Simulating a Gas Distribution Network in a Distributed Object Oriented System

António Ribeiro **Alcino Cunha** **Orlando Belo**

{anr,alcino,obelo}@di.uminho.pt

Department of Informatics, School of Engineering, University of Minho

Largo do Paço, 4709 Braga Codex, PORTUGAL

Abstract

Today's dynamic industrial process simulation problems require systematically new methodologies and sophisticated computational tools. Such processes involve frequently discontinuities, environment structures changes and entities with high functional levels. Furthermore, there are cases where we must integrate intelligent techniques and negotiation protocols. These characteristics are crucial in distributed problems that require resource balance, low cost distribution plans, and stock optimization. In order to analyse the application of a distributed object-oriented simulation system we selected, as a case study, a gas distribution network in which we find all the referred characteristics. This paper presents a brief description of the simulation scenario, the overall system's structure, the intelligent negotiation protocol used by system's objects and the concurrent programming techniques to implement it.

Field **:** Intelligent Simulation Environments.

Keywords **:** Industrial Simulation Environments, Concurrent Simulation, Dynamic Process Simulation, Agent Based Negotiation Protocols, Object-Oriented Frameworks, Java Concurrent Constructs.

1 Introduction

Industrial processes are common "targets" for the application of simulation processes and techniques. This situation is due to the fact that the experimentation of new working methods, or even the re-engineering of the existing ones, are too expensive and difficult to do directly in the implemented systems. We may find applications of simulation to industrial processes covering a large variety of issues, such as stock and white water system dynamics [9], hardmetal tools production systems [12], power plant industry [13], or manual flow lines on manufacturing processes [10].

In the past few years we have assisted to the emergence of new object-oriented techniques and their successful application to real world problems. New object-oriented methodologies, frameworks, and languages have been integrated and applied in simulation case studies, in modeling applications, or even in the development of computational testbeds to simulation environments. Areas such as the design of central solar heating plants [14], multi-facet modeling [8], reactor process modeling [1], and simulation, optimization and visualization of biomechanical systems [7] are some illustrative examples in which object oriented techniques were applied.

The success and the quick adoption of object-oriented techniques on the simulation field is due to the fact that object technology [2] provides high level abstraction for modeling real-world systems, improves and simplifies system development, supports software extendibility and reusability, and reduces software maintenance costs. The gap between the methodological and implementation levels verified in the past practically does not exists nowadays. Many of the current object-oriented methodologies are supported by powerful tools that are able to generate software source code based on the object system specification.

In this paper we describe the modeling and the simulation of a *Gas Distribution Network* (GDN) using a distributed and concurrent object-oriented system. Other approaches to this kind of problem were done before [11]. However, we present a different view of the problem integrating in the distributed object-oriented system agent based negotiation protocols. This combination will allow to simulate a GDN where distributors are intelligent entities that are able to define, together, new and better distribution plans, balance their own capacity of gas stock, and negotiate among them eventual gas supplements in cases of local stock rupture.

2 Modeling the Gas Distribution Network

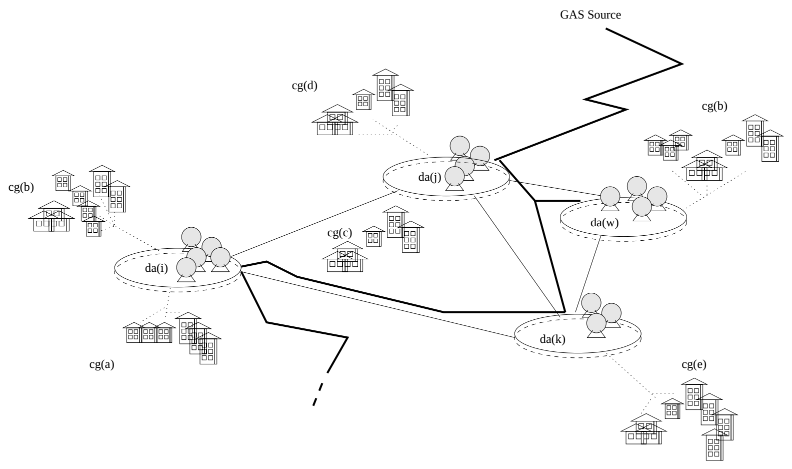

Figure 1: The Gas Distribution Network Model.

The GDN (Figure 1) integrates essentially three main functional elements:

- *gas sources* - are the primer gas suppliers; they are responsible to feed the entire network system in order to allow the fulfillment of the distribution agents' local gas tanks.
- *distribution agents* (da(_)) - responsible to satisfy the consumer groups' needs and manage their local gas tanks.
- *consumer groups* (cg(_)) - final clients of the gas; they demand more or less amounts of gas to the distribution agents according to the current needs of the local infrastructures - houses, industrial plants, agriculture facilities, etc.

In our model we considered a single gas source, a set of distribution agents, and a set of consumer groups. We assumed that, in an initial phase, the gas source supplies all the gas amounts demanded by the distribution agents to fulfill their local tanks. After this phase, the distribution agents begin to evaluate the consumer needs of gas and, according their own capacity, try to satisfy them. When a distribution agent detects potential breaks in gas distribution caused by excessive consumer gas demands, it establishes a plan to avoid ruptures in its local gas stocks and consequent failures in gas distribution.

It is possible to detail the presented model in order to receive more levels of gas distributors, several gas sources or even to define pipeline capacities of gas transportation. Such characteristics could improve the GDN's model towards a more realistic one. However, it is detailed enough in order to reach our two main goals: to improve the distribution processes through intelligent negotiation protocols and optimize gas stocks on the distribution agents' tanks.

In Figure 2 we present the class (entity) diagram of the GDN model (this diagram follows the Booch notation [2]). In this diagram are only drawn the aggregation relationships. They state the system's overall integrity invariant, which guarantees that each client is only related to one distributor and that each distributor belongs just to one gas source.

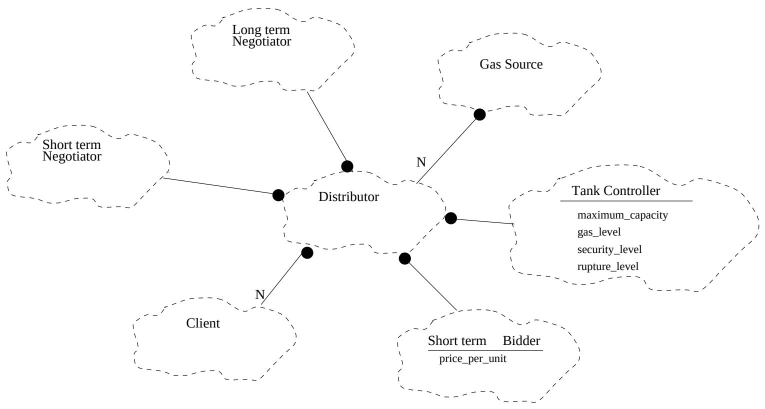

Figure 2: The Object Oriented Model.

Each distribution agent is constituted by four objects:

- *Tank Controller* (TC). Responsible for controlling the gas tank of the distributor. Each tank is characterized by its *maximum capacity* (*mc*) and by two alarm levels: a *security level* (*sl*), that determines when the distributor should start a long term negotiation with the primer supplier, and a *rupture level* (*rl*), that forces the distributor to engage into short term negotiations with nearby distributors in order to be able to maintain its supplying capability. The actual amount of gas in the tank is denoted as *gl* (gas level). The main task of the TC is to warn the negotiator agents when $gl < sl$ or $gl < rl$.
- *Long term Negotiator* (LtN). When the TC announces that the amount of gas in the tank is under the *security level* this agent starts a negotiation process with the primer supplier in order to reestablish the gas level. Basically, it implements the manager role of the *Contract Net Protocol* (CNP) [15, 5], a protocol that tries to reproduce the negotiations that occur in real markets when a particular entity wants to find the best partner to execute a task or supply a product or service. It is responsible to announce to the primer suppliers the amount of gas that the distributor is willing to buy (the *announcing* phase of the CNP) and, afterwards, select the best partner according to the bids received (the *awarding* phase).
- *Short term Negotiator* (StN). This agent has the same functionality of the LtN, but instead of negotiating with the primer supplier it negotiates with other distributors, in order to obtain the gas within a short period of time.
- *Short term Bidder* (StB). Responsible for answering the announces of the StN agents from other distributors. It implements the contractor role of the CNP by submitting a bid with the price

proposed for the amount of gas required (the *bidding* phase of the CNP). A parameter of this agent is the *price per unit* (*ppu*) of gas.

The behaviour of the negotiator objects is not very complex. In the *announcing* phase they simply send a message to all the bidders announcing that they are willing to buy a certain amount of gas. We assumed that the LtN sends the announces to the primer supplier requesting an amount of $mc - sl$ units of gas, while the StN sends them to the StB objects of the distributors, requesting an amount of $sl - rl$ units. In the *awarding* phase they simply collect all the bids and choose the one with lowest price. An *award* message is sent to the chosen distributor and a *reject* one is sent to all the others.

The StB has a more complicated behaviour, since it must handle the situations where multiple announces (and hence bids) were made simultaneously. Let A be the current set of pending announces to which a StB has answered with positive bids and that have not be awarded yet. Given a particular announce $a \in A$, let $gas(a)$ be the amount of gas that is being negotiated in it. Let $total = \sum_{a \in A} gas(a)$ be the total amount of gas that a distributor may have to provide if all bids are accepted (we assumed that the distributors always honor their compromises). Although in most situations *total* will not be the amount of gas sold, when $gl - total < rl$ we assume that the StBs answer with negative bids to all the announces received (a negative bid is a bid where the StB states that it is not interested in selling gas). This possibly inaccurate decision is necessary, since the *rupture level* defines the capacity that allows the distributor to provide a minimum quality service to his clients.

In order to model more accurately real situations, the *ppu* parameter is a function that depends on the amount of gas in the tank. Usually, the price that a distributor is willing to ask for a unit of gas when its tank is almost empty is higher then when it is full. This occurs since in these situations the number of clients that may not have their needs fulfilled is also higher.

To determine the price that will be proposed it is necessary to estimate the amount of gas available to sell. However, it is impossible to predict if each bid will be accepted (this is true when the *ppu* of each agent varies with time and hence can not become common knowledge among the distributors) and, consequently, to predict this amount. In order to define an heuristic to estimate this value, we choose to characterize each StB with a parameter α that defines the distributor optimism level. A value for α near 0 characterizes a pessimistic distributor that does not expect awards to bids it has sent. A value near 1 characterizes an optimistic distributor that expects positive answers to almost all the bids. If $\alpha = 0.5$ we have a neutral behaviour. $\mathcal{E}(total)$, the expected total amount of gas sold, is determined as $\alpha \times total$. Given an announce a the price proposed by the StB is $ppu(gl - \mathcal{E}(total)) \times gas(a)$.

3 Implementation of the GDN Model

One of the most important and innovative features of the Java language [6] is its support for concurrent programming. This is achieved by the incorporation of multithreading in the core language. Therefore, Java is a good candidate to support the simulation of concurrent and distributed systems. Although the multithreading support is present, it is necessary to provide the necessary synchronization techniques between the running threads. This synchronization is done using *monitors*, a well known concept first introduced by Hoare [3].

Once that threads are light-weight processes with their own life cycle, it is necessary to provide a secure and easy mechanism to support synchronization and message passing. In our Java support architecture, this is done by using objects that behave like channels (according to the theoretical basis of CSP [4] and by providing the methods to send and receive messages between asynchronous and possibly remote threads. This is achieved using the core language support to provide thread-safe communication, that is, the `synchronized-wait-notify` construct.

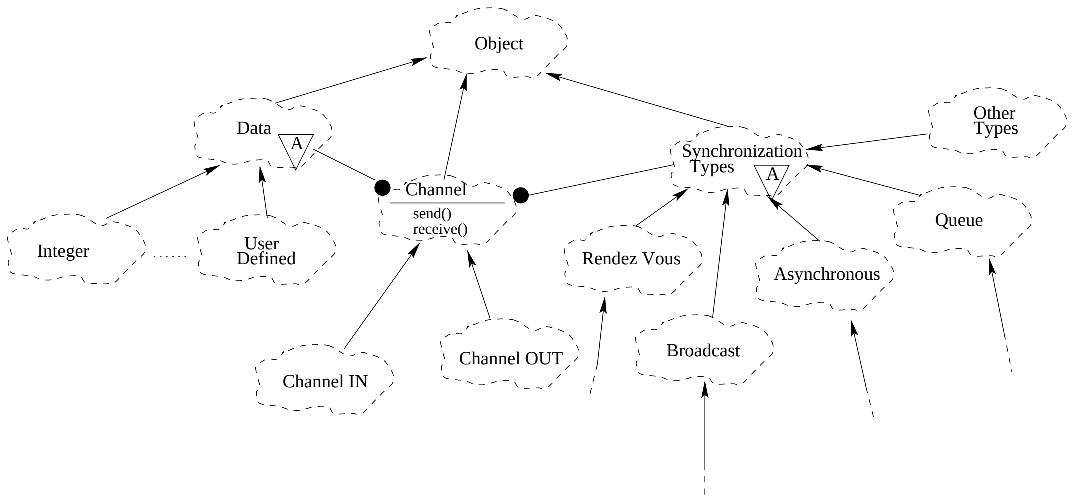

Figure 3: The Object Oriented Architecture for the Simulation of Concurrent Systems.

An architecture was developed in order to simplify the construction process of concurrent and distributed models. The classes provided encapsulate all the necessary synchronization and low level features of this kind of complex modeling. The main class of our architecture (Figure 3) is the *Channel* class. It provides two methods, `send()` and `receive()`, which write and read data into a channel. In order to establish communication between two running threads an instance of *Channel* should be passed to those processes at bootstrap time. Although the `send()` and `receive()` methods are naturally synchronous one may want to model asynchronous behaviours between engaged processes. That is why the constructor of a *Channel* instance also accepts a synchronization mechanism. By default, the synchronous one is provided, ensuring a simple *rendez-vous* behaviour. A channel has the capability of `send` and `receive` values, being therefore bi-directional. However to model the situations in which one only reads or writes values, two subclasses were provided, *ChannelIN* and *ChannelOUT*, to simulate uni-directional channels. We also provide some asynchronous environments in order to model the basic and common asynchronous dialogues. Three environments were implemented:

- *Asynchronous* - to model a one position buffer on which producers and consumers write and read values, without concerning if past messages were read or if the same value is read more than once. This is a non-deadlock environment.
- *Queue* - a normal asynchronous buffer with more than one position. Well known examples of application are the email queues. A writer does not wait that the message is consumed and just write it in an available position of the queue. Some concern must be taken in order to ensure that empty or full queues do not behave synchronously.
- *Broadcast* - extends the above model to multiple queues.

Our architecture is open in the sense that an user can write another synchronization environment. This is done by subclassing the abstract class *Synchronization_ Types*. Support to synchronization between threads in different machines is transparent to the user.

To complete our architecture we must address the *Data* class which provides the mechanisms to deal with our data types. Some are provided, the basic ones, assuming that an end user will subclass *Data* and define his protocol data, that is, the message format.

In order to exemplify the implementation methodology we will show how to build part of the Distributor and Client agents. Since that the most innovative part of our architecture is the enhanced support to synchronization between threads, the following example will only encompass the code necessary to establish communication between the TC object and the clients.

```
public class Distributor {
  protected TankController tc;
  protected Channel to_tank;
  protected Client[] clients;
  // other instance variables
  ....
  // The Distributor constructor
  Distributor( int number_clients, ....) {
    this.to_tank = new Channel( /* kind of synchronization */ );
    // the channel must be shared by the running thread
    this.tc = new TankController(to_tank);
    ....
    cl = new Client[number_clients];
    // It is also necessary to share it with our clients
    for(int i=0;i++;i<= clients.length)
      clients[i].set_communication_port(to_tank);
    ...
  }
}
```

The *TankController* entity has to receive in his constructor the channel reference and need to modify his inherited `run()` method in order to describe properly his own life cycle.

```
    public class TankController extends Thread {
      protected Channel from_clients;

      //Constructor
      TankController(Channel chan) {
        this.from_clients = chan;
      }

      public void run() {
      // Task Controller life cycle implementation
      ....
      }
   }
```

Finally, the *Client* class looks like,

```
public class Client extends Thread {
  protected Channel to_Tank_Ctr;
  ....
  void set_communication_port(Channel chan) {
    this.to_Tank_Ctr = chan;
  }
}
```

4 Conclusions and Future Work

In these paper we presented a concurrent object-oriented simulation environment for a Gas Distribution Network. The use of intelligent negotiation techniques enables to: improve the distribution processes through intelligent negotiation protocols; optimize gas stocks on distribution agents' tanks, storing lesser amounts of gas; and, increase distributors self-adaptation to unpredictable situations of gas demands.

By allowing the price per unit of gas to be function depending on the amount of gas in the distributor's tanks, it is possible to model most of the pricing policies that a real distributor may wish to use. The optimism level of the distributors can also be modeled through a specific user defined parameter.

Java revealed itself as a good simulation environment, since it supports concurrent programming in multi-platform distributed systems. However, writing source code for concurrent programming is usually an hard and error prone task. To overcome this shortcome, we proposed an architecture that provides means of synchronization between running threads. The end user needs only to concentrate on programming the high level functionality of the system. In the future we intend to augment this architecture with a visual layer that will provide different output interfaces to each of the objects in the system.

References

[1] B.Foss, S.Wasbo, and O.Ogard. Object-oriented Process Modelling Applied to a Reactor. In *Proceedings of the Eurosim Congress '95*, pages 463–474, Vienna, Austria, September 1995.

[2] Grady Booch. *Object-Oriented Analysis and Design.* Benjamin Cummings, 1994.

[3] C.Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[4] C.Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[5] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. In A. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 333–356. Morgan Kaufmann, 1988.

[6] David Flanagan. *Java in a Nutshell.* O'Reilly & Associates, 1997.

[7] G.Kramann, J.Seybold, and R.Ruhle. Object-Oriented Modeling, Simulation, Optimization and Visualization of Biomechanical Systems.

[8] H.Tummescheit, M.Klose, and T.Ernst. Modelica and Smile - A Case Study Applying Object-Oriented Concepts to Multi-Facet Modeling. In *Proceedings of the 9th European Simulation Symposium (ESS'97)*, pages 122–126, Passau, Germay, October 1997.

[9] I.Laukkanen, J.Lappalainen, and K.Juslin. Simulation of Stock and White Water System Dynamics of a Modern Paper Mill - A Case Study. In *Proceedings of Eurosim'98 Simulation Congress*, pages 130–137, Espoo, Helsinky, Finland, April 1998.

[10] I.Praça. Simulation of Manual Flow Lines. In *Proceedings of the 9th European Simulation Symposium (ESS'97)*, pages 337–341, Passau, Germay, October 1997.

[11] L.Rothkrantz, R.vanVark, J.deVreught, and V.Broz. Simulation Studies in the Distribution of Gas Flow. In *Proceedings of the 9th European Simulation Symposium (ESS'97)*, pages 689–693, Passau, Germay, October 1997.

[12] O.Belo. A Hardmetal Tools and Wear Parts Production System Simulation. In *Proceedings of the 8th European Simulation Symposium (ESS'96)*, pages 135–139, Genoa, Italy, October 1996.

[13] R.Burelle and J.Salim. Real Time Simulation in the Power Plant Industry. In *Proceedings of the 8th European Simulation Symposium (ESS'96)*, pages 580–584, Genoa, Italy, October 1996.

[14] R.Franke. Modelling and Optimal Design of a Central Solar Heating Plant with Heat Storage in the Ground Using Modelica. In *Proceedings of Eurosim'98 Simulation Congress*, pages 199–204, Espoo, Helsinky, Finland, April 1998.

[15] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, 1980.