An Order-Based, Distributed Algorithm for Implementing Multiparty Interactions

José Antonio Pérez, Rafael Corchuelo, David Ruiz, and Miguel Toro

Universidad de Sevilla, Dpto. de Lenguajes y Sistemas Informáticos
Avda. de la Reina Mercedes, s/n. Sevilla 41.012 (Spain)
jperez@lsi.us.es, www.lsi.us.es/~tdg

Abstract. Multiparty interactions have been paid much attention in recent years because they provide the user with a useful mechanism for coordinating a number of entities that need to cooperate in order to achieve a common goal. In this paper, we present an algorithm for implementing them that is based on the idea of locking resources in a given order. It improves on previous results in that it can be used in a context in which the set of participants in an interaction cannot be known at compile time, and setting up communication links amongst interaction managers is costly or completely impossible.

Keywords: Multiparty interactions, coordination algorithms, open systems, α–core.

1 Introduction

Since Hoare's workon communicating sequential processes [Hoa85], interactions have become a fundamental feature in many languages for distributed computing. Often, they are biparty because they do only involve two entities that need to synchronise before exchanging data, but this concept can be easily extended to an arbitrary number of entities that need to agree and cooperate to achieve a common goal. These interactions are usually said to be multiparty, and they provide a higher level of abstraction because they allow to express complex cooperations as atomic units.

A taxonomy of languages offering linguistic support for multiparty interactions can be found in [JS96], and one of the most recent ones is IP [FF96]. It has also attracted the attention of the designers of the well–known Catalysis method [DW99], which is a next generation approach for the systematic UML–based, business–driven development of component–based systems. Catalysis has been used by Fortune 500 companies in fields including finance, telecommunication, insurance, manufacturing, embedded systems, process control, flight simulation, travel and transportation, or systems management, thus proving the adequacy of this novel interaction model in so different application domains.

In this paper, we present an algorithm that deals with the multiparty synchronisation problem. Our solution improves on others in that it does not require the set of entities participating in an interaction to be defined at compile time,

there is no need for communication links amongst interaction managers, and entities are not directly dependent on each other, which makes our solution suited for open contexts.

The organisation of the paper is as follows: the interaction model we deal with is presented in Section 2; the algorithm is presented in Section 3; we glance at other author's work in Section 4, and finally, our main conclusions are drawn in Section 5.

2 The Multiparty Interaction Model in a Nutshell

In this section, we introduce the main features of the multiparty interaction model we deal with. It is quite usual in the languages described in [JS96], being the only difference that the identity of the entities that can participate in an interaction is only known at run time. We think that this feature makes the model more general and introduces a number of problems that have not been addressed by other authors.

In this context, the terms *entity* or *participant* refer to any computing artifact that is able to perform local computations and decides autonomously when it is interested in participating in a number of interactions. Multiparty interactions are usually provided as guards in multi-choice commands, so that an entity may be willing to participate in several interactions at the same time, although only one shall be finally executed. This model is thus well-suited to coordinate processes, threads, objects or components, as well.

Each interaction is identified by an *interaction name*, and has a fixed number of participants referred to as its *cardinality*. Often, we refer to interactions with cardinality n as n–party interactions. In general, n can be assumed to be greater or equal than two, although the results we show work well with single-party interactions. For an n–party interaction to become *enabled*, n participants need to be *offering participation* in it. Once several entities have been coordinated, communication depends completely on the constructs provided by the language under consideration, but most have been designed so that it is relatively easy to determine data communication requirements.

Roughly speaking, a multiparty interaction can be viewed as a set of data exchange actions that need to be executed jointly and coordinatedly by a number of entities, each of which must be ready to execute its own action so that the interaction can occur. An attempt to participate in an interaction delays an entity until all other participants are available, and after an interaction is executed, the participating entities exchange some data and continue their local computations on their own accord.

It is worth noting that an interaction being enabled does not amount to its execution. Figure 1 depicts a simple system composed of four participants, a two–party interaction and a three–party interaction. Notice that participant P_1 is offering participation in I_1, whereas P_3 and P_4 are offering participation in I_2, and P_2 is offering participation in either I_1 or I_2. This means that these interactions cannot be executed simultaneously and they are said to be *conflicting*

ones. Thus an election needs to be held to decide which one should be executed. The one that is executed is referred to as the *winner interaction* and the other as the *loser interaction*.

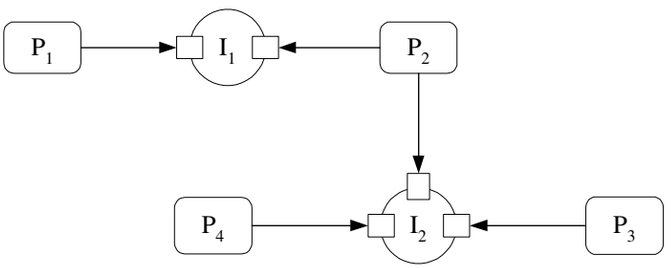

Fig. 1. A system composed of four participants and two conflicting interactions.

3 Our Proposal: α–core

In this section we present a brief description of our algorithm. Its full formal description including correctness proof, as well as a performance analysis and a performance comparison with other proposals can be obtained from the authors.

We call our algorithm α–core, because it is the core of a more general algorithm that we called α. An overview of the α algorithm can be found at [PCRT02]. The α–core algorithm has been devised to detect enabled interactions and select as many as possible amongst conflicting ones in a simple, effective way. It considers an entity that offers participation in more than one interaction as a *shared resource* amongst a number of coordinators responsible for managing those interactions. For an interaction to be executed, its corresponding coordinator must ensure exclusive access to all of its participants, i.e., coordinators must compete for their shared participants so that they can execute the interaction they manage.

We assume that each entity participating in an interaction runs independently from each other, and that each interaction is managed by a different independent *coordinator*. The communication between coordinators and participants is modeled by means of *asynchronous messages* because this communication primitive is available on almost every platform.

The overall picture of the main ideas behind α–core may be sketched by means of a simple example. Figure 2 shows a typical scenario for the system depicted in Figure 1, and Table 1 describes the messages α–core uses. We only need to make three assumptions about the underlying message passing system: (i) every message must be received sooner or later, (ii) messages sent from the same origin to the same destination must be processed in the same order they were sent, and (iii) messages are stored in a FIFO queue until they are processed.

Table 1. Messages used by α–core.

Message	**Description**
$ACKREF$	Message sent from a coordinator to acknowledge it has got a $REFUSE$ message from a participant.
$LOCK$	Message sent from a coordinator to a shared participant to request exclusive access to it.
$OFFER$	Message sent from a participant to a number of coordinators to offer them participation in the interactions they manage.
OK	Message sent from a participant to a coordinator to notify that it grants it exclusive access. This message is sent as a reply to a $LOCK$ message.
$PARTICIPATE$	Message sent from a participant to only one coordinator to inform it that it is only interested in the interaction it manages.
$REFUSE$	Message sent from a participant to a coordinator to cancel an offer.
$START$	Message sent from a coordinator to a locked participant to notify it that the interaction it manages may start.
$UNLOCK$	Message sent from a coordinator to a shared participant to release exclusive access.

In this scenario, P_1 is the first entity ready to participate in I_1. Since it is only interested in this interaction, it notifies its offer to coordinator I_1 by means of a $PARTICIPATE$ message, and then waits for a $START$ message before beginning the execution of this interaction. Assume that P_2 gets then ready to participate in either I_1 or I_2. Since it offers participation to more than one coordinator, it sends two $OFFER$ messages by means of which the coordinators that receive them can infer that this participant is shared with others, although they need not know each other directly.

As soon as coordinator I_1 receives the offers from its two participants, it detects that I_1 is enabled and tries to lock P_2 by sending it a $LOCK$ message. There is no need to lock P_1 because this participant is interested in only one interaction; thus it is not a shared participant. Assume that P_3 and P_4 decide then to participate in I_2 and send a $PARTICIPATE$ message to its coordinator. It then detects that it is also enabled, and tries to lock P_2, too.

Unfortunately, the $LOCK$ message sent to P_2 by I_1 is received before the $LOCK$ from I_2 arrives. Thus, P_2 notifies I_1 that it accepts to be locked by means of an OK message, but it does not acknowledge the lock message received later from coordinator I_2, but records it, just in case I_1 cannot be executed. Coordinator I_2 waits until it gets an answer from P_2 before going on, thus it cannot lock a participant if another lock is still pending. When I_1 receives the OK message, it knows that it has exclusive access to its shared participant, and thus sends a $START$ message to P_1 and P_2. When the shared participant P_2 receives the $START$ message from I_1, it knows that it can execute that

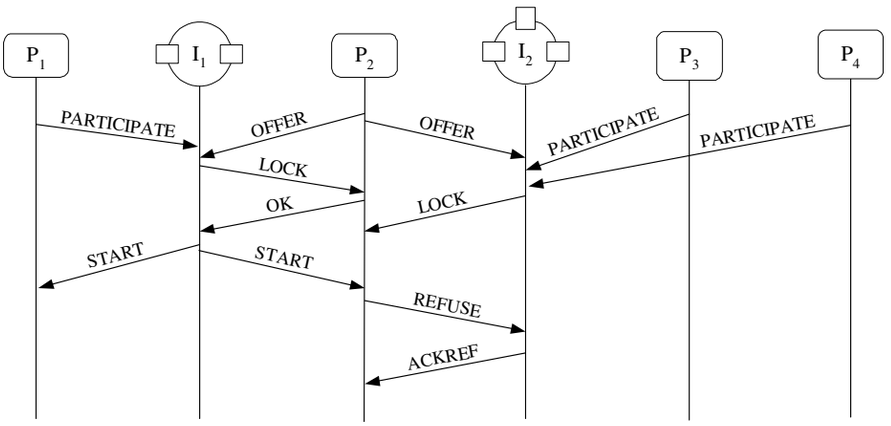

Fig. 2. A possible scenario for the system in Figure 1.

interaction and cancels the offer made to I_2 by sending it a $REFUSE$ message that is acknowledged by means of an $ACKREF$ message.

Therefore, the idea behind α–core consists of locking shared participants, and an interaction may be executed as long as its corresponding coordinator has acquired exclusive access to all of its shared participants. The problem is that locks need to be carried out carefully in order to avoid deadlocks. We use an idea proposed in [EGCS71]: α–core assumes that coordinators may sort their participants according to a given immutable property, e.g., their net address, their Universally Unique Identifier (UUID) [Rog97] or their Universal/Uniform Resource Identifier (URI) [BLFM98], so that lock attempts are made in increasing order. This idea was proven not to produce deadlocks and it is quite effective.

At a first glance, one might think that a solution to the leader-election problem [FF96] suffices to select amongst conflicting coordinators, but it is not enough because the set of conflicting interactions is not static, but changes as new conflicting interactions become enabled. Another key point is that we do not want coordinators to know each other in order to make our solution suitable for open systems, but usual solutions to the leader-election problem, e.g., organising entities in rings, would require neighbouring coordinators to know each other.

4 Related Work

Several solutions to implement multiparty interactions have been proposed. For instance, the simplest one was presented in [FF96], and it consists of using a central manager responsible for all interactions. Each participant sends it a *ready* message when it arrives at a point where it needs to coordinate its activities with others, and the manager uses a counter per interaction to determine when they become enabled on reception of a ready message. If several ones become enabled by the same *ready* message, a random variable may be used to select amongst

them. Although this algorithm may be suitable for certain systems, the concerns of performance and reliability argue for a distributed solution.

The first algorithms for distributed coordination were produced in the context of CSP, and were restricted to two–party interactions. Later, the problem of multiparty interactions became of great interest, and Chandy and Misra [CM88] developed two algorithms that became the basis of Bagrodia's algorithm [Bag89]. Currently, this algorithm is one of the most cited in this field.

Bagrodia presented a distributed version of the basic centralised algorithm called EM that uses a number of interaction managers, each one responsible for managing a subset of interactions. When a participant wants to participate in a number of interactions, it sends *ready* messages to the corresponding managers, that have to coordinate in order to achieve mutual exclusion of conflicting interactions. In this case, mutual exclusion is achieved by means of a circulating token that allows the manager having it to execute as many non–conflicting interactions as possible.

Having a circulating token has several drawbacks because it amounts to additional network load, even if no interaction is enabled. The token also needs to circulate amongst managers in a given order, thus organising them in a unidirectional ring, which may lead to a situation in which a manager can never execute one of the interactions it is responsible for because it never gets to have the token at the right time.

In α–core, there is a coordinator, i.e., a manager, per interaction and the exclusion is achieved by locking participants in a given order. In [EGCS71], this idea was proven to be correct and that it does not produce deadlocks. α–core also reduces the possibility of an interaction never being executed because when a participant is in the $LOCKED$ state and it is unlocked by a coordinator, it selects the next prospective winner coordinator randomly, thus introducing some variability in the exclusion problem. In addition, α–core does not require the set of participants to be known in advance, which is an interesting feature because it allows our algorithm to be used in open systems.

A modified version of the basic algorithm was also presented in [Bag89]. It was called MEM, and it combines the synchronisation technique used in EM with the idea of using auxiliary resources to arbitrate between conflicting interactions. The exclusion problem is solved by mapping the multiparty interaction problem onto the well-known dining philosophers problem or onto the drinking philosophers problem [CM84]. In this extension, conflicting managers are considered to be philosophers that need to acquire a single fork placed between them in mutual exclusion. The fork is then a shared resource whose acquisition guarantees mutual exclusion amongst conflicting interactions. A difficulty arises in MEM because between enablement detection and acquisition of forks, a conflicting interaction may have started executing. The complex part of MEM is the way it uses the information communicated during the mutual exclusion to detect that an enablement is no longer current.

MEM has an important drawback because the number of forks a manager has to acquire to guarantee mutual exclusion increases as the number of conflicting interactions increases. This implies that the probability of acquiring all the forks decreases accordingly. In α–core, there is no need for virtual resources. Instead

shared processes are considered to be resources that coordinators need to acquire. Thus, the probability of gaining mutual exclusion decreases as the number of shared participants increases, but, in general, this is less problematical because most practical multiparty interactions are three– or four–party. Interactions with a higher cardinality are not usual, but, even in those cases, there is an upper, small bound on the maximum number of shared participants.

Both EM and MEM need managers to know each other, which may be considered a drawback in open systems where new interactions may become available unexpectedly. In [JS98], another algorithm that does not use interaction managers and also deals with fairness concerns was recently presented. Unfortunately, participants also need to know each other at compile time and, thus, that algorithm is not applicable to open systems.

In a chapter of [CRT+99], we presented a previous solution to implement multiparty interactions that also had coordinators, and a central scheduler responsible for selecting amongst conflicting ones. Although this solution was suitable for some problems in the traffic control area, the central conflict resolutor is problematical in the general case.

Currently, we are using α–core to implement a run–time system for supporting the Aspect–Oriented language $\mathcal{CAL}$ [PCRT01,CPT00], which is aimed at increasing the level of abstraction of a program by considering the concurrent behaviour of components as an aspect where multiparty interactions are the sole means for synchronisation and communication.

5 Conclusions and Future Work

The multiparty interaction model captures three main issues in the design of distributed systems: synchronisation, communication and exclusion. In this paper, we have presented a solution to implement this interaction model, but we have not addressed the problem of communication because this feature is tightly coupled with the language we are using.

A variety of solutions exist in the literature, and ours is innovative in the sense that we do not require the set of active entities in a system to be fixed and known in advance. In addition, participating entities or interactions do not depend on each other in our solution, which is an important drawback in other proposals. This way, our solution can be easily applied in open contexts such as Internet applications where multiparty interactions can be used to coordinate an arbitrary number of entities.

The algorithm relies on an idea that was introduced and proven to be correct years ago in the field of operating systems. Although this idea did not work well in this field because resources in an operating system are difficult to sort and usually cannot be requested in increasing order, it has been successfully applied in α–core. Due to the short length of this paper, we could not present our experimental results, but we think that those results are satisfactory, and that our algorithm achieves a good performance.

References

[Bag89] R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, September 1989.

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform resource identifiers (URI): Generic syntax, August 1998.

[CM84] K.M. Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison–Wesley, 1988.

[CPT00] R. Corchuelo, J.A. Pérez, and M. Toro. A Multiparty Coordination Aspect Language. *ACM SIGPLAN*, 35(12):24–32, December 2000.

[CRT+99] R. Corchuelo, D. Ruiz, M. Toro, J.M. Prieto, and J.L. Arjona. *A Distributed Solution to Multiparty Interaction*, pages 318–323. World Scientific Engineering Society, 1999.

[DW99] D. F. D'Souza and A.C. Wills. *Objects, Componentes, and Frameworks with UML: The Catalysis Approach*. Addison–Wesley, 1 edition, 1999.

[EGCS71] M.J. Elphick E. G. Coffman and A. Shoshani. System Deadlocks. *Computer Surveys*, 3:67–68, June 1971.

[FF96] N. Francez and I. Forman. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison–Wesley, 1996.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processess*. Prentice Hall, 1985.

[JS96] Y. J. Joung and S. A. Smolka. Strong interaction fairness via randomization. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 475–483, Hong Kong, May 1996. IEEE Computer Society Press.

[JS98] Y.J. Joung and S.A. Smolka. Strong interaction fairness via randomization. *IEEE Transactions on Parallel and Distributed Systems*, 9(2), February 1998.

[PCRT01] J.A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. A framework for aspect–oriented multiparty coordination. In *New Developments in Distributed Applications and Interoperable Systems*, pages 161–174. Kluwer Academic Publishers, 2001.

[PCRT02] J.A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. An enablement detection algorithm for open multiparty interactions. In ?, editor, *Proceedings of the Symposium on Applied Computing SAC'02*, number ? in ?, page ?, Madrid, Spain, March 2002. ?, ? Appearing soon.

[Rog97] D. Rogerson. *Inside COM*. Microsoft Press, 1997.