

EXPERIMENTING WITH DISTRIBUTED MODELING AND SIMULATION USING THE INTERNET.

Jaap A. Ottjes, Hans P.M. Veeke, Arnoud A. Buizer
Sub Faculty of Mechanical Engineering and Marine Technology, Fac. OCP
Delft University of Technology
Mekelweg 2, 2628 CD Delft, the Netherlands
e-mail: J.A.Ottjes@wbmt.tudelft.nl , H.P.M.Veeke@wbmt.tudelft.nl

KEYWORDS

Discrete simulation, process-oriented simulation, model testing, transportation.

ABSTRACT

Traditionally, most applied discrete simulation models are still stand-alone models running on a single computer and using a single sequencing mechanism. In this paper we focus on distributed modeling and the consequences of its use on the time sequencing and interactions between distributed models. As an example a stand-alone model and two different distributed models of a simple shipping system are designed. The important steps and some remaining questions are discussed. This modeling effort was part of the development and verification of the simulation package TOMAS, which fully supports distributed modeling and simulation. The process-oriented modeling approach provided the basis for this. The TOMAS package has been made available on the Internet.

INTRODUCTION

Many real systems are distributed in nature. Important reasons for distributed modeling include the possibilities it provides to apply distributed control, to create aggregated levels and appropriate zooming functionality and the opportunity for the distributed development and execution of models. In this paper some simple models that are being used to develop and test the concepts and the implementation of a distributed modeling and simulation tool TOMAS are described. This development is a follow-up of the simulation software which we have been using many years in both educational and professional practice for the modeling and simulation of the large industrial systems found in the production and transportation industries (Veeke 1982, Duinkerken et al. 1999, Duinkerken and Ottjes, 2000). The modeling has always followed the process- interaction simulation strategy (Zeigler et al. 2000) that was first used in the program language "Simula", which was already object oriented (Birtwistle and Dahl, 1973). This process oriented approach is still being applied in simulation

tools (Crain, 1996) and new software, supporting process modeling, is being launched (Healy, 1997).

Important demands that we impose on our simulation software are: the language should support process-oriented modeling and should be object-oriented, it must be lean, fast, flexible, portable, easy to learn and low cost. It should allow the creation and maintenance of re-usable objects. Delphi was chosen as the basic developing language since it fulfils a number of the above mentioned demands (Veeke, Ottjes 1999). The main effort has been focussed on developing a fast event driven sequencing mechanism to support both process modeling and distributed modeling. After the completion of a object- based process-oriented tool for stand-alone modeling (Veeke, Ottjes, 2000), distributed functionality was added. An important reason for that was to be able to reuse control modules designed and tested with simulation for the control of real objects such as automated guided vehicles, (Ottjes , Hogedoom, 1996). Using existing solutions for distributed modeling, especially HLA (Fujimoto, 2000, Rabe, 2000) would have been the most obvious way. Apart from the advantages, using HLA would however take a lot of extra effort next to the usual simulation efforts (Klein, Strassburger, 1998). Considering that we only need a small subset of all HLA possibilities, it was decided that a basic distributed concept should be obtained by simply using standard Windows facilities in combination with the TCP/IP protocol, allowing communication over the Internet. The concept is further elaborated in (Veeke, Ottjes, 2001).

First we consider some aspects of process-oriented modeling and define pseudo language for important simulation commands. Then a simple shipping model is used to illustrate distributed modeling and some of the implications and new possibilities it provides.

The models presented here are implemented in the simulation package TOMAS (Tool for Object-oriented Modeling And Simulation), based on Delphi or C++ (Veeke, Ottjes, 1999). The package, including most of the source codes, is freely available on www.tomasweb.com.

PROCESS ORIENTED MODELING.

In practice, there are two approaches that are designated as 'process-interacting modeling' (Zeigler et al. 2000). One approach can be characterized as focussing on the elements which are flowing through the system (Robert and Dessouky, 1998). For each type of flow element the path through the system is described, including the claiming of resources such as machines. The second approach, following the Simula concept and used in this work, focuses on the processes of the resources. Provided that the real system has been analyzed thoroughly and that the goals of the simulation project have been agreed, there are two steps in the design of the model:

Step 1: decompose the system into relevant classes of elements, preferably patterned on the real-world elements of the system. A class is characterized by its attributes. The state of each instance of a class is defined by the state or value of its attributes. Methods ascribed to a class are also considered to be attributes. An instance of a class will be called an element. **Step 2:** distinguish the "living" element-classes and provide their process descriptions. A process description governs the dynamic behavior of each instance of the element class.

A process defines the dynamic behavior of an element. Two types of process-activities are distinguished: Activities which consume no simulation time, for example the determination of the fastest route for a ship, and activities consuming simulation time, for example the actual sailing of the route by the ship. For the description of a model in the design stage we use pseudo-language. We consider this informal modeling stage as an essential feature of a simulation project. A model in pseudo-language can be produced more quickly and is accessible to a broader audience, especially the "problem owners". Because in most cases model validation appears to be difficult, the design of an informal model in collaboration with the problem-owner provides a possible opportunity for structural validation. The translation from informal model into formal code should be straightforward, with a minimal chance of errors in interpretation putting some very specific demands on the simulation software used.

We use a pseudo-code that is as close as possible to the formal implementation in program code. In the process description we use "advance t" to indicate that an element needs t time units to carry out an activity. If such an 'advance-statement' is encountered in the process description, the processing of the particular element halts until time t has elapsed and then continues. In other words the process is waiting for a specific time event. The continuation of a process has to be automatically controlled by the sequencing mechanism. Analogous to that, it is possible for a process to wait for a 'state event' e.g. for a specific condition to be fulfilled. In pseudo-code this is written as: "advance while/until condition". At the moment that the while condition becomes false or the until condition becomes true, the process is automatically resumed.

If "advance" is encountered without any additional parameter, the element becomes passive and can only leave this state and proceed with its process with the help of another element which reactivates it. Reactivation of an element is indicated by "resume element". A process is activated for the first time by "start". The "advance" clause may have aliases that fit into the model context, such as wait, work, hold and standby. We restrict ourselves to "advance". A process which is in an advance state may be interrupted with the "cancel" command originating from another process. The cancelled process may be continued with the "resume" command. "tNow" is used to indicate the current time in the simulated system.

The sequencing mechanism, necessary to synchronize the activities and to manage the event calendar, must be supplied by the simulation package used.

Additional features are "queues" or "sets", which may contain elements and, in the case of stochastic behavior, "distributions". Queues, sets, and distributions and also user defined methods may be used as attributes of element classes. It is good practice to express the function of an attribute by its name. If relevant, the working of a method may be explained separately. A self repeating process starts with "loop". If an element has finished its process this is indicated with "finish". For qualifying of attributes the "dot" notation is used. Example: *myPort.shipQ.first* means the first element in the shipQ of myPort.

Next we will describe some models of a system consisting of an arbitrary number of ports and ships. The ships are sailing between the ports and are handled by quay cranes. First a model will be elaborated as a stand-alone model and after that two distributed models will be defined. These models were originally designed to test and evaluate our distributed modeling software.

A STAND-ALONE MODEL OF A SYSTEM OF PORTS

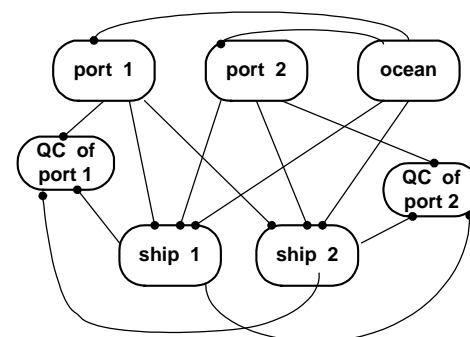


Figure 1. Stand-alone model with two ports each with one crane, two ships and an ocean in stand-alone model

The model contains a number of ports that are instances of the class 'Port' and one instance of the class 'Ocean'. Each port

generates one or more instances of the class 'ship' which are going to sail according the 'free trade' principle. The ships are generated when the port is created. If a ship arrives in a port it is moored and then waits to be handled. After that it leaves the port and its next destination is randomly chosen from all other connected ports. Figure 1 shows the elements and their relationships in a model with two ports and two ships. Only the classes 'ship' and 'quayCrane' (QC) own a process. The classes, their attributes and the processes are listed in pseudo-code in Figure 2. Elements with attributes referring to another element 'know' the class of this element and can refer to any instance of it.

```
Stand alone SHIPPING MODEL
Classes, attributes, processes

Class 'Port'
□ ShipQ: queue with ships
□ MyOcean
□ CraneQ
□ GetMooringTime: function

Class 'Ocean'
□ ShipsSailingQ
□ PortList

Class 'QuayCrane'
□ ShipInHand: refers to ship
□ MyPort: refers to port
□ GetHandlingtime: function
PROCESS
Loop
1. Advance while myPort. shipQ is empty
2. ShipInHand = myPort. shipQ.first
3. Remove shipInHand from myPort. shipQ
4. Advance getHandlingtime
5. ShipInHand.resume

Class 'Ship'
□ Destination: refers to port
□ MooringTime
□ GetNextPort: function
□ GetSailingTime: function
PROCESS
Loop:
1. MooringTime = destination.getMooringTime
2. Advance mooringTime
3. Enter destination.shipQ
4. Advance
5. Enter ocean.shipsSailingQ
6. Destination = getNextPort
7. Advance getSailingTime
8. Leave ocean.shipsSailingQ
```

Figure 2. The stand-alone shipping model in pseudo code.

Time sequencing

In the stand-alone model one time sequencing mechanism is operating. Every "advance" statement automatically results in the generation of an event that is automatically processed by the sequencing mechanism.

Initializing and running the model

A minimal initial configuration consisting of only one port and the ocean are sufficient to run the model. More ports may be initialized at the start of the simulation run or during its execution, for example via the model user interface. All participating ports are contained in the portList of the 'ocean'.

DISTRIBUTED MODEL 1: PORTS AND OCEAN

Now we are going to elaborate the consequences of splitting the stand-alone model into several parts called "member models". We want each port to operate in an autonomous 'Port' model. The 'Ocean' model is to be a separate model containing ships while sailing. In the HLA terminology 'port' and 'ocean' would be called 'federates' and together they form a 'federation' (Fujimoto 2000). Furthermore it should be possible to generate and terminate any port at any time, provided that at least one port stays connected. If a port is 'terminated' then its ship should remain in the system.

We distinguish between two types of models which may be member models of the distributed model: the 'Port' model, containing a port, one or more cranes and ships in port and the 'Ocean' model containing ships that are sailing. Port models and the Ocean model form the "distributed model". Any Port model and the 'Ocean' may run on any 'Windows computer', provided that it is connected to the Internet.

Process synchronization

Every model has its own local event based sequencing mechanism like that used in the stand-alone model. In order to ensure repeatability no "look ahead" features are used (Fujimoto 1997). A time-server model is introduced to synchronize the simulation time of all connected models. It will be referred to as the 'server'. The server operates on an arbitrary computer with an Internet connection, controls the global model time and passes information to and from member models. The server time management is conservative. In fact it acts in a way identical to that of the stand-alone model synchronization, maintaining a global event list. From the moment a member model connects to the server it subjects itself to the server and sends its first local time stamped event to the server. All local sequencing mechanisms remain active but subordinate to the server. The server 'knows' the first event of each of the member models and uses this information to manage the global simulation time, taking into account the order of simultaneous events. The server permits models to proceed and notifies all models in the case of changes in global time.

Figure 3 shows the set of distributed models schematically. All models may exchange messages with the server model.

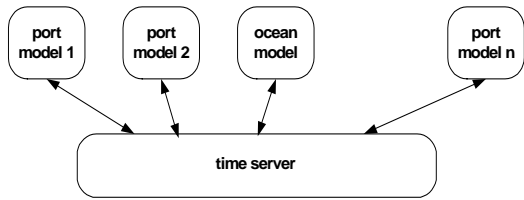


Figure 3. Distributed model 1. The exchange of messages is indicated with double arrows.

In Figures 4a and 4b the member models are shown in pseudo code.

```

MODEL: PORTMODEL
ReceiveShip: procedure
1. create instance of class ship
2. start this instance

Class: Port
□ craneQ
□ shipQ: queue with ships
□ getMooringTime: function

Class: quayCrane
□ shipInHand: refers to ship in port
□ myPort
□ getHandlingtime: function
PROCESS
Loop
1. advance while myPort. shipQ is empty
2. shipInHand = myPort. shipQ.first
3. remove shipInHand from myPort. shipQ
4. advance getHandlingtime
5. shipInHand.resume

Class: Ship (in port)
□ thisPort
□ mooringTime
□ transferToModel(mName): procedure
1. Send message with relevant information to model
with name "mName"
PROCESS
1. mooringTime = thisPort.getMooringTime
2. advance mooringTime
3. enter destination.shipQ
4. advance
5. transferToModel(oceanModel)
6. finish
    
```

Figure 4a. The "portModel" as a member of the distributed model 1 in pseudo code.

Model interactions

In order to manage a set of synchronized simulation models, interaction between models and server and mutual model interaction is needed. In the current implementation of TOMAS this is obtained by exchanging messages between models and server using the TCP/IP protocol and standard Windows Messaging facilities.

```

Model: OCEANMODEL
    
```

```

□ receiveShip: procedure

Class: Ocean
□ shipsSailingQ

Class: Ship (at 'Ocean')
□ portList: name list of Portmodels
□ destination: Portmodel name
□ updatePortlist: procedure
□ getNextPort: function
□ getSailingTime: function
□ transferToModel(mName): procedure
PROCESS
1. enter ocean. shipsSailingQ
2. updatePortlist
3. destination = getNextPort
4. advance getSailingTime(destination)
5. transferToModel(destination)
6. leave ocean.shipsSailingQ
7. finish
    
```

Figure 4b. The "oceanModel" as a member of the distributed model 1 in pseudo code.

Messages may be exchanged synchronously if the models use the same time-base or asynchronously if time synchronization is not relevant. The sending model has to code its information into a message and the receiving model has to decode it and interpret the message, so a proper communication set must be defined. Example: The *transferToModel* method of the ship in figures 4a and 4b sends a message containing the model name of the destination and the coded specific information needed, for example cargo information. This will be further illustrated in tables 1 and 2. We distinguish two types of interaction:

- autonomous interactions
- user defined interactions

Autonomous interactions

Autonomous interactions take care of the right sequencing of events in the participating models and consist of messages exchanged between the server and the models. For example an "advance" clause causes a message containing the model name and event time to be sent from the model to the server. When that event time is reached the server sends a message back to the model, allowing it to resume. Autonomous interactions are generated and interpreted automatically without interference by the user (modeler).

User designed interactions

User-designed interactions concern all model- specific interactions. These may vary from asking the server for a list of participating models to passing elements with attributes between models. In this model a ship is distributed over two models. Both the port model and the ocean model contain ships. In our implementation elements, which are virtually transferred to another model, pass the relevant information to that model. In the example a ship, which has to transfer from a

port to the ocean, sends its ID (name) and its cargo data, coded in a message, to the ocean model. In the ocean model this message is decoded and interpreted, resulting in the creation or re-creation and activation of the ship with additional attributes to manage sailing. A ship that transfers from the ocean to a port passes on its cargo and its name. In the port model the ship is re-created with the addition of an attribute 'mooringTime', which is calculated according specific port data and the cargo. Both models need the proper element definition in order to be able to reconstruct the ship. The ship process is also distributed over the two models, so both element definitions contain part of the ship-process. The element definitions have to be agreed on by the modelers and provided in the right model.

Initialization and running

As in the stand alone model, the distributed model may work if at least one port model and the ocean model and, of course, the server are running. Other port models can be started from any location from any computer connected to the Internet. If a port model, "gets rid of" the ships in its shipQ before disconnecting from the server, these ships will remain in the system.

However, this is not a very logical way to model. The ship, with its attributes and methods, has to transfer from one model to another. In fact these attributes and methods, including the process, may differ in different models. It makes more sense and it is more appropriate to the process-oriented approach to keep the ship in its own model. Only the relevant data elements have to be transferred between member models then. This is done in the second distributed model.

DISTRIBUTED MODEL 2: PORTS AND SHIPS

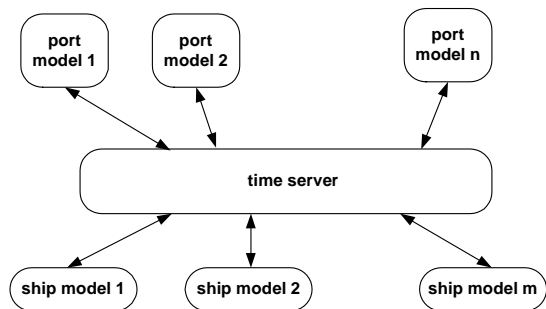


Figure 5. Distributed model with ports and cranes in port models and ships in ship models. The double arrows represent the exchange of messages between server and model.

In this model only distributed models of ports and ships appear. The Ocean as a "container" for sailing ships has been left out. Ports as well as ships have to connect to the server model after they have been created. Every time a ship reaches a port it asks this port, via the server, for its mooring time and advances that time. After mooring, the ship sends its ID and cargo data to the port model (notifyPortModel). After receiving this information,

the port model creates a job for its crane. The job contains the ship ID and the cargo as attributes. Later the ship ID is used to direct a remote resume message to the ship after the job is done and the ship has to depart.

```

Model: PORT MODEL
□ receiveJob: procedure
1. create Job and set attributes ship and cargo
2. put job in port.jobQ

Class: Port
□ craneQ
□ jobQ: queue with jobs

Class: Job
□ ship refers to shipModel
□ cargo

Class: QuayCrane
□ jobInHand refers to job
□ myPort
□ getHandlingtime: function
□ notifyShipModel(mName): procedure
PROCESS
Loop
1. advance while myPort.jobQ is empty
2. jobInHand = myPort.jobQ.first
3. remove jobInHand from myPort.jobQ
4. advance getHandlingtime
5. notifyShipModel(jobInHand.ship)
  
```

Figure 6a. The "portModel" as a member of the distributed model 2 in pseudo code.

```

Model: SHIP MODEL
□ resumeProcess: procedure

Class: Ship
□ portList list =name list of current ports
□ destination refers to port
□ mooringTime
□ getMooringTime: procedure
□ updatePortlist: procedure
□ getNextPort: function
□ getSailingTime: function
□ notifyPortModel: procedure
PROCESS
Loop
1. updatePortlist(timeServer)
2. destination = getNextPort
3. advance getSailingTime
4. mooringTime= getMooringTime(destination)
5. advance mooringTime
6. notifyPortModel(destination)
7. advance
  
```

Figure 6b. The "shipModel" as a member of the distributed model 2 in pseudo code.

Again it should be possible for models (port and ship) to connect to and to disconnect from the server at any time from any location. Figure 5 shows the model structure. In Figures 6a and 6b the port and ship class definitions and processes are shown in pseudo code.

Model interactions

In this example there are several user-defined messages and interpretations. Every time a ship has to decide about its next port, it has to know which ports are still connected. In the stand-alone model a ship only has to look into the portList of the ocean. In the distributed case only the server knows all current models, so a ship asks the server for a list of models called 'clients' in the server (updatePortList) see also Table 1. To this end, standard communication is provided via the 'sendMessageToServer' routine. If a ship has moored it passed its relevant information, coded in a message, to the port: notifyPortmodel, passing its ID and cargo information. The Port model receives that message and calls its receiveJob procedure to create a job with proper attributes.

Initializing and running

The distributed model 2 works if the server runs and one Portmodel and one shipmodel are connected. Additional ship models and port models may connect from any computer and may disconnect at any time.

Practical implementation

To connect to the server a member model only needs to know the IP address of the computer on which the server is running. In TOMAS this IP address has a default value (local host; all models run on the same computer) and may be changed via the user- interface or read from a configuration file.

TESTS

The models described are implemented in TOMAS. It appears that the architecture as described is working properly on single computers (server address = local host), on local networks and via the Internet operating from several service providers.

In Figure 7 the sever log of an experiment with distributed model 1 is shown serving some port models that are running. Table 1 shows part of the trace output of the Ocean model and table 2 shows part of the trace of one of the member models (Rotterdam) during a run with distributed model 1. In table 1 Ship_ANTWERP is at the Ocean and is going to choose the next Port. Lines 7-12 show the list of connected models, the server has send to this OceanModel after a request of Ship_ANTWERP. The message is preceded by \$\$clients, followed by the total number of connected clients (member models), in this case 5, the OceanModel included. The Ship_ANTWERP chooses ROTTERDAM (line 16) and finishes its process in the OceanModel at time 47.80. At that time the Ship-ANTWERP is re-created in PortModel Rotterdam. In table 2 lines 14-16 show the coded message that was send by Ship_ANTWERP from the OceanModel. The \$\$from clause is followed by the model name of the sender,

then the specific message that contains the ship name, its cargo "hallo from ANTWERP" and the time the ship is supposed to be re-created in the receiving model.

Table 1. Part of Trace OceanModel

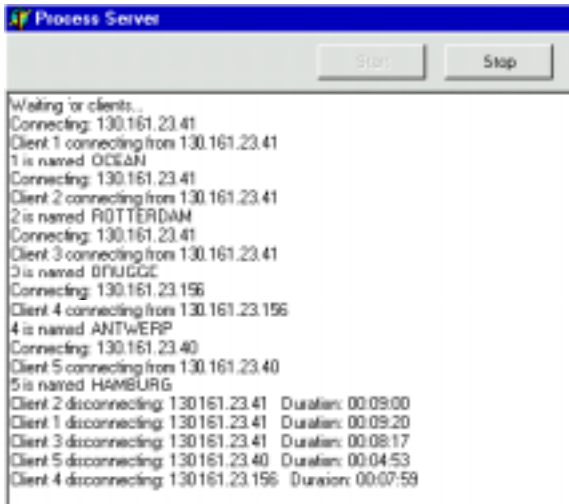
Line Nr	Trace text
1.
2.	Ship_ROTTERDAM transfers to ROTTERDAM
3.	47.00 Ship_ROTTERDAM finished
4.	47.80 Ship_ANTWERP is current now
5.	Ship_ANTWERP asks Server for update portList
6.	INCOMING Message in OCEAN
7.	\$\$clients:5
8.	!!HAMBURG!!5!5-2-2001 10:39:01
9.	!!ANTWERP!!4!5-2-2001 10:37:15
10.	!!BRUGGE!!3!5-2-2001 10:35:01
11.	!!ROTTERDAM!!2!5-2-2001 10:34:11
12.	!!OCEAN!!1!5-2-2001 10:33:54
13.	47.80 Ship_ANTWERP is current now
14.	47.80 Ship_ANTWERP out of Sailing ShipsQ
15.	Ship_ANTWERP proceeds after respons clients
16.	Ship_ANTWERP transfers to ROTTERDAM
17.	47.80 Ship_ANTWERP finished
18.	INCOMING Message in OCEAN
19.	\$\$from:HAMBURG
20.	!!Ship_HAMBURG!!hallo from HAMBURG!!47.80
21.	47.80 Ship_HAMBURG created
22.	47.80 Ship_HAMBURG is current now
23.	47.80 Ship_HAMBURG to tail of Sailing ShipsQ
24.	Ship_HAMBURG Now at Ocean model
25.	47.80 Ship_HAMBURG holds until 49.00
26.	48.00 Ship_BRUGGE is current now
27.	Ship_BRUGGE asks Server for update portList
28.	...

Table 2. Part of Trace PortModel ROTTERDAM

Line Nr	Trace text
1.
2.	INCOMING Message in ROTTERDAM
3.	\$\$from:OCEAN
4.	!!Ship_ROTTERDAM!!hallo from ROTTERDAM!!47.00
5.	47.00 Ship_ROTTERDAM created
6.	47.00 Ship_ROTTERDAM is current now
7.	47.00 Ship_ROTTERDAM out of ShipsToHandleQ
8.	Ship_ROTTERDAM wait for handling by Port
9.	47.00 Ship_ROTTERDAM suspends
10.	47.00 QC is current now
11.	QC:Ship_ROTTERDAM start handling
12.	47.00 Ship_ROTTERDAM out of ShipsToHandleQ
13.	47.00 QC holds until 48.00
14.	INCOMING Message in ROTTERDAM
15.	\$\$from:OCEAN
16.	!!Ship_ANTWERP!!hallo from ANTWERP!!47.80
17.	47.80 Ship_ANTWERP created
18.	47.80 Ship_ANTWERP is current now
19.	47.80 Ship_ANTWERP to tail of ShipsToHandleQ
20.	Ship_ANTWERP wait for handling by Port
21.	47.80 Ship_ANTWERP suspends
22.

\$\$clients and \$\$from are key-words recognized and interpreted by the server: \$\$clients being a request for a list of models

connected and \$\$from being a message send by one model to another model



```
Process Server
Start Stop
Waiting for clients...
Connecting: 130.161.23.41
Client 1 connecting from 130.161.23.41
1 is named OCEAN
Connecting: 130.161.23.41
Client 2 connecting from 130.161.23.41
2 is named ROTTERDAM
Connecting: 130.161.23.41
Client 3 connecting from 130.161.23.41
3 is named DUTIGGC
Connecting: 130.161.23.156
Client 4 connecting from 130.161.23.156
4 is named ANTWERP
Connecting: 130.161.23.40
Client 5 connecting from 130.161.23.40
5 is named HAMBURG
Client 2 disconnecting: 130.161.23.41 Duration: 00:09:00
Client 1 disconnecting: 130.161.23.41 Duration: 00:09:20
Client 3 disconnecting: 130.161.23.41 Duration: 00:08:17
Client 5 disconnecting: 130.161.23.40 Duration: 00:04:53
Client 4 disconnecting: 130.161.23.156 Duration: 00:07:59
```

Figure 7. Server log from a run with distributed model 1 with 4 PortModels and the OceanModel.

RESULTS, CURRENT AND FUTURE WORK

Three examples of models that have been used for testing and verification of the distributed simulation package TOMAS are discussed on the level of pseudo language. In TOMAS, time synchronization of models is obtained with the help of a time server using TCP/IP protocol and Windows sockets. The distributed model may run on computers connected to the Internet. It is important to state that, though the set of models is distributed, only one model is running at a time. Parallel computing does not apply. TOMAS is fully operational and available free on the Internet.

The process-oriented modeling approach provides a natural way for the design of distributed models. In pseudo language there is hardly any difference between a stand-alone model and its distributed variants. Distributed modeling facilitates parallel modeling at distributed locations. A proper communication set must be established between member models and between member models and the time server.

Other current work relates to the server being used to develop distributed control of simulated objects. The control system then runs on one or more sub-models on one or more computers connected to the server and controlling distributed simulation models. The advantage of this is that local control functions may be developed and tested dynamically. Moreover the testing can be expanded by substituting the real elements for the simulated elements in this way leading to reuse of software.

Some topics which will be elaborated in future are the further development of pseudo language to design models and the streamlining of the messaging between member models. Further work relates to aggregated modeling with zooming

possibilities, and real time simulation with substitution of simulated elements by real world elements.

REFERENCES

- Birtwistle, G. M., O. J. Dahl, B. Myhrhang, K. Mygrard (1973), Simula Begin, Van Nostrand Reinhold, New York.
- Crain, Robert C. *Simulation using GPSS/H*. Proceedings of the 1996 Winter Simulation Conference ed. J.M. Charnes. pp 453-459.
- Duinkerken, Mark B. and Jaap A. Ottjes, 2000. A simulation model for automated container terminals. In *proceedings of Advanced Simulation Technology Conference (ASTC2000) April 16-20, 2000*, Washington, D.C. pp. The Society for Computer Simulation International (SCS), ISBN: 1-56555-199-0.
- Duinkerken, M.B.; Evers, J.J.M., Ottjes, J.A. 1999. *TRACES: Traffic Control Engineering System*. Proceedings 31st Summer Computer Simulation Conference. Chicago [SCS] 1999, pp. 461-465.kl
- Fujimoto R.M. 1997. *Zero Lookahead and Repeatability in the High Level Architecture*. 1997 Spring Simulation Interoperability Workshop.
- Fujimoto R.M. 2000. *Parallel and Distributed Simulation Systems*. Wiley Series on Parallel and Distrinuted Computing. Wiley, New York.
- Healy, J. R.A. Kilgore 1997. "Silk.: A Java-Based Process Simulation Language". *Proceedings of the 1997 Winter Simulation Conference*, IEEE, Klein, U., Strassburger, S., Beikirch, J. *Distributed Simulation with JavaGPSS based on the High Level Architecture*. International Conference on Web-based Modeling and Simulation, Jan. 11.-14. 1998, San Diego.
- Ottjes, J.A., F.P.A. Hogedoorn; "Design and control of multi-AGV systems: reuse of simulation software." *Proceedings of 8th European simulation symposium*. Society for Computer Simulation Internationa Genoa 1996, p. 461-465. ISBN: 1-56555-099-4
- Ottjes, J.A., Duinkerken, M.B., Evers, J.M., Dekker, R. "Robotised inter terminal transport of containers", *Proc. 8th European Simulation Symposium 1996* Genua [SCS] pp. 621-625, ISBN 1-56555-099-4 Vol I
- Rabe M. 2000. *Mission:Modelling and Simulation Environments for Design. Planning and Operation of Globally Distributed Enterprises*, Fraunhofer Institute for Production, Systems and Design Technology (IPK), Berlin; Esprit Project No. 29656.
- Robert, C.A., Dessouky, M., 1998. "An Overview of Object-Oriented Simulation". *Simulation* vol:70:6, pp. 359-368. 1998.
- Veeke, H.P.M. 1982. "Process simulation as a management Tool, *Proceedings of the IASTED International Symposium Applied Modeling and Simulation*, Paris, 1982.
- Veeke, H.P.M. and Ottjes, J.A. 1999. "Problem oriented modeling and simulation", *Proc. 1999 Summer computer Simulation Conference (SCS'99) Chicago, Illinois pp.110-114, ISBN 1-56555-173-7*
- Veeke, Hans P.M., Jaap A. Ottjes, 2000. *TOMAS: Tool for Object-oriented Modeling And Simulation*. In *proceedings of Advanced Simulation Technology Conference (ASTC2000). April 16-20, 2000*, Washington, D.C. pp. 76-81. The Society for Computer Simulation International (SCS), ISBN: 1-56555-199-0
- Veeke, H.P.M., Ottjes, J.A., "2001, "Applied Distributed Discrete Process Simulation", submitted to ESM 2001, Prague.
- Zeigler B.P., Praehofer H and Kim T.G.. 2000. "Theory of Modeling and Simulation 2nd Ed. Academic Press, San Diego