Implementing Multiparty Interactions on a Network Computer

Rafael Corchuelo, David Ruiz, Miguel Toro and Antonio Ruiz
Dpto. de Lenguajes y Sistemas Informáticos, Universidad de Sevilla
Avda. de la Reina Mercedes s/n, Sevilla 41.012, Spain
{corchu, druiz, mtoro, aruiz}@lsi.us.es

Abstract

Classical client/server interaction primitives such as remote procedure call or rendez–vous *are not adequate when we need to describe the behaviour of three or more processes that need to collaborate simultaneously in order to solve a problem. Multiparty interactions are the key to describe these problems, and there are several languages that use them for the description of reactive systems. In this paper, we show and compare two different fair implementations of this mechanism and also outline the research we are carrying out in an effort to improve them.*

1. Introduction

One of the most popular models for describing and implementing distributed systems is the client/servel model, which emphasises two entities exchanging messages, and, thus, communication. Nevertheless, this approach is not adequate when we need to describe a system where several processes need to collaborate simultaneously in order to solve a problem. A classical example is the leader election problem, where a number of processes need to agree so that only one of them performs a certain task in a system. It is obvious that this problem can be described in terms of client/server primitives, but the key here is coordination among a number of processes, not binary communication. In other words, solving this problem requires the achievement of an agreement among multiple parties.

Multiparty interaction is a way to describe coordination among several processes, and there are several languages that incorporate it. Among them, IP [7] stands out because it is equipped with sound semantics that turn it into a language amenable to formal reasoning, but it is also a viable implementation tool one can use to express abstract solutions that can be compiled and animated. Several algorithms for implementing the multiparty interaction statements IP incorporates have been described in the literature [3, 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 executed 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 and compare two different implementations of the multiparty interaction mechanism IP incorporates that are easily portable and allow for easy incorporation of fairness. It is organised as follows: section 2 recalls the notion of multiparty interaction by means of well–known problems, and presents some changes we have made in IP syntax; section 3 describes two different implementations as well as the algorithm we have implemented to deal with fair selection of conflicting interactions; section 4 glances at other authors' work, and, finally, section 5 shows our conclusions and the work we are planning on doing.

2. Multiparty interactions in a nutshell

In this section, we introduce multiparty interaction in the context of IP. We use the dining philosophers problem in order to illustrate how multiparty synchronisation works, and the leader election problem in order to illustrate communication. We also criticise IP communication mechanism and

propose a slight modification that turns it into a more robust mechanism.

2.1. Preliminaries

We assume that the reader is familiar with IP [7], so we only recall the main concepts. In IP, systems are understood as collections of cooperating 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. IP also provides guarded non–deterministic choice statements $[[]_{i=1}^{n} G_i \rightarrow S_i]$ and guarded non–deterministic loops $*[[]_{i=1}^{n} G_i \rightarrow S_i]$. Guards are of the form $B\&a[\overline{x:=e}]$, where B is a boolean expression and the rest is an usual interaction statement. They are passable, i.e., their corresponding statements can be executed, as long as the boolean expression holds and all of the processes which can eventually execute an interaction statement involving a have arrived at a point where executing that statement is one of their possible continuations. When a process has arrived at such a point, we say that it is readying that interaction, and when an interaction is readied by all of its participants, we say that it is enabled. A process is said to be a participant of interaction a if it has an interaction statement involving a in its body.

2.2. Synchronisation

We illustrate synchronisation by means of the dining philosophers problem, which is a classic multi-process synchronisation problem. It 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. In addition, each philosopher should be able to eat as much as the rest, i.e., the whole process should be fair. This problem is the core of a large class of problems where a process needs to acquire a set of resources in mutual exclusion. This situation can be the case of a debit card system where there is a set of point–of–sales terminals, several computers that hold customer accounts and a number of computers that hold merchant accounts. When a clerk inserts a debit card into a terminal, a three–party interaction needs to be carried out in order to transfer funds from a customer's account to a merchant's account.

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 shall starve. If we used multiparty interactions, each philosopher would pick up his/her two forks at the same time so that no deadlock could 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 interaction. After $Philosopher_i$ has picked his/her forks up, he or she eats, releases the forks and spends some more time thinking.

DIN_PHIL :: [$||_{i=1}^{n}$ Philosopher$_i$ || $||_{i=1}^{n}$ Fork$_i$], **where**

Philosopher$_i$ ::
*[get_fork$_i$[] —→ *eat*; release_fork$_i$[]; *think*]

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

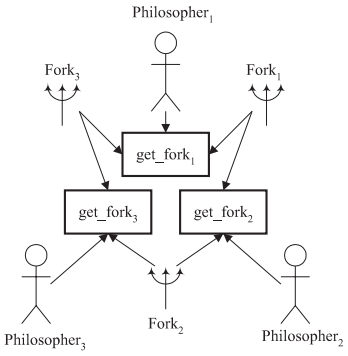

Figure 1. A solution to the dining philosophers problem in IP.

Note that 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 shall eventually be selected for execution consists of assuming that the underlying conflict resolution mechanism is fair. This topic is dealt with in section 3.4.

2.3. Communication

We illustrate the notion of multiparty communication by means of the leader election problem. It has been paid much attention because it is the core of a large class of problems where there are a number of processes 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. This situation can be the case of an initialisation procedure that must be executed initially or a recovery procedure that must be executed after a crash of a system. It is not possible to assign a process to the role of leader because the set of active processes might not be known in advance. 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 (its net address, for instance), and the leader is the process P_i satisfying that $w_i = \max_{1 \leq j \leq n} \{w_j\}$. Notice that this is a clear example where a number of processes need to collaborate simultaneously in an n–party interaction because there is absolutely no way to select a leader if a process does not have information about the weights of the whole set of processes.

An immediate solution to this problem is shown in figure 2. The multiparty interaction $Elect$ synchronises all of the processes, allowing 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 so that each process can read information in the state of other participants of that interaction. Processes need to know which variable has the information they need and exchange information by means of a parallel assignment where the variables on the left are local to the process executing the interaction statement, whereas the expressions on the right have access to variables in the global combined state. This way, each process computes the maximum in parallel, compares it to its own weight and stores the result of this comparison in its local variable $leader_i$. After interaction, the one that finds itself having the maximum value executes the appropriate algorithm.

Thus, this is a symmetric inter–process communication construct involving an arbitrary number of participants, and it is a different view of communication, which is more traditionally broken into sending and receiving information.

2.4. Communication robustness

Unfortunately, IP uses a global naming scheme, thus avoiding two different processes from having local variables with the same names. We think that this scheme is not well–engineered because it does not ensure that the processes participating in an interaction have initialised the variables other processes need, and we cannot use local names. Therefore, we have decided to change IP syntax so that a process can own a local, private state with local, private names.

LEADER :: [$\|_{i=1}^{n}$ P_i], **where**

P_i ::
  $\{w_i$: natural; $leader_i$: boolean$\}$
  w_i := *a weight*;
  Elect[$leader_i$:= (w_i = $\max_{1 \leq j \leq n} \{w_j\}$)];
  [$leader_i$ → *execute algorithm*].

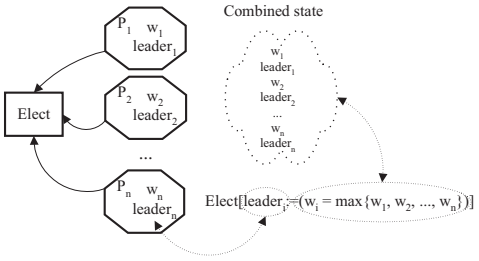

Figure 2. A solution to the leader election problem in IP.

Figure 3 shows our changes: first of all, interactions have to be explicitly declared, and each one owns a local state which is equivalent to IP global, combined state (notice that there is no need for indexed variables now); secondly, each interaction is given a set of participating processes synchronising on it; thirdly, each process is responsible for initialising a part of the interaction state; finally, communication parts in interaction statements have been decomposed into two parts separated by an "&": the first one initialises the interaction state, and the second one allows synchronised processes to exchange information. These modifications allow us for easy implementation, and the compiler can catch many errors that would not be caught otherwise.

3. Implementing multiparty interactions

The bulk of implementing distributed multiparty interactions consists of the so-called pre–synchronisation, communication and post–synchronisation problems. The former problem consists of detecting which interactions are enabled and selecting among them which one should be executed. The communication problem consists of transmit-

LEADER :: [$\|_{i=1}^{n}$ P_i], **through**

Elect[h: array(1..n) of natural]
among P_i **writes** h(i) ($i = 1, 2, \ldots, n$),

where

P_i ::
{w: natural; leader: boolean}
w := *a weight*;
Elect[h(i) := w & leader := (w = $\max_{1 \le j \le n} \{h(j)\}$)];
[leader $\rightarrow$ *execute algorithm*].

Figure 3. Changes in IP syntax.

ting 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.

We have implemented an IP compiler, and the target language we selected was SR [2], a well–known, widely–available language for writing concurrent programs that is based on resources and operations. Resources encapsulate processes and data, and operations provide the primary mechanism for client/server interaction. It supports local and remote procedure calls, *rendez–vous*, message passing, dynamic process creation, semaphores, and makes distribution of processes and data extremely easy. It is suitable for writing parallel, distributed programs for both shared– and distributed–memory machines, including DEC, HP, IBM, NeXT, Silicon Graphics, Sequent Symmetry or Sun.

Our prototype runs on a network of workstations and personal computers running Solaris, AIX and Linux, the platforms we have in our laboratories. It is not bound up with the underlying network architecture, so it is very portable and can be compiled on virtually any UNIX–like system. We use network computers composed of several virtual machines, a term that comes from SR and describes a collection of resources located on a physical machine. Several virtual machines can be hosted by the same physical machine, but the way they communicate is transparent, thus allowing for easy distribution on a heterogeneous network while preserving effectiveness.

3.1. Our centralised solution

Our basic compiler produces code that implements a centralised solution where each process runs on a different virtual machine, and there is an interaction scheduler which solves the pre– and post–synchronisation problems and several interaction managers which solve the communication problem. Figure 4 depicts our solution in the case of an election under two IP processes[1], and figure 5 shows a detailed message trace.

Processes do local computations and, when they arrive at a point where they are readying an interaction, they send messages to the interaction scheduler to inform it about the set of interactions they are readying. These messages are of the form $Readies(P, I)$, being P the name of a process and I the set of interactions it is readying. Upon reception of these messages, the interaction scheduler detects enabled interactions, selects one of them fairly and sends messages to the processes to inform them about which interaction has been selected for execution. This message is of the form $Selected(I^{\star})$, where $I^{\star}$ is the name of the selected interaction.

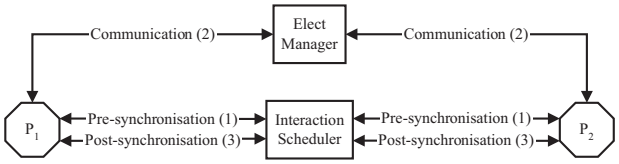

Figure 4. A centralised leader election.

After synchronisation, communication takes place. The changes we have made in IP original syntax and the new syntactic structure of interaction statements have been designed so that it is rather easy to determine data communication requirements. Thus, we have implemented a simple solution where, after informing a process which interaction has been selected, it sends the data it is responsible for to the corresponding interaction manager by means of messages of the form $Write(x, v)$, being x the name of a slot and v its value. When every slot has been initialised, the interaction manager sends a message to the processes connected to it and they, in turn, use messages of the form $Read(x)$ to read the slots they need.

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 ensure this: we use a commit protocol in which every participant sends a message of the form $Finished(P)$ indicating it is finished to the scheduler, which waits until the last participant is done and sends then messages of the form $Continue()$ to let them know they can continue.

[1] We distinguish between IP processes and compiler-generated processes such as schedulers or managers.

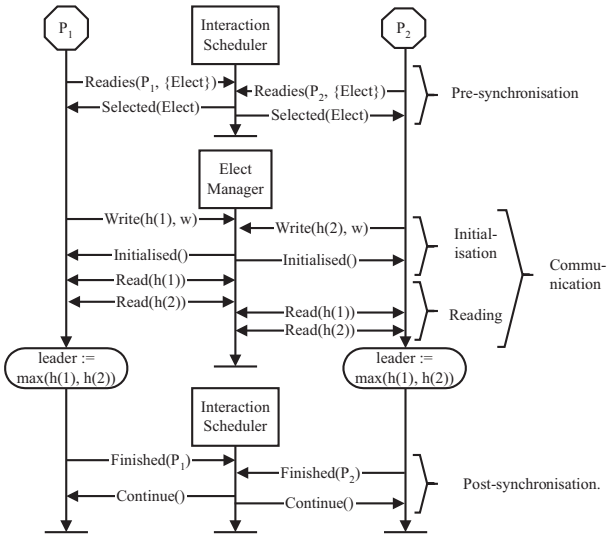

Figure 5. Message trace in a leader election.

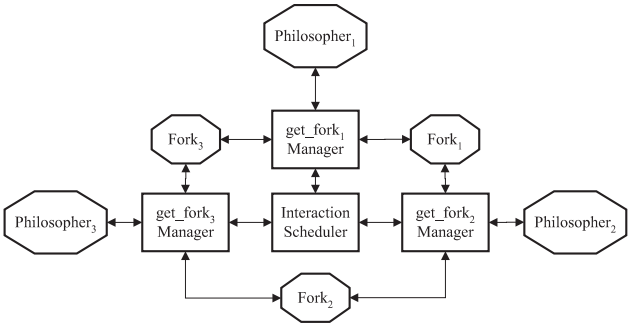

Figure 6. A distributed philosopher's dinner.

3.2. Our distributed solution

Our centralised solution works quite well, but the interaction scheduler does a lot of work and uses complex data structures in order to store information about the whole set of interactions and their participants. This work can be distributed among interaction managers so that they do not only care about communication but also about detecting enablement. This way, detecting enablement is quite easy, because each interaction manager does only need to care about the set of processes participating in the interaction it manages. Notice that several interactions might be enabled at the same time, so managers need to agree so that only one interaction is executed at the same time. In other words, an election under the managers should be held, but we obviously cannot use multiparty interactions because SR does only provide client/server primitives. The solution we have implemented is simple: we use a central interaction scheduler which is sent information about enablement or disablement of interactions and selects which one should be executed so that no–one is neglected forever and the whole process is fair.

Figure 6 depicts our solution in the case of three philosophers who are trying to pick up their forks. Each process is connected to the managers of the interactions it participates in, and they send messages to these managers in order to inform them whether they are readying an interaction or not. When a manager detects enablement or disablement, it sends its result to the interaction scheduler which, in turn, selects one interaction fairly.

In general, distributing the pre–synchronisation problem among a set of managers makes this solution more efficient because the algorithm and the data each process has to manage is simpler than in our centralised solution. Our experimental results in section 3.5 support this conclusion.

3.3. Optimising communication

Communication should be as inexpensive as possible. Beyond the simple scheme we have just shown, many optimisations are possible. In this context, the simplest consists of determining what expressions can be locally calculated in an interaction manager so that it can send its result instead of the values needed to calculate it.

To illustrate this optimisation, we consider again the problem in figure 3 and the statement:

$$\text{Elect[h(i) := w \& leader := (w = } \max_{1 \le j \le n} \{\text{h}(j)\})]$$

In the message trace shown in figure 5, it is clear that four messages are needed to send P_1 and P_2 the information they need. Nevertheless, there is no need for four messages and two processes wasting time calculating exactly the same result. If the interaction manager calculated the result of the expression $\max_{1 \le j \le n} \{\text{h}(j)\}$ locally, network load would have been reduced significantly to only two messages. In general, in a system composed of n processes, it can be reduced from n^2 to n messages.

We can also optimise the reading phase. Notice that two messages are needed to read a slot in the scheme we have presented: a message requesting it, and another to transmit its value. Nonetheless, the same result can be achieved with only one message if the interaction manager knows what slot each process needs and sends them after the last slot has been initialised.

3.4. Fairness

Fairness is an important class of liveliness properties that requires that every element of a non–deterministic program that is enabled sufficiently often, shall eventually progress. For instance, consider the following multi–choice statement:

[true → skip [] true → x := x + 1]

It states that it does not matter whether x is increased or not, and an implementation that does not take into account the second choice is, in principle, as correct as another which selects options arbitrarily. Nevertheless, if we incorporate this small fragment into the following loop, assuming fair selection of enabled choices is the only way to guarantee termination.

*[x $\leq$ 10 → [true → skip [] true → x := x + 1]]

In the context of IP, fair selection of enabled interactions is the only way to ensure liveliness, termination or eventual response to an event. Consider, for instance, the following fragment where we define a set of client processes P_i and a semaphore–like process SEM clients use to ensure mutual exclusion when they enter their critical regions.

SEM :: *[$[]_{i=1}^{n}$ Enter$_i$[] → Exit$_i$[]]
P$_{i=1,2,\ldots,n}$ :: *[Enter$_i$[] → *critical region*; Exit$_i$[]]

Notice that process SEM is initially ready to participate in any interaction, the difference being that if $Enter_i$ is selected process P_i enters its critical region whereas the other processes block until P_i exits it and the top level is reached again. Unless a fairness assumption is taken into account, an implementation that tests interactions from 1 up to n would fulfil IP semantics but would neglect processes other than P_1 from entering their critical regions.

In general, a fairness assumption guarantees that every interaction which is enabled "sufficiently often" shall eventually be selected for execution. According to the meaning of "sufficiently often" we have the following two main 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 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[2].

3.5. Experimental results

In this section, we report the results of some empirical tests we have carried out in order to find out how our two implementations perform. The tests were run on an inexpensive i486 machine running at 120 MHz that is equipped with 32 Mb of memory, Linux 2.0, and SR 2.3.1.

The test consisted of executing the program in figure 7, which consists of $n+1$ processes and an interaction having a natural slot which is randomly initialised by process R. The other processes just synchronise and read this slot. We executed this test 15 times giving n values from 1 up to 14, i.e., we increased the number of participants in int from 2 up to 15.

TEST :: [R || $||_{i=1}^{n}$ P$_i$], **though**

int[x: natural] **among** R **writes** x, P$_i$ (i=1, 2, . . . , n)

where
R ::
{ count: natural }
count := 0;
*[count < 5000 → int[x := random() &];
count := count + 1]
P$_i$ ::
{ count: natural; y: natural }
count := 0;
*[count < 5000 → int[& y := x]; count := count + 1]

Figure 7. Our test program.

Figure 8 shows the time the test took to complete (E) and number of interactions per second (I) that were executed in each case. I obviously, decreases as the number of participants increases, but it is clear that the speed achieved by our distributed solution is always greater than the speed of our centralised solution. We have also carried out a regression analysis that points out that these magnitudes can be accurately approximated by means of the expressions in table 1. This study was done at a 95% confidence level,

[2]Contact the authors if you are interested in this theoretical result.

and shows that the number of interactions per second, for instance, decreases at approximately the same rate of reduction in both versions, the difference being that the expected value is about 44 greater in our distributed version. 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. This analysis also points out that the user time (U) our centralised implementation consumes increases 4.39 times as fast as the distributed solution does. This is because the amount of work the central scheduler needs to do increases as the number of processes participating in an interaction increases. Nevertheless, system time (S) is very similar and small in both cases, so its influence is not significant.

Centralised Version

Prediction	R^2
$I = -19.28N + 263.85$	0.85
$E = 8.78N - 2.14$	0.98
$U = 8.78N - 2.13$	0.98
$S = 0.10N - 0.04$	0.95

Distributed Version

Prediction	R^2
$I = -19.66N + 307.73$	0.88
$E = 4.39N + 6.79$	0.99
$U = 4.39N + 6.80$	0.99
$S = 0.09N - 0.02$	0.96

Table 1. Regression analysis.

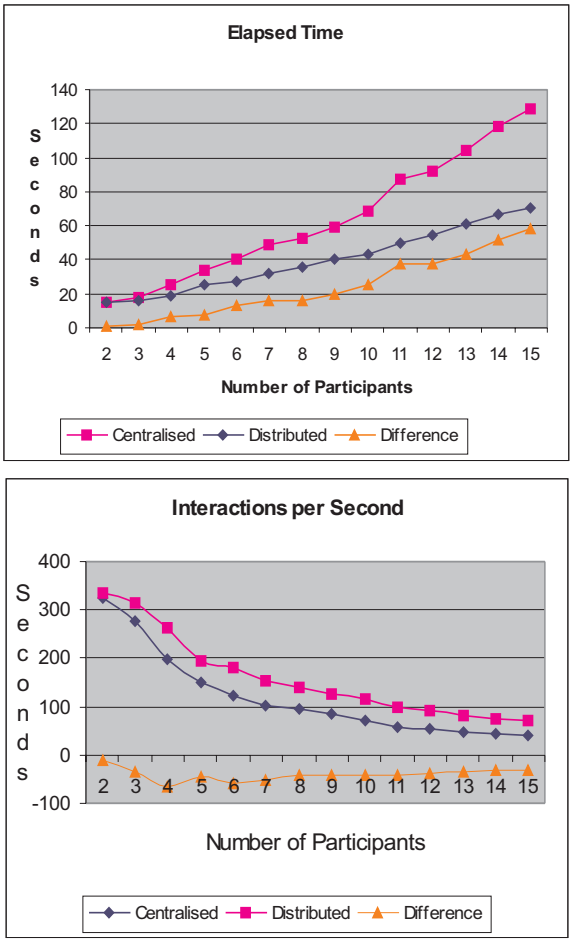

Figure 8. Experimental results.

4. Related work

Recently, the problem of multiparty interactions has become of great interest, and one of the most outstanding results is the algorithm by Bagrodia [3]. In this algorithm, there are interaction managers associated with each interaction, and they are similar to those in our distributed solution because they are sent messages when processes are ready to interact and detect 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. The significance of 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 for different network architectures [8, 9], but, in general, those solutions 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 the algorithms we have mentioned is rather difficult. At present, the research is centred on implementing stronger fairness assumptions than those provided by the underlying network.

As far as we know, IP has been implemented in the laboratory by Adir [1], and runs on a transputer–based computer. Unfortunately, the compiler is only intended for terminating IP programs. Our compiler can be run on virtually any UNIX–like system, and the programs it produces can be run

on heterogeneous networks composed of inexpensive workstations and personal computers. Furthermore, it can deal with both terminating and non–terminating programs. In any case, we have not got access to Adir's implementation, so no empirical comparison has been possible.

5. Conclusions and future work

In this paper, we have presented two different solutions to the problem of distributed multiparty interaction. We have also reported some experimental results that show that our distributed solution is more effective than our centralised solution. The result was predictable because the interaction scheduler of our centralised solution does a lot of work that has been easily distributed among a set of interaction managers in our distributed solution. These managers execute simpler algorithms and use simpler data structures, so they are more efficient. We think that the solution we have presented is attractive because it is not bound up with the underlying network. We rely on SR for efficient distribution of data and processes, and the compiler we have produced is portable to any UNIX–like system.

The main problem with the solutions we have presented is that they require the whole set of enabled interactions to be known before one of them is selected for execution. Roughly speaking, this implies that the speed our implementation can achieve is bounded up with the speed of the slowest process in a system, and this should be improved. Speeding up the system is easy, as reported in [7], because we only need to execute interactions as soon as they are detected to be enabled, but this solution can be clearly unfair and can show conspiratorial behaviours. This behaviour occurs when a number of fast processes prevent an interaction from being executed because several slow processes participating in it are not readying it at the same time than the fast processes. In other words, executing interactions as far as they are detected to be enabled can prevent an interaction from being executed because the processes participating in it run at very different speeds. We are working out an algorithm that solves this problem, but, for the time being, we have not implemented it.

Currently, we are also developing $\mathrm{IP}_{IC}(\chi)$ [4], which is an assembler language intended to be used to implement high–level system specification languages such as LOTOS or TESORO [10]. These languages are based on multiparty interaction, the difference being that they use constraints in order to state the concrete set of interactions a process is readying. Introducing constraints complicates the implementation, but we can still reuse the solutions we have presented in this paper. The main problem is fairness, because if constraints are used to state interaction, a constraint solver able to deal with it is needed. The problem here is that very little has been said about fairness and constraints. Our research is centred on solving this problem, and we have developed an algorithm for dealing with weak fairness in the context of $\mathrm{IP}_{IC}(\chi)$ [5].

In the future, we are going to introduce multiparty interaction in the context of CORBA. We think that IP shall not replace current programming languages, but we think that the notion of multiparty interaction is quite important because it allows for simpler programs than client/server primitives and enjoys a higher level of abstraction. Therefore, it would be desirable for languages such as C++ or Java to support it, and introducing it as a CORBA service would be the best way.

References

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

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

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

[4] R. Corchuelo, O. Martín, and M. Toro. Symbolic constraints as a means for multiparty synchronisation and communication. *Journal on Computers and Information*, May 1999.

[5] R. Corchuelo, O. Martín, M. Toro, A. Ruiz, and J. Prieto. Weak fairness in the context of constraint-based multiparty interactions. In *Proceedings of the Simposio Español de Informática Distribuída SEID'99*, pages 99–107, Santiago de Compostela, Spain, Feb. 1999.

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

[10] J. Troyano, J. Torres, and M. Toro. TESORO: A technique for distributed systems specification. In *Proceedings of the 3^{rd} Euromicro Workshop on Parallel and Distributed Processing*, pages 563–570, San Remo (Italy), Jan. 1995.