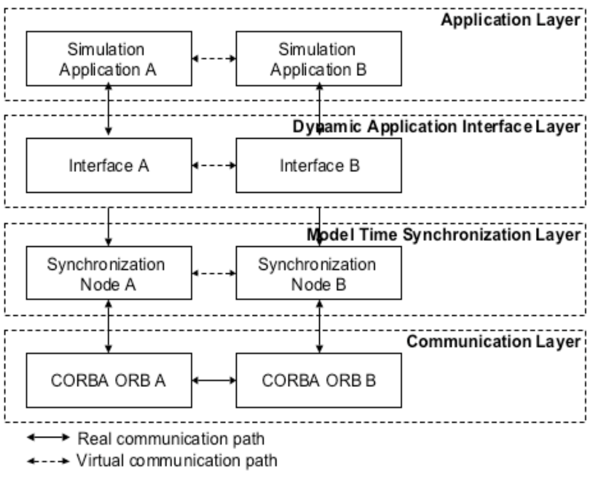


Fig. 1 Logical layers scheme

From the abstract description and the data exchange model is possible to understand that the global model can be viewed as a set of simulation nodes, where each node exchanges data with the other nodes. Starting from that assumption it is possible to decompose each into a set of layers, having each layer its own set of competences. As it is possible to observe in fig. 1, the model is composed of four layers: the application layer, the dynamic application interface layer, the model time synchronization layer and the communication layer.

4.1 Application layer

The application layer is where the simulation application itself lies. The expected computational behaviour is a discrete simulation which proceeds from one time point to another.

In a non-distributed system all the variables are stored locally. In a distributed system it is necessary to update the variables with their correct values throughout the whole system.

The original simulation application should replace the normal data storage with calls to the dynamic application interface layer, using a shared memory abstraction.

Any kind of languages can be used with this application but during its development it was tested mainly with C/C++ and the C-Sim extension [9] as well as with Java and the J-Sim extension [10] will be used.

4.2 Dynamic application interface layer

The layer is responsible for masking the communication and control functions of the model time synchronization layer, presenting a "cleaner" interface that can be used by the simulation application programmer. The interface presented consists of a group of classes/libraries (one for each simulation LP). In fig.2 is possible to see the UML of each node that constitutes the layer.

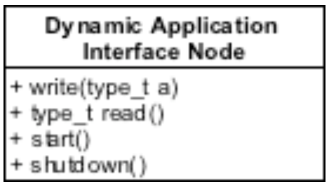
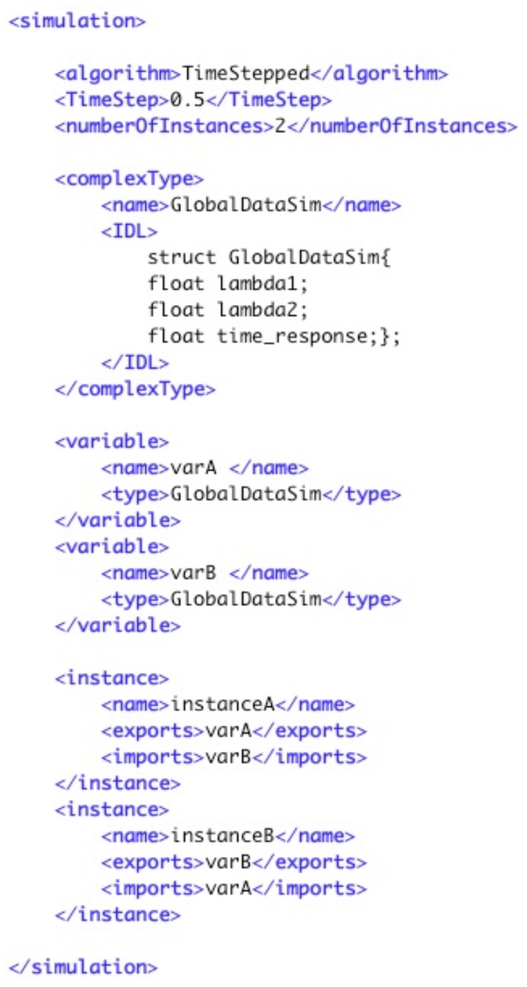


Fig. 2 Dynamic application interface layer UML

The model time synchronization layer requires some information regarding the simulation configuration, as the number of simulation LPs, and other similar control parameters. Also, it only makes available generic methods to update and read data that are based on the CORBA *Any* type. It is clear, that in favour of transparency and ease of use, the configuration of the simulation and the handling of the data encapsulation should be separated from the simulation application programming, and the simulation layer should only "see" an interface that is tailored for its particular necessities, hiding the generic methods made available by the synchronization layer, and its configuration specificities. On the other side, it allows the programmer of the synchronization layer to worry only with the algorithm in itself and to not concern about the data and configuration issues.

This layer can be generated automatically, using a configuration file, written by the simulation application programmer. The configuration file extends the abstract description of the simulation, as so it must include the definition of types, the definition of the shared memory structure, and of the distributed simulation application, as well as the control parameters. The objective of using such a file is to separate the global simulation definitions, from the simulation application normal control flow, making the integration easier. In this way it is possible to tune the simulation outside of the application scope, and to test different algorithms and sets of parameters, and automatically generate modified code that is ready to be used.



```
<simulation>

    <algorithm>TimeStepped</algorithm>
    <TimeStep>0.5</TimeStep>
    <numberOfInstances>2</numberOfInstances>

    <complexType>
        <name>GlobalDataSim</name>
        <IDL>
            struct GlobalDataSim{
            float lambda1;
            float lambda2;
            float time_response;};
        </IDL>
    </complexType>

    <variable>
        <name>varA </name>
        <type>GlobalDataSim</type>
    </variable>
    <variable>
        <name>varB </name>
        <type>GlobalDataSim</type>
    </variable>

    <instance>
        <name>instanceA</name>
        <exports>varA</exports>
        <imports>varB</imports>
    </instance>
    <instance>
        <name>instanceB</name>
        <exports>varB</exports>
        <imports>varA</imports>
    </instance>

</simulation>
```

Fig. 3 example of an XML configuration file

It was decided that XML should be used for the configuration file. The World Wide Web Consortium (W3C) created the Extensible Markup Language (XML), which is in nowadays the standard for the kind of configuration file necessary. It is widely known, and easy to create and understand by both Man and machine. Many XML parsers are available, in many different languages, making it easy to extract the information from the configuration file.

The main problem of the XML approach is that it is difficult to implement complex data types is the XML definition. Because these data types are shared through CORBA, they need to be created in IDL and compiled using an IDL compiler as well. A possible solution would be to create XML structure for definition of complex types, but it seems to be simpler to keep the IDL syntax when defining the data types, and during the processing of the XML file, internally compile the IDL code, and generate the data types. The created code can then be included in the new generated simulation stubs.

As it is possible to observe in fig. 3, the mapping of the shared memory description is quite simple, and the XML easily understandable and modifiable. The only drawback is that the IDL code is included as text, in what is a practical tough not very elegant solution.

A compiler was built, which processes the global configuration file, and generates code that serves as a bridge between the application and synchronization layer. The compiler was developed as a proof of concept so at this time it only compiles for Java and J-Sim languages. But it could easily be transformed to generate code for other languages as well. As part of its processing, the compiler converts IDL data definition code, into Java source code. This is done by simply mapping the data types to Java, when the data types are primitive, using the default mappings defined by the OMG, or by running an IDL compiler, when the types are complex, and then, importing the generated classes, since their resulting syntax is known *a priori*.

The created code must be unique for each application instance, supplying methods to write the state of the variables owned by the specified instance, and methods to read the subscribed variables from the other instances, as well as to control the simulation. The predefined control parameters are also included in the generated code, including the time-step parameter. It is also a responsibility of this layer to know when to update, according to the specified time-step value.

Two sets of classes are generated for each simulation instance. One is the generic class that can be used by any Java program, and the second is a J-Sim specific stun. The main task of the generic classes is to insert and extract the defined data in a CORBA *Any* type, and to send and receive the data updates. The CORBA *Any* type is the best way to send data from one node to the other without knowing exactly what is inside, on compilation time. The insertion in an *Any* can be done automatically for primitive types, but requires access to "helpers" generated by the IDL compiler for complex types. This task is necessary because the underlying model time synchronization layer uses only *Any* types, for the available *write/read* methods.

The J-Sim classes use the generic ones to mask even more the underlying layers. Instead of using write and reads the application programmer needs only to initialize (use the constructor) the given J-Sim object, passing references to the shared variables.

Fig. 4 illustrates the compilation of the configuration file example shown on fig. 3. It is possible to see the CORBA generated classes, the generic classes, and the J-Sim classes.

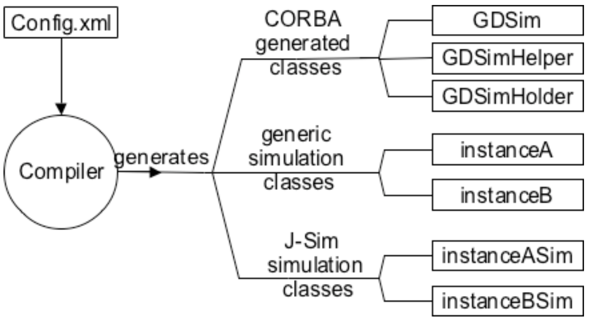


Fig. 4 Example of generated classes

4.3 Model time synchronization layer

The model time synchronization layer main responsibility is to manage the data exchange between different simulation nodes, guaranteeing that a consistent (i.e. synchronized) state image of the whole system is presented to the upper layers.

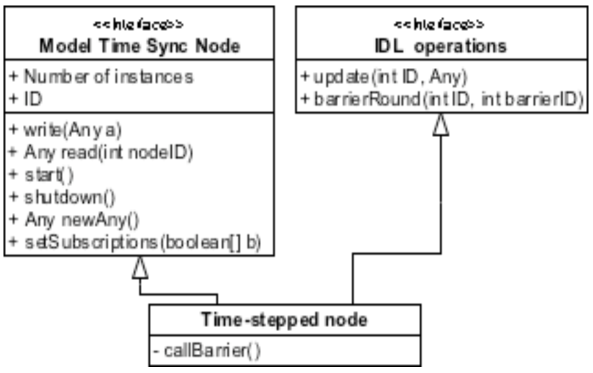


Fig. 5 Interfaces implemented by the model time synchronization layer

This layer uses two interfaces that have to be implemented, as can be seen in fig. 5. On one side is the interface that supplies to the upper layers, a shared-memory interface. This interface contains functions that allow writing and reading of data, and control functions that allow configuring the synchronization behaviour. Any implementation of this layer needs to implement the common shared memory interface model, independently of the underlying model time synchronization algorithm.

The other interface is defined using the CORBA Interface Description Language (IDL). It can change depending on which is the synchronization algorithm in use. What is defined is the internal communication between different synchronization nodes. This duality allows that either time-stepped or event-driven algorithms can be used, using the same shared-memory interface but different internal communication schemes, simply by replacing the whole model time synchronization layer. In figure 5, both the IDL operations interface and the private *callbarrier* method are specific to the time-stepped implementation.

Has it has been said before, simulation applications written in different languages, or running in different operating systems should be able to be used together. The main requisite for this to happen is the porting of the synchronization layer to the desired language. The design of the communication between nodes made in IDL is easy to use, includes many kinds of data structures, including *unions*, *sequences*, *structs*, a generic type, *Any*, and of remote interaction modes. Because all the nodes implement the IDL interface, the porting of the layer is easy.

4.4 Communication layer

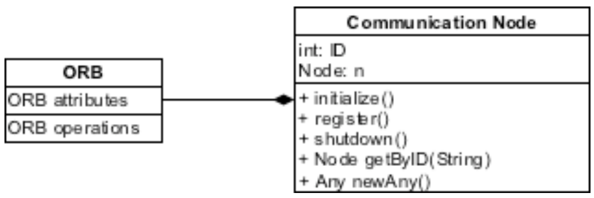


Fig. 6 Communication layer UML

This layer is tightly coupled with the model time synchronization layer. Its main responsibility is to provide a framework for the communication needed by the synchronization layer, and thus by the whole model.

This layer is strongly based on the CORBA communication model. By using the available IDL language and compiler it is possible for the upper model time synchronization layer, to develop its own communication model, and subsequently generate skeletons and stubs that connect single applications with this layer.

As it is possible to observe in fig. 6, it includes methods to initialize an ORB, register and retrieve CORBA objects (in the case, nodes of the model time synchronization layer), and to shutdown the ORB in the end.

5 Using a time-stepped synchronization algorithm

The created model can use different kinds of model time synchronization algorithms, but the algorithm that maps better the shared memory model is a time-stepped algorithm.

5.1 What are time-stepped algorithms?

In a time-stepped simulation all the participating entities in a simulation are at the same time step at any point in wallclock time. Typically the entire span of simulation can be seen as divided into equal-sized time-steps, with the simulation advancing from one time step to the next. Events that happen in the same time step are considered simultaneous and assumed not to have an effect on each other. This is important because it allows actions occurring within each time step to be executed concurrently by different computers. In this paradigm, if two actions have a causal relationship, that has to be modelled in the simulation, which means that, those actions must be simulated at different time steps.

At each i^{th} step, the algorithm simulates all the events that occur in the time interval $[(i-1)\Delta, i\Delta]$, where Δ is a design parameter [13,14]. If Δ is too small, the efficiency of the method degenerates, since barrier synchronization is wasted on intervals that cannot contain any events. On the other side, if the value of Δ is too large, the simulation becomes coarse-grained, as all the events in the interval are simulated as occurring simultaneously. The value given to Δ is important because it determines how precise is the simulation and consequently its results, so, Δ should be chosen large enough so that a LP has several events to process in any given interval $[(i-1)\Delta, i\Delta]$, but not too large or the precision will be affected.

A commonly used technique in time-stepped synchronization algorithms is to use a barrier primitive. When a process invokes the barrier primitive, it will block until all other processes have also invoked the barrier primitive. When the last process invokes the barrier, all processes can proceed further in simulation time. Each time step is separated from the next by using the barrier primitive.

It is possible to use variations of time-stepped algorithms in real-time and scaled real-time simulations. In these cases, the simulation must control advances in its simulation time to be synchronized with the real-time or scaled real-time sub model. In a normal time-stepped simulation, a new system state is computed after each time step. The real-time version is similar, except that the computation for each time step must be computed before wallclock time advances to the next time step or the simulation will lag behind wallclock time.

5.2 Possible applications

There are many kinds of large-scale simulation applications that can be parallelized using a time-stepped approach:

- Large-scale queuing networks, e.g. urban traffic systems. Single parts of a distributed simulation contain models of network parts. Local discrete time flow in parts is synchronized with a global time step and data values influencing another part (e.g. a density of traffic on parts connecting routes) are interchanged among the parts.
- The same for weather simulations and many other similar problems based on numerical solving of partial differential equations. The geographic areas correspond to single parts of distributed simulation model. After a properly chosen global

time step, data values (e.g. wind direction and strength) on a part boundary are exchanged.

- Simulation of population development from system biology: every part of model simulates a separated sub-population of (possibly different) biological objects. Values from one part influencing another part can be imported/exported after every global-time step.
- Logical systems: part models of a system are synchronized after every clock period (global time step as well) and signal values on parts connections are interchanged. Here the time-stepped approach is quite natural.
- A part of distributed simulation system is a real-world part. Time-stepped approach can be used conveniently, assuming we can guarantee that all local simulations (in model time) run faster than real time, so they can be synchronized after a real time tick with real-world parts.

A main advantage of using a time-stepped simulation is that simulation sub-models forming a distributed simulation program (i.e. programs residing in single nodes of distributed system) need not to be changed substantially. On the other side, event-driven programs usually force the application programmer to change the way the program was originally designed.

5.3 Implementing a time-stepped algorithm

In the developed time-stepped model, a time step was defined as a common property of all simulation instances. On each time step global data exchange happens, synchronized with a chosen application-dependent model-time interval. Each single application simulation keeps its own updated state and at each time step, the data updates can be written, and sent to the other simulation applications. For the developed case study the application instance uses C-Sim.

The simulation instances using the tailored methods made available in the dynamic application interface layer. Each simulation instance can use an individual class/library, with methods specifically generated for itself. As it was possible to see in the UML design (fig. 2) there are two kinds of methods available: start and shutdown, to control the simulation; write and read, to pass and obtain the shared data. There can be more than one write or one read available, depending on the definition of shared memory. The time-step value is used in this layer, and the data is only passed to the next layer according to its value.

At the level of the model time synchronization layer, there are available methods to read and write, but using a generic *Any* type. Each call to write contains the instructions to update (send) its variable state throughout the system. The call to the update is surrounded by two barrier calls. The distributed barrier is called firstly with each update called by the application, thus guaranteeing that all the LPs (logical processes) are in the same time step, and again after the update, to ensure that all the updates are over, and that the whole shared memory state is stable.

```
interface TSNode{
    void barrierRound(in long callerID, in long barrierID);
    void updateState(in long callerID, in any state);
};
```

Fig. 7 Time-Stepped IDL

In terms of communication between the nodes of the model time synchronization layer it was necessary to define an IDL communication interface. As it is possible to observe in the fig 7, two remote methods are used: the method *updateState*, which is used to deliver its updates of the variable state, and the auxiliary *barrierRound* method, which is part of the barrier synchronization algorithm.

During the initialization of the synchronization node, the application must indicate to which synchronization nodes its update data will be sent. Then, during each update round, the data is “multicasted” only to the nodes that need it.

Upon completion of the update round, the write calls unblock, and the simulation application instances can proceed to reading the variable states that they subscribed, being that they are already updated and in a stable state in the synchronization node.

When the application instances want to finish their computation, a shutdown call is made. This call also uses the distributed barrier to guarantee that all the nodes have completed their work, and that the simulation can be safely ended.

In fig. 7 is shown the IDL interface used in the internal communication. The interface corresponds to the IDL operations interface shown on fig. 5. An IDL compiler is used to process that IDL code. The IDL compiler generates skeletons that are implemented by the nodes of the model time synchronization layer. By implementing those interfaces they become CORBA objects.

The communication layer makes available generic methods to register and retrieve CORBA objects, and those methods are used to register the model time synchronization nodes.

The nodes are registered in a CORBA name service, which has to be running. There are several methods to locate the CORBA name service, and at the moment the communication layer uses a remote reference file to find the service.

All the layers developed are reusable and can be changed independently of the others. For purpose of testing, the model implementations in C and Java were made.

5.3.1 Choosing a distributed barrier algorithm

The proposed time-stepped synchronization algorithm makes use of a barrier mechanism. This is a mechanism that causes all the processes to wait at a certain point for all the others. The barrier is only broken when all of the processes have arrived. In the first implemented prototype [1], a centralized barrier was used, which made the implementation of the barrier quite easy. But, centralized architectures cause bottlenecks and should not be used. It was decided that a distributed barrier should be used.

In related work [15] a comparison of different barrier implementations was made, and its performance compared. It is possible to observe how centralized algorithms do not scale very well, and that the Dissemination Barrier has a fairly good performance. In another survey [16] several distributed barrier algorithms are examined, and it is concluded that a dissemination barrier is the more appropriated for a kind of distributed architecture like CORBA, which is based on remote-calls, that can be seen as message-passing, rather than a shared memory architecture, despite that the interface provided on a higher abstraction layer is a shared memory one. The used algorithm was originally written as a shared-memory algorithm. As so, some modifications to the original algorithm are required. The implementation of the algorithm both in C and in Java was based on an MPI implementation of the Dissemination Barrier, which was developed in [17].

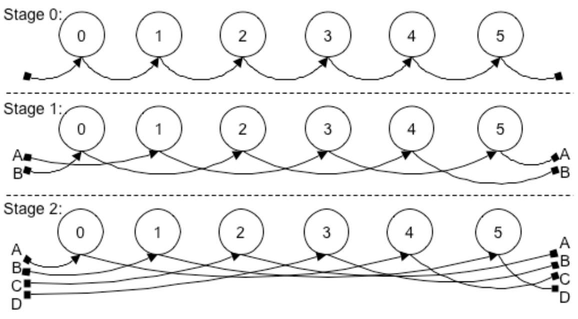


Fig. 8 Dissemination barrier

The Dissemination Barrier, originally introduced by [18] is mostly an improvement of another barrier algorithm, the Butterfly Barrier, for non-power of two process counts. In each round s each process p_i synchronizes with p_j where $j = i + 2^s \bmod p$. Each process is waiting for the cyclically next to set its flag and for his own flag set by a circular previous process. The algorithm is similar to the one used in the Butterfly Barrier but with different partners. While in the Dissemination algorithm the synchronization occurs in a circular way, in the Butterfly algorithm each process A synchronizes with B, and B with A, changing the pairs in each round. As a result of these modifications, the Dissemination Barrier improves the Butterfly Barrier, and has $O[\log_2(N)]$ (being N the number of processes) concurrent network transactions.

Figure 5 shows an example of synchronization using a dissemination barrier.

6 Case study: queuing networks

6.1 Testing application

A testing application was constructed in order to verify a level of usefulness of the presented distributed simulation computational model and methodology. The application serves like a benchmark aimed to test as many designed concept properties as possible.

The chosen application was a queuing network example contained as a part of the C-Sim (OQN example) distribution package. This queuing network consists of two *n*-channel serving nodes with n-channel nodes and infinite FIFO queue. The network has two input streams of transactions and two output streams. The structure of the network is depicted at figure 9.

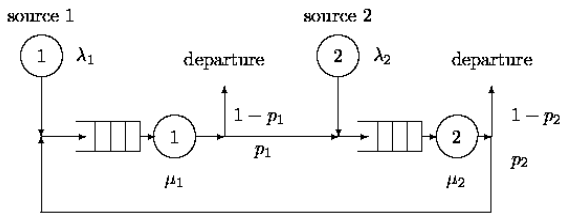


Fig. 9 Queuing network

Assuming exponential pdf of interarrival time of transactions within input streams as well as exponential distribution of serving times within the servers and channels, the queuing network can be solved analytically. Using λ_1, λ_2, μ_1 and μ_2 as the model parameters, we can obtain numerical results like mean frequency of internal streams, mean number of transactions within the system, mean time that a transaction stays within the system, and others.

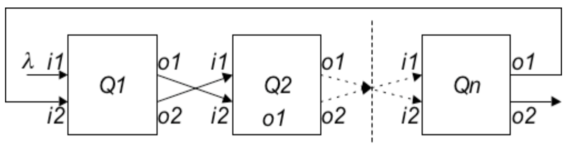


Fig. 10 Queuing networks scheme

The application should be generic and scalable. These properties can be reached by connecting number *N* instances of the given network in a cascade. i.e. outputs of the *i*th member of the cascade are connected to the inputs of the *i*th*+1* member. The final structure of the modelled queuing networks is illustrated by fig. 10.

It is possible to observe that output 1 (*o1*) of each queuing network Q connects to the input 2 (*i2*) of the next queuing network, the same for output 2 (*o2*) of the first, and input 1 (*i1*) of the next. Output 1 of the last member is connected to the input 2 of the first member. The input 1 of the first member has firmly set λ input rate (stream of transactions from the modelled system environment) and output 2 of the last member goes outside (i.e. it models a stream of transactions leaving the modelled system).

This kind of network can be used, for example, as a model of transportation system like a system of roads and crossings within a large town, transactions are abstract models of cars and single instances of the basic queuing network corresponds to parts (i.e. suburbs) of the town transportation system.

The entire generic network can still be solved analytically (i.e. mathematical solution can be obtained for every *n*), so there is the possibility to check a precision of simulation results.

Also, it enables the measurement of performance parameters, especially the speedup when we split a large simulation model into several communicating instances.

6.2 Organization of the tests

The queuing network model was written in C/C++ (the original examples from the standard distribution of the C-Sim simulation package have been slightly modified for the given purpose) representing a single instance involved within the distributed simulation program. The distributed simulation program is composed from *N* communicating instances.

Communication is transparent from the side of single instance simulation program, i.e. the communication is provided by the underlying middleware layers, what means that no great modifications were need on the original simulation program.

Single instances from the distributed simulation composition exports three simple data items:

- λ_1 - Mean frequency of the first input stream
- λ_2 - Mean frequency of the second input stream
- Mean response time - time taken by a transaction since it arrives at the queuing network until it leaves.

The data exported by each instance *i* is then imported by the *i+1* instance (following instance in the cascade of instances), so the exported data record has the same structure as the imported data record. These data items are updated within the simulation instance with a chosen period. Internally actualized data are periodically exchanged among simulation instances within the time step synchronization action.

It is possible to observe in figure 3 an example of the XML file used for definition of the simulation global data exchange. The case shown uses two instances.

As a global result of the simulation, the mean time response (i.e. the mean passing time of transactions) for every member of the network cascade (sub-network) is computed.

The tests were run in a single computer, using two versions of the test program. In the first one, communication between the instances is simulated (i.e. the data exchange is done by simple assignment), being this test equivalent to running the simulation on a single computer, i.e. non-distributed version of simulation. The second version uses the developed CORBA framework for communications, in a single computer, in a concurrent way.

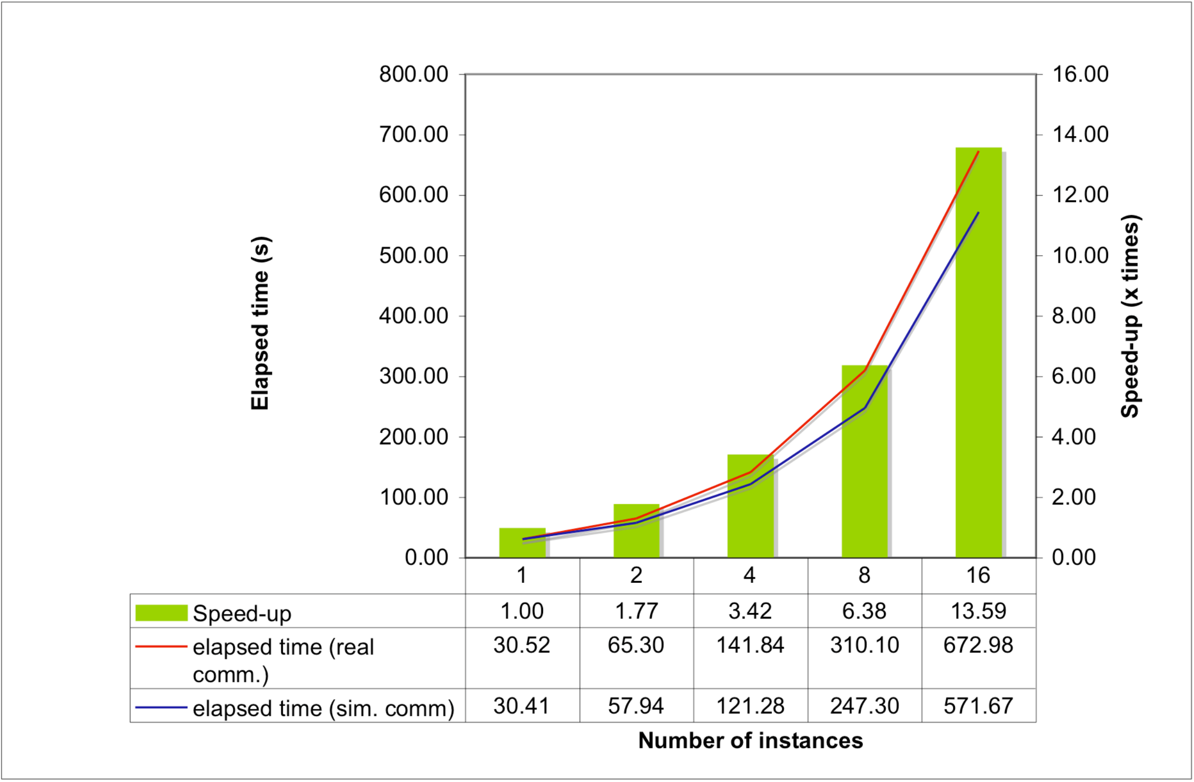
The tests were intended to measure two parameters. The first test was made to measure the communication overhead and find out whether a speed-up happens and how that speed-up would vary. For this first test runs with both versions were executed using a fixed time-step of 5000 and varying the number *N* of simulation instances.

The second parameter that it was intended to test was the precision of results compared with the simulated ones. Runs of both versions were made, using a fixed number of instances (in the case, four), and by varying the time-step. By measuring the final average mean time response according to different time-steps it is possible to evaluate the precision obtained, and also which is the best time-step for the chosen model.

The numerical parameters of the model were set in a way, that the mean value of every lambda within the network should be 0.4 (i.e. 0.4 passing transactions per model-time unit in average) and the time response (i.e. mean passing time of transactions) for every sub-network then should be 10.0.

In both test the maximum simulation time was set to one million units of the model time.

6.3 Results



	1	2	4	8	16
🟩 Speed-up	1.00	1.77	3.42	6.38	13.59
elapsed time (real comm.)	30.52	65.30	141.84	310.10	672.98
elapsed time (sim. comm)	30.41	57.94	121.28	247.30	571.67

Fig. 11 Chart illustrating speed-up and duration of simulation depending on the number of instances

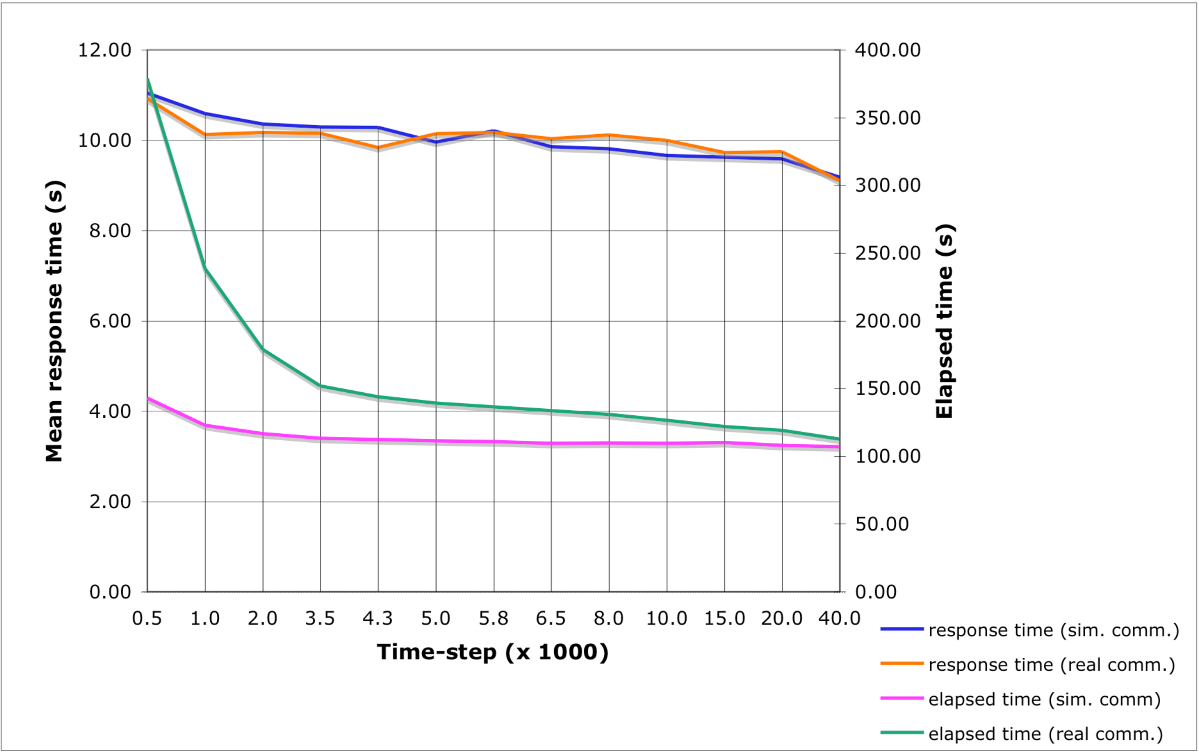


Fig. 12 Chart illustrating precision and elapsed time depending on value of time-step

6.3.1 Speed-up

In fig. 11 it is possible to observe the elapsed time for the version with simulated communication and for the version with real communication, using the developed framework.

It is possible to observe that the difference between the duration of the version with simulated communication and of the version with real communication is not too big. It is expected to be smaller when the computation ran in a network, since most of the communication would be parallel and not serialized as when it is run concurrently on a single computer.

Speed-up was calculated as $n\frac{t_{sim}}{t_{real}}$, being n the number of instances. It is possible to observe in the table in figure 11 that the speed-up achieved grows almost linearly according to the number n of instances.

6.3.2 Precision and time-step

In the fig. 12 it is possible to observe how the average mean time response of the simulation instances varies according to the chosen time-step. The theoretical value of the mean time response computed from side-staying mathematical model is exactly 10.0.

The time-step values that yield the best precision in the version with simulated communication were the values of 5000 and 6500 (value of time-step simulation time). For the version with real communication the values were not so clear as with the simulated communication but the best time-step tested was about 6500.

The result puts the version with distributed communication in a similar level of precision as the version with simulated communication.

It is also noticeable that very small values of the time-step (<2000) are prohibitive, since both the elapsed time and the precision are not good. For too big values of the time-step (>10000) the simulation runs faster but the precision is lost. There is a trade-off between performance (smaller elapsed time) and precision: except for very small time-step values, a shorter time step means better performance but worse precision

7 Conclusion

The proposed general programming model for discrete-time distributed simulation has several important advantages:

The developed multi-layered scheme allows integrating single simulations in an easy and transparent way. It also makes possible easily exchanging or modifying the synchronization model in use. It brings the possibilities of testing multiple kinds of synchronization algorithms. Moreover, allows the model to be not only a platform to build distributed simulation program but also a basis of a framework aimed to develop and study synchronization algorithms.

The fact that all the distributed simulation data exchange configuration can be specified in one place, and allowing the generation of tailored code ready to be used by the application ready to be integrated, simplifies in a great way the creation of a complex distributed simulation program.

Several advantages come from using the CORBA communication model. It gives the model the possibility that different simulations, written in different languages, and running in different systems can easily be integrated, thus achieving inter-operability. Another advantage of the developed model is its reusability, once the model is implemented in one programming language, it can be used without modifications of the communication framework.

The tests conducted shows that the developed model allows the parallelization of a simulation application with significant gains in performance, and without loss of precision, namely when the simpler (time-stepped) algorithm of model-time synchronization is utilized.

8 References

[1] J. Duarte. Parallel Simulation using CORBA: A Feasibility Study. *Proceedings of Modelling, Identification, and Control*. Innsbruck. Austria. 2007.

[2] C. Shen, A CORBA Facility for Network Simulation. *Proceedings of the 1996 Winter Simulation Conference.* 1996

[3] A. Cholkar, P. Koopman. A widely deployable web-based network simulation framework using CORBA IDL-based APIs. *Proceedings of the 1999 Winter Simulation Conference.* 1999.

[4] Ki-Hyung Kim, Won-Seok Kang. CORBA-Based, Multi-threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One. *Computational Science and Its Applications - ICCSA 2004*.167-176. 2004.

[5] J. Dahmann, R. Fujimoto, and R. Weatherly. The Department of Defense High Level Architecture. *WSC '97: Proceedings of the 29th conference on winter simulation*. Atlanta, Georgia, United States.1997.

[6] A. Buss, L. Jackson. Distributed Simulation Modeling: A Comparison of HLA, CORBA and RMI, *Proceedings of the 1998 Winter Simulation Conference.* 1998.

[7] A D'Ambrogio, D. Gianni. Using CORBA to enhance HLA interoperability in distributed and web-based simulation, *LNCS vol. 3280/2004, Proceedings of the 19th International Symposium on Computer and Information Sciences (ISCIS'04)*, Antalya, Turkey.2004.

[8] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. *Proceedings of SuperComputing '95*.1995.

[9] M Marin, R. Miranda, and A. Alvarado. Parallel Discrete-Event Simulation Framework.*Proceedings of the XXIII International Conference of the Chilean Computer Science Society.* Chile. 2003.

[10] R. M. Fujimoto. *Parallel and Distributed Simulation Systems,* Wiley-Interscience, ISBN 0471183830) .2000.

[11] http://www.c-sim.zcu.cz/

[12] http://www.j-sim.zcu.cz/

[13] S. Eick, A. Greenberg, B. Lubachevsky and A. Weiss. Synchronous relaxation for parallel simulations with applications to circuit-switched networks. *ACM Trans. Model. Comput. Simul.* vol. 3,issue 4*,* 287-314. 1993.

[14] S.Tay , G. Tan, and K. Shenoy. Piggy-backed time-stepped simulation with 'super-stepping'. *Proceedings of the 2003 Winter Simulation Conference*. Vol. 2. 1077- 1085. 2003.

[15] C. Ball, M. Bull.Barrier Synchronization in Java. *Technical report.* UKHEC. United Kingdom. 2003.

[16] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. A Survey of Barrier Algorithms for Coarse Grained Supercomputers. *Chemnitzer Informatik Berichte*. Technical University of Chemnitz. Vol 04, Nr. 03,ISSN: 0947-5152.Germany.2004

[17] T. Hoefler, A. Lumsdaine. Design, Implementation, and Usage of LibNBC. *Technical Report.* Open Systems Lab, School of Informatics, Indiana University. United States.2006.

[18] D. Hengsen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, Vol. 17, Issue 1, 1998.