A Distributed Solution to Synchronous Multiparty Interaction

RAFAEL CORCHUELO, DAVID RUIZ, MIGUEL TORO, JOSÉ L. ARJONA, AND JOSÉ M. PRIETO
Departamento de Lenguajes y Sistemas Informáticos
Facultad de Informática y Estadística, Universidad de Sevilla
Avenida de la Reina Mercedes s/n, 41.012, Sevilla
ESPAÑA — SPAIN

Abstract: Multiparty interactions are the key to describe problems where three or more processes need to collaborate simultaneously in order to solve a problem, and this paper aims to show the way we have implemented this mechanism in a network computer. The main feature of our solution is that it is not bound up with the underlying network, so it is highly portable. We also report some experimental results that show that our prototype performs quite well on low cost computers.

Key words: Multiparty interaction, network computers, fairness, IP, SR. *CSCC'99 Proceedings*, Pages:3511-3516

1 Introduction

When describing the behaviour of a system implies that more than two processes need to collaborate simultaneously in order to solve a problem, classical inter–process interaction primitives such as *rendez–vous* or remote procedure calls are not adequate because the solution is usually too sophisticated. These primitives are examples of the classical client/server model that emphasises two entities exchanging messages, and they are clearly insufficient in these situations because we need to decompose natural multiparty interactions into several low–level interactions that turn our solutions into tricky descriptions.

This motivated several researchers to introduce multiparty interaction constructs into languages for the description of distributed, reactive systems. Scripts, Raddle or UNITY are good examples, but IP (Interacting Processes) [7] stands out because it is intended to have a dual role: on the one hand, it is intended to be a distributed system specification language equipped with sound semantics that turn it into a language amenable to formal reasoning; on the other hand, it is intended to be an assembler language supporting more sophisticated high–level specification languages such as LOTOS or ESTELLE. IP is equipped with a rich set of statements, being the most important the interaction statements that are used to describe coordination among a set of processes.

Several algorithms that implement the multiparty interaction statements IP incorporates have been described in the literature [4, 8, 9], but they are closely related to the underlying network architecture and they cannot be easily adapted to other networks. This is problematical because it makes them difficult to port, and incorporating the notion of fairness into them is usually quite tricky. Fairness is an important property that ensures that every interaction is given a chance to be executed. In general, several interactions may be ready for execution at the same time, but IP semantics states that only one can be fired at each synchronisation point. Thus, when a conflict occurs, one interaction is executed to the detriment of the rest. Fairness enforces that no interaction is neglected forever, but incorporating it into the algorithms we have cited is rather difficult. As a result, few IP implementations are available. The one described in [1] is the state–of–the–art compiler, but it is not in wide spread use because it runs on a transputer and it is only intended for terminating programs.

This paper aims to describe a solution we have implemented to this interaction mechanism on a network computer, which is a collection of workstations whose links can be logically rearranged at runtime. This allows us for easy distribution, it is efficient enough, and makes incorporation of fairness ex-

tremely easy while preserving portability. We have organised it as follows: section 2 recalls the notion of multiparty interaction by means of well–known problems; section 3 describes our implementation, the algorithm we have implemented to deal with fair selection of conflicting interactions, and we also report some experimental results that show that our algorithms perform well enough; section 4 glances at other authors' work and compares it with ours; finally, section 5 shows our conclusions and the work we are planning on doing.

2 Multiparty interactions

In this section, we introduce multiparty interaction in the context of IP. We assume that the reader is familiar with this language, so we only recall the main concepts. If it is not the case, please consult [7].

In IP, systems are understood as collections of co–operating sequential processes whose relationships are based on multiparty interactions. An interaction statement is a statement of the form $a[\overline{x:=e}]$, where a is referred to as the name of the interaction and $\overline{x:=e}$ is a sequence of parallel assignments usually referred to as the communication part. A process is said to be a participant of interaction a if it has an interaction statement involving a in its body, and when a process has arrived at a point where executing such interaction is one of its possible continuations we say that it is readying it. When an interaction is readied by all of its participants, we say that it is enabled, and when several interactions are enabled at the same time we say that a conflict has occurred.

IP also provides guarded non–deterministic choice statements of the form $[[]_{i=1}^{n} G_i \rightarrow S_i]$, guarded non–deterministic loops $*[[]_{i=1}^{n} G_i \rightarrow S_i]$ and a dummy statement denoted by the key word $skip$. Guards are of the form $B\&a[\overline{x:=e}]$, where B is a boolean expression and the rest is an usual interaction statement. A guard is said to be passable, i.e., their corresponding statements can be executed, as long as B holds and a is enabled.

2.1 Synchronisation

We illustrate synchronisation by means of the dining philosophers problem, which is a classic multiprocess synchronisation problem that consists of five philosophers sitting at a table who do nothing but think and eat. There is a single fork between each philosopher, and they need to pick both forks up in order to eat. This problem is the core of a large class of problems where a process (the philosopher) needs to acquire a set of resources (the forks) in mutual exclusion.

The obvious solution to this problem, using two–party interactions, consists of picking up forks in sequence. Nevertheless, a problem arises if each philosopher grabs the fork on his/her right, and then waits for the fork on his/her left to be released. In this case, a deadlock has occurred, and all philosophers will starve. If we used multiparty interactions, each philosopher would pick up his/her two forks at the same time so that no deadlock may arise. Figure 1 shows a solution to this problem in IP. The philosophers are represented by processes $Philosopher_i$, and the forks by $Fork_i$ ($i = 1, 2, \ldots, n$). $Philosopher_i$ eternally tries to get his/hers associated forks by interacting in the three–party interaction get_forks_i together with $Fork_i$ and $Fork_{i-1}$ (we assume that index arithmetic is cyclic, i.e., $1-1 = n$ and $n+1 = 1$). Thus, acquiring a resource is specified as synchronising with the corresponding processes in an interac-

DIN_PHIL :: [$\|_{i=1}^{n}$ Philosopher$_i$ $\|$ $\|_{i=1}^{n}$ Fork$_i$], **where**
 Philosopher$_i$ ::
  *[get_fork$_i$[] $\longrightarrow$ *eat*; release_fork$_i$[]; *think*]

 Fork$_i$ ::
  *[
   get_fork$_i$[] $\longrightarrow$ release_fork$_i$[]
  []
   get_fork$_{i+1}$[] $\longrightarrow$ release_fork$_{i+1}$[]
  ].

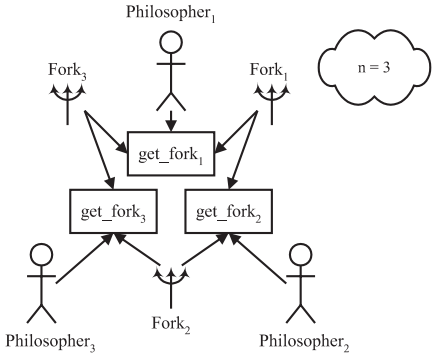


Figure 1: A solution to the dining philosophers problem in IP.

LEADER :: [$\|_{i=1}^{n}$ P_i], **where**
 P_i ::
  {w_i: natural; leader$_i$: boolean}
  w_i := *a weight*;
  Elect[leader$_i$:= (w_i = $\max_{1 \le j \le n}$ {w_j})];
  [leader$_i$ $\rightarrow$ *execute algorithm*].

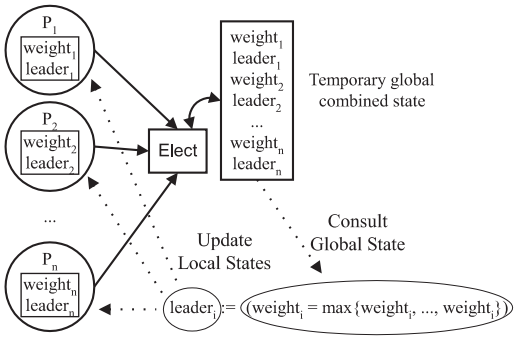
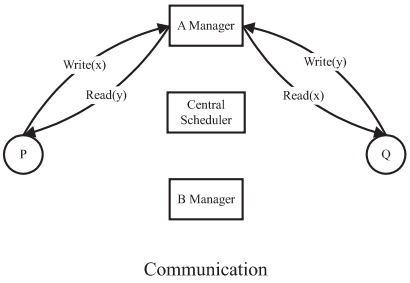


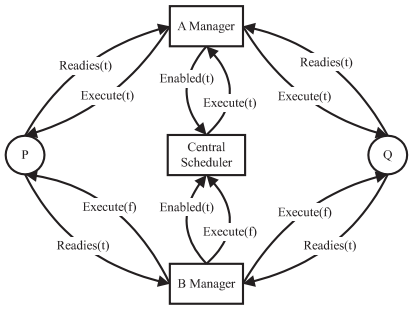
Figure 2: A solution to the leader election problem.

EXAMPLE :: [P || Q], **where**
 P_i ::
  { x: natural }
  *[A[x := y] $\rightarrow$ skip [] B[x := y] $\rightarrow$ skip]

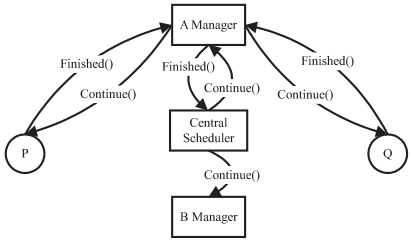
 Q ::
  { y: natural }
  *[A[y := x] $\rightarrow$ skip [] B[y := x] $\rightarrow$ skip]



Communication



Pre-synchronisation



Post-synchronisation

Figure 3: A global picture of our solution.

tion. After $Philosopher_i$ has picked his/her forks up, he or she eats, releases the forks, spends some more time thinking, and the whole process is repeated again.

2.2 Communication

We illustrate the notion of multiparty communication by means of the leader election problem, which is a classic multi-process communication problem that consists of a number of processes that are able to execute an algorithm, but there is no a priori candidate to run it. Therefore, an election under the processes needs to be held. The criterion processes use to select a leader is quite simple: each of them is supposed to have a different natural weight w_i in the system, and the leader is the process P_i satisfying that $w_i = \max_{1 \le j \le n} \{w_j\}$.

The usual solution to this problem, using two–party interactions, consists of arranging the processes in a unidirectional ring where only pairs of neighboring processes can exchange their weights and calculate a local maximum. These maximums are propagated in the ring so that after $n-1$ rounds the global maximum has been calculated. The problem here is that synchronizing the whole set of processes so that each one passes its local maximum at the right moment is quite tricky. If we used multiparty communication, all of the processes would synchronise and have access to the weights other processes have simultaneously. An immediate solution to this problem is shown in figure 2. Here, the multiparty interaction $Elect$ synchronises all of the processes, allow-

ing them to exchange information and decide which one has to be assigned to the role of leader. When several processes synchronise and interact, a temporary global combined state is formed by combining the local states of the processes participating in that interaction so that they can read information in the state of other participants. This way, each process synchronising on $Elect$ can read the weights the other processes have, compute the maximum in parallel, compare it to its own weight and store the result of this comparison in its local variable $leader_i$. After interaction, the one that finds itself having the maximum weight executes the appropriate algorithm.

3 Implementing interactions

The bulk of implementing multiparty interactions consists of the so-called pre–synchronisation, communication and post–synchronisation problems. The former, consists of detecting which interactions are enabled and of resolving conflicts. The communication problem consists of transmitting the piece of information each process needs so that network load is minimum. Finally, the post–synchronisation problem consists of stopping all of the processes participating in an interaction until the others have completed their communication parts.

This section shows the solution to these problems we have implemented[1], and also reports some experimental results that show that our implementation performs quite well in low cost computers.

3.1 Our solution

We have implemented a distributed solution to multiparty interactions where each IP process runs on a different virtual machine, and there is a set of compiler–generated processes that deal with the problems we have just mentioned. Our solution associates a process called manager with each interaction, and there is also a central scheduler. Each manager is responsible for detecting enablement or disablement of its corresponding interaction, and the central scheduler deals with fair selection of interactions.

Each IP process is logically connected to the managers of the interactions it participates in, and they send them messages in order to inform them whether they are readying their associated interactions or not. When a manager detects enablement or disablement, it sends its result to the central interaction scheduler, which, in turn, selects one enabled interaction fairly. In order to detail how our solution works we use the program and the trace we show in figure 3. It consists of two processes P and Q that can exchange the values of their local variables x and y either by participating in interaction A or B, which are permanently in conflict.

Processes do local computations and, when they arrive at a point where they are readying an interaction, they send messages to the interaction managers in order to inform them whether they are readying the interaction they manage or not. These messages are of the form $Readies(b)$, being b a boolean value. Upon reception of these messages, the interaction managers can detect enablement or disablement very easily because they only need to see if all of the processes that are connected to it are readying the interaction they manage or not. Once they have this information, they send it to the interaction scheduler by means of messages of the form $Enabled(b)$, being b a boolean value. It then selects one of the enabled interactions fairly and sends messages of the form $Selected(b)$ to the interaction managers to let them know whether their associated interaction has been selected or not. In any case, the interaction managers pass these messages to the processes that are connected to it, thus completing the pre–synchonisation stage.

After synchronisation, communication takes place. Those processes that have got a message of the from $Selected(true)$ from an interaction manager know that they can execute the corresponding interaction, so they start communication by sending it the data they are responsible for by means of messages of the form $Write(v)$. After all the data has been collected, the interaction manager sends each participating process the piece of information it needs by means of messages of the form $Read(v)$. In our first prototype, communication was more expensive because we used two messages to read data from the interaction manager: a message of the form $Request(x)$ to to inform it we were interested in variable x, and a subsequent message to send its value from the manager to the corresponding process. In our latest version, the manager knows what piece of

[1] Due to space limitations, we only present a detailed description but not a formalisation. The reader who is interested can contact the authors in order to get a copy of our algorithm and its formalisation.

information each process needs and sends it without any need for a $Request$ message.

According to IP semantics, no participant in an interaction can continue until they all have completed their communication parts. We have implemented the simplest solution to enforce this: we use a commit protocol in which every participant sends a message indicating it is finished to the corresponding manager, which waits until the last participant is done and informs then the central scheduler. It then sends messages to let the processes know the interaction is finished and they can continue.

3.2 Fairness

Fairness is an important concept that ensures that every element of a non–deterministic program that is enabled sufficiently often, will eventually progress, i.e., none of them is neglected forever. In the context of IP, fair selection of enabled interactions is the only way to ensure liveliness, termination or eventual response to an event. Notice, for example, that in the program in figure 1, interactions get_fork_i and get_fork_{i+1} are always in conflict when they are both enabled, but only one can be executed. The only way to guarantee that each interaction that is enabled "sufficiently often" will eventually be selected for execution consists of assuming that the underlying conflict resolution mechanism is fair. According to the meaning of "sufficiently often" we have the following levels of fairness: weak, if every element continuously enabled is selected infinitely often, and strong, if every element that is infinitely often enabled is infinitely often selected.

We have implemented strong fairness by associating a priority variable p_a with each interaction a, as suggested in [6]. These variables are initially assigned random values, and the central scheduler selects among the set of conflicting interactions that whose counter has the minimum value (maximum priority). If more than one variable is minimum over the set of priority variables, one of them is uniformly selected. Upon termination of the selected interaction, its associated priority variable is reset to an arbitrary random value while the counters associated with those interactions which were neglected are decreased by 1. This algorithm has been proved correct in [6], but, unfortunately, we have proved that it loses completeness if counters are finite, i.e., there are fair executions that cannot be generated by this algorithm. Please, do contact the authors if you are interested in this theoretical result.

3.3 Experimental results

We have implemented an IP compiler, and the target language we selected was SR (Synchronising Resources) [2], a well–known, widely–available language for writing concurrent programs. Our prototype runs on a network computer composed of several computers running Solaris, AIX and Linux, the platforms we have in our laboratories.

In this section, we report the results of some empirical tests we have carried out in order to find out how our implementation performs. The tests were run on a set of 10 low cost IBM 320H computers running at 25 MHz. They are equipped with 16 Mb of memory, AIX 3.2.5, SR 2.3.1, GNU C 2.4.7, and they are interconnected by means of a 10 Mbps Ethernet LAN. Our test consisted of executing the following program:

TEST :: [$\|_{i=1}^{n}$ P_i], **where**
 P_i ::
  { count: natural := 0 }
  *[count $<$ 500 $\rightarrow$ Int[]; count++; *work 1 sec.*]

It consist of n processes that just synchronise on Int 500 times, and do some work that takes them 1 second. We executed it 15 times in a single machine giving n values from 2 up to 10, i.e., we increased the number of participants in Int from 2 up to 10. We then executed this test assigning a process to each of our machines, thus composing a network computer.

We have also carried out a regression analysis at a 95% confidence level whose results are reported in the table below. It shows that the time our algorithms take increases about 726 seconds each time a new participant is added in the case of a single computer (T_{SC}), whereas the rise is only 423 seconds in a network computer (T_{NC}). The number of interactions per minute also decreases as the number of participants increases, but our network computer executes 9.42 more interactions per minute than our single computer. This approximation is quite accurate as the coefficient of determination R^2 shows. This coefficient ranges in value from 0 to 1, and the higher its value is, the more accurate the approximation is.

In general, these results show that our distributed implementation performs quite well in low cost workstations.

Magnitude	Prediction	R^2
Time	$T_{SC} = 725.93n + 718.78$	0.99
	$T_{NC} = 423.24n + 140.88$	0.83
Int./Min.	$I_{SC} = 20.30e^{-0.19n}$	0.95
	$I_{CN} = 43.69e^{-0.20n}$	0.85

4 Related work

The first algorithms for distributed co-ordination were produced in the context of CSP, and were restricted to two–party interactions. Nevertheless, more recently, the problem of multiparty interactions has become of great interest. Chandy and Misra [5] developed two algorithms that became the basis of Bagrodia's algorithm [4]. In this algorithm, each interaction has an associated manager, which is similar to our distributed solution because it is sent messages when processes are ready to interact and detects enablements. When one of them detects an enabled interaction, a mutual exclusion algorithm is run in order to prevent two different interactions from being executed at the same time. The problem here is that Bagrodia's algorithm assumes that the underlying communication network has only those links connecting the processes that participate in an interaction. This is problematical because it is not always possible to place processes at adequate nodes in a real network.

Several more algorithms have been developed by Garg [8] or Joung and Smolka [9] for different network architectures. In general, these papers also focus on architectural aspects we are not interested in. Instead of making our solution dependent on the underlying network, we have decided to rely on SR for efficient distribution. This makes our algorithms portable, and incorporating strong fairness into them has been very easy, whereas incorporating this notion in other well–known algorithms is rather difficult. At present, the research is centred on implementing stronger fairness assumptions than those provided by the underlying network [3].

As far as we know, IP has been implemented in the laboratory [1], and runs on a transputer–based computer. Unfortunately, the implementation is only intended for terminating IP programs. Ours can be run in virtually any network computer composed of inexpensive workstations and personal computers. Furthermore, it can deal with both terminating and non–terminating programs.

5 Conclusions and future work

In this paper, we have presented a solution to the problem of distributed multiparty interactions. We have also reported some experimental results that show it is effective enough to be used in practical applications. We also think that the solution we have presented is attractive because it is not bound up with the underlying network, and incorporating an algorithm for fair selection of interactions has been straightforward.

At present, we are working on introducing multiparty interaction in the context of CORBA. We agree with the authors of IP in that it will not replace current programming languages, but we think that the notion of multiparty interaction is quite important and it would be desirable for languages such as C++ or Java to support it. This way, we are implementing multiparty interactions using CORBA, which is a middleware that is very successful in the industrial world.

References

[1] A. Adir. *Compiling Programs with Multiparty Interactions and Teams*. PhD thesis, Technion, 1994.

[2] G.E. Andrews and R.A. Olson. *The SR Programming Language*. The Benjamin–Cummings Publishing Company, 1993.

[3] P.C. Attie, I.R. Forman, and E. Levy. On fairness as an abstraction for the design of distributed systems. In *Proceeding of the 10th International Conference on Distributed Computing Systems*, Paris, France, June 1990. IEEE.

[4] R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, September 1989.

[5] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison–Wesley, 1988.

[6] N. Francez. *Fairness*. Springer–Verlag, 1986.

[7] N. Francez and I. Forman. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison–Wesley, 1996.

[8] V.K. Garg and S. Ajmani. An efficient algorithm for multi–process shared events. In *Proceedings of the 2^{nd} Symposium on Parallel and Distributed Computing*, 1990.

[9] Y.J. Joung and S.A. Smolka. A completely distributed and message-efficient implementation of synchronous multiprocess communication. In Pen-Chung Yew, editor, *Proceedings of the 19^{th} International Conference on Parallel Processing. Volume 3: Algorithms and Architectures*, pages 311–318, Urbana-Champaign, Illinois, August 1990. Pennsylvania State University Press.