Elements for a course om design of distributed algorithms

Noël Plouzeau, Michel Raynal

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1582

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

ELEMENTS FOR A COURSE ON THE DESIGN OF DISTRIBUTED ALGORITHMS

Noël PLOUZEAU
Michel RAYNAL

Janvier 1992

RR-1582

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Elements for a course on the design of distributed algorithms*

Noël Plouzeau Michel Raynal

12 décembre 1991

Publication Interne n° 622 - 12 pages

Projet ADP – Programme 1 –
INRIA/IRISA

Campus de Beaulieu – 35042 RENNES CEDEX – FRANCE

Abstract

Sequential algorithms design and operating system principles have always been fundamental courses in any computer science curriculum. Protocols are now a well established discipline and parallelism and concurrency issues are becoming more and more popular in academic courses. Along these guidelines distributed algorithms have now emerged as a proper topic of computer science; studying them demands some prerequisite on algorithms, parallelism and protocols but they cannot themselves be reduced to these three domains.

In this paper we present elements for a course on the design of distributed algorithms performing common operating system services. The fundamental aspects of this course lie in teaching the students that no global state can be instantaneously caught because of the asynchronism of the processes and message transmission delays. We state basic problems addressed during the lecture (mutual exclusion, rendez-vous implementation, snapshot computation, network traversals and distributed evaluation of predicates) and present how students are faced with distributed problems in practical classes, using a distributed memory parallel machine to implement their solutions.

Eléments de cours sur la conception d'algorithmes répartis

Résumé

L'étude des algorithmes séquentiels et des fondements des systèmes d'exploitation a toujours été un aspect fondamental de l'enseignement de l'informatique. De plus en plus de cursus prennent en compte la maturité de l'algorithmique des

*This work has been supported by the french GRECO C^3

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE (U.R.A. 227) UNIVERSITE DE RENNES I I.N.S.A. DE RENNES
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE (UNITE DE RECHERCHE DE RENNES)

programmes parallèles. Au sein de ce vaste sujet, l'algorithmique des programmes répartis se distingue par ses problèmes et ses solutions spécifiques.

Le présent document décrit un cours sur l'étude des algorithmes répartis qui rendent des services courants dans les systèmes d'exploitation répartis : exclusion mutuelle, mise en œuvre du rendez-vous, calcul d'état global, parcours de réseau et évaluation répartie de prédicats. Les difficultés propres aux algorithmes répartis sont soulignées. Les étudiants sont confrontés à des mises en œuvre lors de travaux dirigés et de travaux pratiques.

1 Introduction

Any Computer Science curriculum must include courses on concurrency and has to cope with the many faces of this topic, from hardware level issues up to the specification and implementation of parallel and distributed applications. As far as software development is concerned, it is now clear that a good understanding of algorithmics is an absolute prerequisite. In this paper we focus on the teaching of distributed algorithmics basis.

Students attending a course on distributed algorithms should have a good knowledge of sequential algorithms as well as basic notions on parallelism and interprocess cooperation for centralized computations. The first important issue of the course is to clearly define the differences between on the one hand parallel computations on multiprocessor machines with common memory and on the other hand distributed computations made of processes with no common memory.

In the next section a taxonomy of classical algorithms will be presented, mainly to show how these algorithms are built from solutions to specific problems. This style of presentation aims at improving the understanding of the paradigms and exposing somewhat hidden similarities and differences. Students should then be able to adapt these solutions to new hypothesis on the environment.

2 Some fundamentals of distributed algorithmics

As stated in [5], *a major concern in the study of parallel computation is* performance *of algorithms, whereas a major concern in the study of distributed computing is* dealing with uncertainty. Hence, every student has to realize that the exact state of the whole computation cannot be instantaneously computed: every distributed operation may involve network traversals, which cannot be performed instantaneously.

Following the usual way of structuring general algorithmics, another important issue is the separation between *paradigm* and *implementation*: for instance a tree abstract data type has properties and uses of its own, whatever its implementations. In the same way, distinguishing distributed algorithmics paradigms and implementations is important.

In a distributed context paradigms can be classified into two categories: on the one side "interpreter-like" paradigms perform a global control upon the computation, and in some sense supply the user with distributed operations; on the other side "observer-like" paradigms aim at giving the user some view of the global state of the distributed system.

In the rest of this section we present a few algorithms; for each of them we will give a short presentation of the problem they solve, together with a naive implementation. Then limits of this implementation are emphasized and the implementation is refined; this exposition process aims at the justification of the structure of the algorithms presented to the students.

2.1 Maintaining a global invariant

Many algorithms are best expressed as invariant maintainers: the algorithm specification gives the set of all meaningful computation states and the implementation is then designed to forbid "bad" states. A fair amount of distributed algorithms fall into this category, especially the ones implementing system services,

such as mutual exclusion, resource allocation and interprocess coordination.

An algorithm has to ensure that the global states are *safe* but also that the computation is *live, i.e.* ensure that the service implemented by the algorithm will be eventually performed.

2.1.1 Mutual exclusion

This well-known type of system service (or interpreter-like algorithm) is a classical problem of global invariant maintenance: at any time, at most one process is inside the global mutual exclusion region. Common distributed algorithms can be divided into two classes: in the first class all processes ask some of the other processes the privilege to enter the critical section and consequently these algorithms are called *permission-based*. In the second class the right to enter the critical section is attached to the possession of a special object (a *token*); algorithms of this class are called *token-based* [16].

The well-known Ricart and Agrawala's algorithm [18] is the basic representative of the first class. To obtain the mutual exclusion privilege a process P has to ask every other process in the network and then wait for their positive replies. A process with a pending request differs its answer, thus preventing the violation of the mutual exclusion invariant. Unfortunately, this simple implementation is *safe* but may lead to deadlock: to suppress this problem some priority system is needed. Statically assigned priorities may lead to starvation; hence Ricart and Agrawala's algorithm use a dynamic priority scheme based on Lamport's logical clocks [12].

In the token-based class, the uniqueness of the token trivially ensures safety. So a token-based algorithm has only to define the way the token moves along from site to site in order to ensure liveness. Defining a token-based algorithm is mainly a question of network traversal implementation: the token has to traverse the network of processes to satisfy the requests. In its simplest form this token traversal is made along a path including all network processes and no request messages are used; a process P receiving the token let it go as soon as P doesn't use it. The simplest path is a logical ring: in that case the algorithm is the well-known token ring and the route followed by the token is defined statically. More sophisticated algorithms can be found in [13].

Faced to these algorithms, students learn two concepts: what a network traversal is and how to maintain a global invariant.

2.1.2 Resource management

The resource management paradigm is a generalization of the mutual exclusion one. One particular class is the well-known producer/consumer problem. Several distributed solution to this problem are detailed in section 4.

2.1.3 Interprocess synchronization

The binary rendez-vous concept is an important synchronization tool. A study of its implementation in asynchronous distributed systems is an interesting case of interprocess synchronization problems. Building synchronous primitives above an asynchronous message-passing interprocess communication layer is not a trivial task: many algorithms were proposed [2, 4], some of them being quite complicated. The rendez-vous facility is defined as follows: a process P_i defines a rendez-vous set RVS_i made of process identities and waits until a rendez-vous

with one of these processes occurs (see [9]). The implementation of the rendez-vous is *safe* when rendez-vous occurs only for pair of processes waiting at their rendez-vous point with each one belonging to the *RVS* of the other and it is *live* when two processes continuously able to synchronize will eventually do so.

We chose to present the Bagrodia implementation [1] to the students as this algorithm is fairly easy to understand and to expose in a modular way. To sketch out the principles of this algorithm we first choose a local point of view, *i.e.* we observe the execution of the algorithm on some process P_i. Whenever process P_i wants to establish a rendez-vous it tests sequentially every process from its own subset RVS_i, one after another, by sending a *query* message and waiting for an answer. If the answer is positive then the rendez-vous is performed. Otherwise P tests another untested process from the subset. From the point of view of P_i's communications, we have a sequence of one-to-one message exchanges. At this stage of the solution, a question is raised: what to do when all replies are negative. A second principle of the algorithm is that for every pair of processes only one process has the right to query the other. This technique solves the question raised above: the *query* message is treated as a token; sending this token to the other process means that the sender is querying the receiver. Sending the token back is interpreted as a positive reply. Thus when a process has queried all processes in vain, the only behavior left is waiting for queries.

Now we need to consider global behavior issues: process P_i is not the only active process. A process P_i receiving a query from a third process P_k while engaged in a test (*i.e.* waiting for a reply from some P_j) may either (1) answer negatively or (2) delay its answer. If every process behaves symmetrically, it is easy to see that solution (1) may lead to a livelock situation (processes keep testing but to no end) and solution (2) may lead to deadlock (eventually all processes are waiting). A simple priority scheme is sufficient to prevent livelocks and deadlocks: process P_i answers negatively when $k < i$ and delay its answer when $k > i$. At the end of its current test, P_i will reply to the delayed queries.

With this algorithm, students are faced to a liveness problem and are presented a basic implementation to solve it.

2.2 Observing a distributed computation

2.2.1 Snapshot computation

Perhaps the most natural paradigm from the class of algorithms aiming at observing a distributed computation that we have to teach to the students is the computation of its global state. From a pedagogical point of view, the presentation of an algorithm for snapshot computation is interesting because it emphasizes that taking the global state of a distributed application instantaneously is impossible.

Given a distributed computation, the problem is the definition of what a computation's global state is and also the implementation of an algorithm to compute it. The only way to compute such a state is by collecting the local states of processes (the value of its variables and its control state) and of communication channels (the set of messages in transit with respect to these local states). As local states are taken at different moments, inconsistencies may occur, for instance P_j's state records the reception of one message from P_i but P_i's state does not record its emission. Any consistent set of local states

is acceptable; such a set is called a *snapshot*. It is obvious that the result of a snapshot computation is not unique: in fact a snapshot describes a global state possibly reached by the distributed computation.

The first implementation of a snapshot algorithm is Chandy and Lamport's one [3]. Its basic principles are as follows: one process (e.g. P_i) triggers a snapshot by recording its local state and sending a *marker* message on every outgoing communication channel, that we assume to be FIFO and loss free. Because of the FIFO property of the channels, a marker message sent on a channel from P_i to P_j "pushes" any message in transit towards P_j. When P_j receives a marker for the first time from one of its incoming channels, it records its local state and then sends markers on its outgoing channels. Moreover, the state of an incoming channel is defined by the sequence of messages a process has received between the time of the local state recording and the reception of a marker from this incoming channel.

With this basic problem students can understand some subtleties of distributed observation and then realize that the computable states are necessarily past ones.

2.2.2 Global predicate evaluation

A global predicate is a predicate defined over the global state of a distributed computation, *i.e.* the cross-product of the states of processes and channels. Some interesting properties of distributed application are easily expressed with a global predicate: e.g. the property of being terminated or deadlocked. This kind of property is called a *stable* one: once satisfied by a computation, it remains so forever [4, 6].

A simple idea to design a global predicate evaluator is taking a snapshot of the distributed computation and then evaluate the predicate over that snapshot. Unfortunately, snapshots are past values of the global state, and the result of the evaluation may be incorrect in the general case although some stable predicates can be safely evaluated in this way: the global state of a deadlocked or terminated application is stable, and thus coincides with all snapshots.

Some algorithms are tailored to detect the occurrence of an application's termination. Instead of collecting the local state of each process they simply collect the control state of the process and the count of messages in every channel. When all processes are waiting for messages or have finished their own job and all channels are empty then termination is detected. As in the snapshot computation the main problem is collecting consistent information while the distributed computation is progressing.

The global predicate evaluation presentation helps students to try and adapt techniques from snapshot algorithms to solve specific global state oriented problems.

2.3 Network traversals

Every distributed algorithm may need to use network traversal techniques: some sites and some channels have to be visited. Separating the traversal aspect from other aspects of an algorithm improves the understanding of the algorithm and also allows adaptation of a solution to other network topology or environments.

Common network paradigms are: broadcasts with or without reply, depth-first traversals and message waves [19, 8].

For instance, Ricart and Agrawala's mutual exclusion algorithm requires a broadcast with reply facility to route re-

quests and answers. Implementing such a facility is easy in a fully connected network (*i.e.* when every pair of process is connected by a direct link): in such a network, broadcasting is simply sending a message to every neighbor. Routing a token towards a process is also trivial. If the network is not fully connected then a protocol performing a broadcast with reply from each addressee has to be added to the algorithm [7]. This last case shows the students that addressing routing issues within the algorithm itself and not in an independent network layer may improve the efficiency because of "intelligent" routing techniques.

Many algorithms use a wave of messages to trigger some action on each process or to collect data. Two types of wave exist: the one-way and the two-way (bouncing) wave. One-way waves traverse the network and disintegrate when all processes have been visited. For instance, the Chandy and Lamport's snapshot algorithm uses a one-way wave to trigger the recording of local states [3]. Two-way waves reflect back to the initiator when all processes have been visited. Waves reveal themselves as powerful paradigms of distributed control structures [8].

3 The Estelle framework

The previous section has laid out important paradigms and typical algorithms (which where informally described). Practical classes strengthen the learning of distributed algorithmics with exercises and practical case studies. Students are invited to describe formally some of the algorithms and to try and adapt them to other problems. A programming language suited to this task is the ISO normalized Estelle language [10], mainly because it uses asynchronous message-passing communication and also because we can use a distributed implementation of a subset of Estelle [11]; Estelle programs are executed on a 64 nodes hypercube. The following paragraph is a quick presentation of the Estelle concepts taught to the students; these concepts allow the specification of simple distributed algorithms.

3.1 Introducing Estelle

An Estelle program is a description of a network of extended finite state automata, called *processes*. Each process has a control state and local variables (hence the word "extended"), and interacts with the environment by exchanging messages through ports. These ports are connected by FIFO bidirectional channels. A process text is a declaration of a set of Pascal variables, a set of typed communication ports and a set of transition declarations. A transition declaration has a guard made of guard constructions (among these ones is the message reception guard) and a body which is a Pascal *begin-end* block. Each process evolves by repeating endlessly the following algorithm: compute the set of fireable transitions by evaluating the guard expressions and then execute the body of one transition from this set if it is not empty. The concept of parallelism is expressed in Estelle by the asynchronous progress of the processes and the distribution aspect lies in the fact that message transfer delays are finite but unpredictable.

Interprocess communication is achieved by input and output instructions; an input instruction is a class of guard expression specifying a port and a type of message to wait for on the port.

3.2 Pedagogical issues

Introducing Estelle to our students is fairly simple because they have already learn some communicating automata theory and have been using Pascal for two years. A few simple examples of Estelle programs (e.g. specifying a simple vending machine) are sufficient to lay out the semantics of Estelle parallelism.

4 A case study

We use the *distributed parking lot* problem [20, 15] as a subject for practical work sessions. The specifications of this problem are as follows.

A town has a parking lot with N parking places and two gates. At both gates a keeper regulates the traffic: a car is allowed to enter the parking lot only if there is at least one free parking place. Clearly the two keepers K_1 and K_2 have to coordinate; they use the mail to exchange messages about the count of cars entering and leaving by their gates. The mail transport facility is assumed to be reliable: letters never get lost and they arrive in the order they were sent.

4.1 Distributing a precondition

The problem belongs to the class of producer/consumer control problem; each keeper is a free parking place producer as well as a consumer. The students are first asked to express a global invariant and a global precondition which has to be true to let a car enter the parking lot. This condition can be expressed with two global counters, *in* et *out*, counting the number of cars which entered the parking lot and the number of cars which left it. The global precondition is then: $in - out < N$. Each counter is in fact the sum of local counters, one for each gate.

Unfortunately, each keeper has an exact knowledge of the counters at her gate (in_1 and out_1 for K_1) but the other gate is out of sight and transmission delays prevent her from knowing exactly the value of the other keeper's counters in_2 and out_2. In other words, the main problem to solve is the distributed evaluation of this global precondition $in_1 + in_2 - (out_1 + out_2) < N$ that no keeper is able to evaluate correctly. The next question is then: find out a local precondition which implies the global precondition and is computable locally by each keeper. To obtain the local precondition for keeper 1, it is sufficient to replace the counter in_2 of the remote gate by a local counter Min_1^2 such that $Min_1^2 \geq in_2$ always holds and replace out_2 by a local counter $mout_1^2$ such that $mout_1^2 \leq out_2$ always holds. The local precondition for keeper 2 is obtained *mutatis mutandis*. It is easy to prove that the local precondition $in_1 + Min_1^2 - mout_1^2 - out_2 < N$ implies the global precondition.

Students have to design a protocol between the keepers to ensure that the previous relations on (Min_1^2, in_2) and $(mout_1^2, out_2)$ always hold while trying to fill the parking lot as much as possible. Maintaining the relation on $mout_1^2$ is easily performed: keeper 2 sends a message *inc_mout* to keeper 1 when a car exits by gate 2, thus updating $mout_1^2$ with a delay due to the transmission time of the message. Maintaining the relation on Min_1^2 implies that the remote keeper 2 increases its local counter in_2 after the increase of Min_1^2. A request/acknowledge protocol is sufficient to ensure this. More precisely, keeper 2 is allowed to increase its in_2 counter only after it has sent a request message and has received an acknowledge from keeper 1; upon receiving

a request message, keeper 1 increases its Min_1^2 counter and sends back an acknowledge message.

4.2 Ensuring liveness and fairness

The protocol given above is not deadlock free: nothing is said about the behavior of the keepers when two request messages cross. When waiting for an acknowledgment, a keeper receiving a request from the other keeper may either delay her answer or answer immediately. In the first case a deadlock situation is possible; in the second case, the keeper who was waiting for an acknowledge has to abort her request otherwise the protocol may not preserve the global invariant (consider the case where there is only one parking place left); the test of the local precondition and the subsequent action is in fact a *transaction*, and the two transactions will not be executed under mutual exclusion but will interleave. Therefore a mechanism to abort one of these transactions or both of them is needed. Aborting transactions in a naive way may lead to livelock or starvation. Thus some device is needed to select a "winner" in a fair way: this problem is isomorphic with the mutual exclusion problem and students have to adapt to this problem previously learned solutions.

Students are then asked to find out a generalization of the solution, *i.e.* solving the problem for a parking lot with k gates. It is fairly easy to see that the main problem is the one just mentioned: when several keepers are waiting for an acknowledge message, only one of them is allowed to go on; adapting mutual exclusion algorithms is then trivial. Students implement their solution in Estelle; the clear semantics of this language helps to solve precise details of message interleaving and atomicity implementation. An interactive simulation environment allow them to test the behavior of their implementation down to the level of the automata transitions.

5 Conclusion

Design methodologies for distributed algorithms are emerging for several years; some basic techniques are now common [14, 17]. As this field is becoming mature, a lecture about it is not a collection of apparently totally different algorithms. In this paper we presented several important algorithms in the manner we expose them during lectures, by decomposing algorithms to show how each part solves a particular aspect of the problem. We presented how students have to cope with programs in practical classes and how they implement their solutions with a distributed language on a distributed memory parallel machine.

Références

[1] R.L. Bagrodia. Synchronization of asynchronous processes in CSP. *ACM Toplas*, 14(4):585–597, 1989.

[2] G. N. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Toplas*, 5(2):223–235, April 1983.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 3(1):63–75, 1985.

[4] K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley, 1988. 516 p.

[5] M.J. Fischer. A theorician's view of fault-tolerant distributed computing. *Springer Verlag, LNCS*, 448:1–9, 1990.

[6] J.-M. Hélary, C. Jard, N. Plouzeau, and M. Raynal. Detection of stable properties in distributed applications. 6^{th} *ACM SIGACT-SIGOPS, Symp. Principles of Distributed Computing, Vancouver, Canada*, 125–136, 1987.

[7] J.-M. Hélary, N. Plouzeau, and M. Raynal. A distributed algorithm for mutual exclusion in an arbitrary network. *The Computer Journal*, 31(4):289–295, 1988.

[8] J.M. Hélary and M. Raynal. *Synchronization and Control of Distributed Systems and Programs*. John Wiley & Sons, 1990. 124 p.

[9] C. A. R. Hoare. Communicating sequential processes. *Comm. of the ACM*, 21(8):666–677, August 1978.

[10] ISO 9074. *Estelle: a Formal Description Technique based on an Extented State Transition Model*. ISO TC97/SC21/WG6.1, 1989.

[11] C. Jard and J.-M. Jézéquel. A multiprocessor Estelle to *C* compiler to experiment distributed algorithms on parallel machines. In *Proc. of the* 9^{th} *IFIP International Workshop on Protocol Specification, Testing and Verification, University of Twente, The Netherlands*, North Holland, 1989.

[12] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986. 106 p.

[14] M. Raynal. *La communication et le temps dans les réseaux et les systèmes répartis. Collection EDF*, Eyrolles, 1991. 230 p.

[15] M. Raynal. *Networks and Distributed Computation*. MIT Press, 1988. 165 p.

[16] M. Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *ACM Operating Systems Review*, 25(1), 1991.

[17] M. Raynal. *Synchronisation et état global dans les systèmes répartis*. Eyrolles, 1992. 225 p.

[18] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Comm. ACM*, 24(1):9–17, January 1981.

[19] A. Segall. Distributed networks protocols. *IEEE Transactions on Inf. Theory*, IT29(1):23–35, January 1983.

[20] J.-P. Verjus. *Synchronization in Distributed Systems: an Informal Introduction*, pages 3–22. Academic Press, 1983.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 615 ON FAULT-TOLERANT SYMBOLIC COMPUTATIONS
Bernard DELYON, Oded MALER
Novembre 1991, 18 pages.

PI 616 USING COHERENCE TO ACCELERATE RADIOSITY
Pierre TELLIER, Eric MAISEL, Kadi BOUATOUCH, Eric LANGUENOU
Novembre 1991, 16 pages.

PI 617 INTERVAL APPROXIMATIONS OF MESSAGE CAUSALITY IN DISTRIBUTED EXECUTION
Claire DIEHL, Claude JARD
Novembre 1991, 44 pages.

PI 618 RETOUR SUR LE RESEAU SYSTOLIQUE DU PALINDROME
Hervé LE VERGE, Patrice QUINTON
Novembre 1991, 14 pages.

PI 619 TRANSFORMATIONS DU GRAPHE DES PROGRAMMES SIGNAL
Olivier MAFFEIS, Bruno CHERON, Paul LE GUERNIC
Novembre 1991, 82 pages.

PI 620 METHODES D'ANALYSE EVOLUTIVE EN TRAITEMENT D'ANTENNES
Olivier ZUGMEYER, JeanPierre LE CADRE
Décembre 1991, pages.

PI 621 GENERATING MEMORY-EFFICIENT IMPERATIVE DATA STRUCTURES FROM SYSTOLIC PROGRAMS
Zbigniew CHAMSKI
Décembre 1991, 20 pages.

PI 622 ELEMENTS FOR A COURSE ON THE DESIGN OF DISTRIBUTED ALGORITHMS
Noël PLOUZEAU, Michel RAYNAL
Décembre 1991, 12 pages.

ISSN 0249 - 6399