Prototype platforms for distributed agreements [1]

Alberto Baragatti, Roberto Bruni, Hernán Melgratti,
Ugo Montanari and Giorgio Spagnolo

Dipartimento di Informatica, Università di Pisa, I-56127 Pisa, Italy,
{baragatt,bruni,melgratt,ugo,spagnolo}@di.unipi.it

Abstract

We present a prototype application for coordinating distributed agreements in multi-parties negotiations, where participants can dynamically join ongoing negotiations and where participants know only those parties they have interacted with. Our prototype is tailored to Ad-Hoc network scenarios involving the assignment of tasks for a rescue team operating over disaster areas. Our application is based on asynchronous communication and it exploits the D2PC protocol for committing or aborting a negotiation. Parties have been developed both in Jocaml+Perl and Polyphonic C$^\sharp$. The implementation of the commit protocol allows components of both types to participate within the same negotiation.

Key words: Multiway transactions, Compensations, distributed negotiations

1 Introduction

When developing distributed applications, in particular when combining independent, heterogeneous components, the orchestration of agreements emerges as a typical problem. Hence, patterns and frameworks to handle distributed negotiations become essential [8]. In this paper we introduce an approach to orchestrate agreements whose structure may change dynamically and we present a "proof of concept" prototype application, where some parties are written in Jocaml and Perl, and others in Polyphonic C$^\sharp$.

As a running case study we consider a typical scenario within the context of *Mobile Ad-hoc NETworks* (MANETs), i.e. networks where agent mobility coexists with dynamic infrastructures and net topology. MANETs are typical of

[1] Research supported by the Italian MIUR within the framework of the IS-MANET project (Software infrastructures for mobile Ad-Hoc networks in difficult environments).

wireless scenarios for small mobile units and their infrastructures (emergency teams, medical teams, security units, press and information groups, hi-tech research and business meetings), where many local agents are involved (laptops, PDAs, and last generation mobile phones). Our case study considers a rescue unit composed by a central base and several teams, each of them having a *leader* and several *operators*. Roughly, the idea is that after having exchanged several messages, each member can either decide to commit her/his negotiated involvement in the task, or to abort the negotiation when the assigned activity cannot be performed. Note that team members often have a limited knowledge about the other participants in the task, i.e., they only know those members they have interacted with (by sending or receiving messages).

In order to implement such kind of agreements in a fully distributed way, we rely on the *distributed two phase commit* protocol (D2PC) proposed in [3]. The D2PC has been specified in the Join calculus [7] by taking advantage of its main features, namely, asynchronous communication, reflexive description of processes, creation of fresh names, and name mobility. Consequently, the D2PC can be straightforwardly coded in any programming language that implements Join features, such as Jocaml [6] or Polyphonic C$^\sharp$ [1]. Our prototype implementation exploits both languages and allows agreements to be orchestrated among Linux components running Jocaml and Perl code and .Net components written in Polyphonic C$^\sharp$. As different parties communicates via TCP sockets, components of both types can participate to the same negotiation.

Components running Jocaml and Perl code are structured on three layers. The bottom layer hosts the distributed transaction manager, which is written in Jocaml. The other two layers (GUI and coordinators) are written in Perl, because of its simplicity for developing prototypes. Components written in Polyphonic C$^\sharp$ follow the object oriented paradigm: the instances of the class *d2pc* are responsible for performing the commit protocol.

In both cases, programs at the application-level are just responsible for keeping track of the involved parties and to initiate the agreement protocol. The execution of the commit protocol is transparent to the application-level (and hence to team members) and it is handled either by the two lower layers in the Jocaml and Perl implementation or by the class *d2pc* in Polyphonic C$^\sharp$. This abstracts away the application-level from the orchestration of the agreement, making the negotiation mechanism reusable for developing new applications.

Structure of the paper.

An original case study describing the assignment of activities to rescue teams is given in § 2. The mechanism for orchestrating agreements is presented in § 3, while the D2PC is summarized in § 3.1. The architecture and functionalities of our prototype implementations are detailed in § 4.

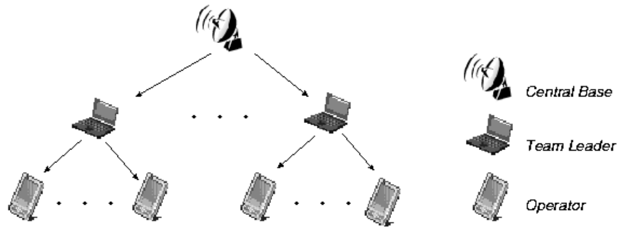


Figure 1. Logical structure of a rescue unit.

2 Scenario

This section presents a typical scenario requiring the orchestration of distributed agreements between several parties.

The scenario is within the more general context of [9] and considers rescue teams structured in a hierarchical way (as shown in Figure 1), where different nodes correspond to different computation and communication capabilities. Note that the tree in Figure 1 is not the communication graph. We shall abstract away from routing mechanisms and we assume that any team member can send messages to any other reachable member. The main goal of the application is to provide a set of functionalities to support the coordination of a rescue unit during ground operations. A rescue unit is divided into several rescue teams and is coordinated from a *Base* able to communicate via satellite or cellular telephony with a wired network. Additionally, the *Base* can communicate with the different rescue teams operating on the area (i.e., by using 802.11 devices). Any rescue team has a *team leader* who coordinates the team, consisting, e.g., of five operators. Team leaders can also act as operators if needed, but they have different computing power: leaders have laptops and operators are provided with PDAs. Moreover, all operators are equipped with a device for a georeference system that provides the *Base* with real-time information about their positions.

The assignment of tasks to people is organized in a top-down way. That is, the *Base* assigns general activities to the different teams by sending a message to the team leader. The leader will in turn split and distribute the task to team operators. Clearly, there can be different situations in which the distribution of activities may require an agreement between all involved members. Note that agreements cannot be established unilaterally and that a commit require the consensus of all participants.

The scenario described below considers a rescue unit consisting of four teams that cover different contiguous zones of an area where an avalanche occurred (as shown in Figure 2). This scenario specifies how the *Base* tries to assign an activity to the team T_1.

2.1 Scenario: Assignment of an Activity

Normal Flow:

Figure 2. A rescue unit distributed over a disaster area.

(i) The scenario starts when the *Base* sends a message to the leader l_1 of the team T_1 signaling the need of looking for an escape of gas in an area situated between the zones covered by teams T_1 and T_2.

(ii) After receiving the request, the leader l_1 decides that two operators will be needed to cover the whole area.

(iii) Consequently, the leader l_1 selects from T_1 the three operators that are closer to the compromised area hoping that at least two of them will be able to carry on the tasks, and sends them a message requiring their availability for performing a new task. After that, l_1 waits for operator's answers.

(iv) Any operator who receives the request will answer the message either by offering her/his availability or by refusing the task. Operators commit their participation to the negotiation when refusing a request, because they are not interested in the result of the agreement. (Note that refusal is not an abort).

(v) When l_1 receives the answers from the three operators, one of the following situations takes place:
   (a) All operators have answered in the affirmative. In this case l_1 chooses two of them and sends them detailed instructions for carrying out the activity. Moreover, l_1 communicates the decision to the excluded operator. Additionally, l_1 confirms the *Base* about the successful assignment of the activity and commits the negotiation.
   (b) Two operators have offered their help and the other refused the request. In this case the choice is the obvious one, and the leader sends messages only to the two chosen operators and to the *Base*, and he/she commits.
   (c) Less than two operators are available for the required task. In this case there are three alternatives:
   - l_1 refuses the activity by aborting the negotiation. In this case the *Base* will try to assign the activity to another team, for instance T_2.
   - l_1 asks the remaining operators of T_1 about their availability. The

scenario follows analogously from point 4.

- l_1 requires help from other teams (the scenario follows as described below in § 2.2).

(vi) If l_1 has managed to assign the task, then the chosen operators receive the specific instructions to perform the activity. After that, they will commit the agreement.

(vii) Also the *Base* receives the notification of the successful assignment of the activity to T_1 and commits the agreement.

(viii) When all participants have committed, all of them are notified about the successful completion of the agreement.

Exceptions: Any participant is able to withdraw its decision at any moment before it explicitly commits. In this case the scenario ends by making all participants aware about such decision. Typical cases are the following:

- The *Base* has been informed that the gas provider has safely stopped the provision on the area, and therefore the activity is no longer useful.
- The team leader l_1 receives a request to perform an activity with higher priority, for instance to move people out of the area.
- The operator realizes that is unable to reach the area.

As described before, during the assignment of an activity, a particular team may need some extra operators in order to carry out the task. Teams may also need help while they are performing an already assigned task, i.e. if an operator is unable to fulfill an activity that becomes harder or more complex. In such case, the operator will ask support to its own team by sending a message to the leader, who will manage to assign the new task to other members of the team (similarly to the task assignment described above as Normal Flow). It could be the case that the team is unable to provide the required support, doing necessary the participation of operators from other teams. The following scenario describes such situation.

2.2 Scenario: A team requires support from other teams

Normal Flow:

(i) The team leader l_1 asks the *Base* to find additional operators from other teams, for instance n operators.

(ii) The *Base* selects the k closest teams and forwards the request.

(iii) When a leader receives a request, it follows a protocol similar to that described in § 2.1 to inquire operators availability.

(iv) After receiving answers from operators, the leader informs the *Base* with the number of available operators.

(v) When the *Base* receives enough answers to satisfy the original request from l_1, it notifies all selected teams and l_1. The *Base* implicitly commits the agreement at this moment.

(vi) After receiving the confirmation, l_1 decides to commit the agreement.

(vii) Chosen leaders forward the received notification to their operators and commit the agreement.

(viii) Chosen operators receive the confirmation and then they commit.

(ix) All involved parties are notified when all involved participants have committed.

Exceptions: Analogously to the scenario presented in § 2.1, any participant can withdraw its decision and abort. In such cases, the scenario ends by making all participants aware about the abort.

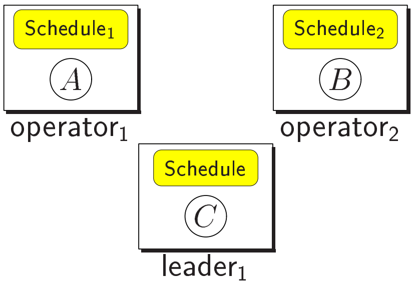
3 Coordination pattern

Agreements in our case study depend mainly on the particular dynamic interactions among the different members of a rescue unit: operators and leaders are getting involved in a negotiation when exchanging messages with other parties involved in that agreement. Hence the global structure of negotiations can be neither determined a priori nor statically fixed.

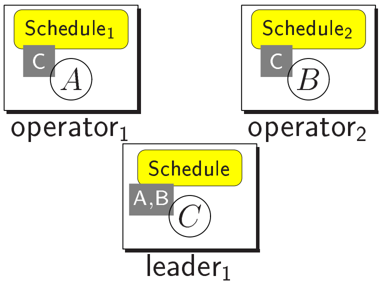
The most general scenario consists of distributed processes that can start local activities to be executed in the context of a larger negotiation. When a participant starts an activity to be part of an agreement, it creates a local manager to handle such negotiation. Local managers follow the distributed commit protocol of [3] described below (see § 3.1). Figure 3(a) shows a partial view of the state of several components in a rescue team after they have initiated their transactional processes. In particular, the participant leader_1 has an active process $\mathsf{Schedule}$ for handling the assignment of a particular task. Since $\mathsf{Schedule}$ runs as part of an agreement, it is managed by the local coordinator C. Similarly, any participant $\mathsf{operator}_i$ has an active process $\mathsf{Schedule}_i$ managed by the corresponding coordinator (A or B).

Now suppose that the activity $\mathsf{Schedule}$ sends a message to the process $\mathsf{Schedule}_1$ for assigning a particular activity to $\mathsf{operator}_1$. This interaction joins both activities $\mathsf{Schedule}$ and $\mathsf{Schedule}_1$ into the same negotiation. In our approach, this is achieved by making both participants aware about the identities of the corresponding coordinators. Similarly, if leader_1 also requires the support from $\mathsf{operator}_2$ to perform that activity, and then leader_1 contacts $\mathsf{operator}_2$, then the states of involved parties are updated as in Figure 3(b).

Consider that at this time all participants leader_1, $\mathsf{operator}_1$ and $\mathsf{operator}_2$ have all the information needed to decide independently either to commit or to abort. In this case, every participant locally activates the commit protocol described below and waits for the outcome decision.



(a) Initial situation.



(b) After the interactions.

Figure 3. Interaction between task in an agreement.

3.1 The Distributed Two Phase Commit Protocol (D2PC)

This section provides an informal description of the D2PC proposed in [3]. Originally, it was proposed to implement *zero-safe nets*, a transactional extension of Petri nets. The D2PC is a variant of the *decentralized 2*PC protocol [2]. Roughly, it implements a distributed agreement protocol among a set of participants (or their *managers*) that have a partial knowledge about the whole set of parties. The algorithm assumes a reliable asynchronous communication between participants. Moreover, participants can abort, but do not crash. The D2PC has been proved to be correct in such setting, assuring that all participants will asynchronously take the same decision (details can be found in [3]). Although in a MANET nodes can disconnect and communication is not highly reliable, in this work we do not deal with failures because we are aimed at studying how a protocol like the D2PC can be used to coordinate negotiations in scenarios like that described in § 2. Note that when communication reliability cannot be guaranteed by the MANET middleware (dynamic routing and retransmission mechanisms) the correctness proof of the D2PC is no longer valid. To deal with the more general case, we plan to develop and use a suitable distributed version of the *three phase commit protocol* (non-blocking and with less guarantees), but this is left for future work.

All participants in the D2PC act as transaction managers, all of them having the same behavior and communicating in an asynchronous way. Any manager maintains a list of all known parties (for that transaction), called the *synchronization set* ($\mathcal{S}$) and a list of committing parties ($\mathcal{C}$). At the beginning of the transaction both lists are empty. During the transaction, the synchronization set is updated to include parties from which a message has been received and also parties to which a message has been sent. Therefore, when the D2PC is activated to conclude the transaction, the synchronization set contains just those parties with whom a direct interaction occurred. Both lists $\mathcal{S}$ and $\mathcal{C}$ are updated during the execution of the protocol, until either there is an abort or the two lists become equal (meaning that all other participants to the transaction are known, they have voted for commit, and the commit vote has been sent to all of them). More precisely, any participant

Initial State of the j-th participant P_j.

- $\mathcal{S}_j$: set of all known parties (those with whom P_j cooperated directly).
- $\mathcal{C}_j = \emptyset$
- $state_j \in \{committing, failed\}$

Algorithm.

- **Committing.** While in state *committing* perform the following steps

(i) If $\mathcal{S}_j = \mathcal{C}_j$ then finish with "**commit**".
(ii) Otherwise, send the own synchronization set $\mathcal{S}_j$ to every known party in $\mathcal{S}_j$ to which the message has not been already sent (message `LOCK`).
(iii) for any received message `LOCK`($\mathcal{S}_i$) from the participant P_i update the state in the following way:
 · $\mathcal{S}_j = \mathcal{S}_j \cup \mathcal{S}_i$
 · $\mathcal{C}_j = \mathcal{C}_j \cup \{P_i\}$
(iv) if a message `ABORT` is received, send all `LOCK` messages and then pass to the state $failed$.
(v) goto 1.

- **Failed.** When the state $failed$ is reached, finish with "**abort**".

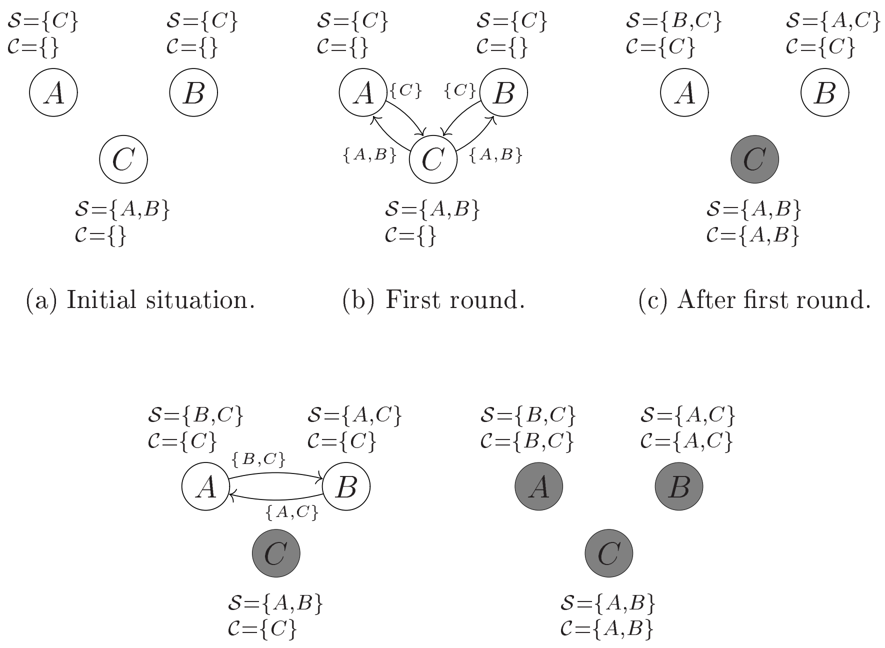
  While in state $failed$ answer with `ABORT` to any received message of type `LOCK`.

Figure 4. D2PC algorithm.

performs the algorithm described in Figure 4. We refer the interested reader to [3] for the formal definition of the protocol in Join. (Perhaps the meaning of the notation `LOCK` for those messages including synchronization sets is not obvious to the reader: it means that the parties in the synchronization sets are "locked" until an agreement / abort is established.)

In Figure 5 we illustrate a run of the D2PC with the three coordinators A, B and C from Figure 3(b), any of them willing to commit. The initial configuration (Figure 5(a)) shows the partial view that any participant has about the other parties in the agreement (see the local synchronization sets $\mathcal{S}$): A and B know only that C is part of the agreement processes, while C knows both A and B. Moreover, every participant initializes the set of commit confirmations $\mathcal{C}$ with the empty set.

When the protocol starts (Figure 5(b)) every participant sends its *ready to commit* vote together with its synchronization set $\mathcal{S}$ to any known participant. After this round (Figure 5(c)), all participants update their states with the information contained in the received messages. Note that C has received votes from both A and B without information about other participants. In this case both sets $\mathcal{S}$ and $\mathcal{C}$ of C coincide and thus C knows that all parties in the negotiation are willing to commit. At this time C can commit, because no party has decided to abort. Differently, A and B have received the commit vote from C containing participants not known previously, thus they update their state and must continue the execution of the protocol. In the next step,



(a) Initial situation. (b) First round. (c) After first round.

(d) Second round. (e) Final situation.

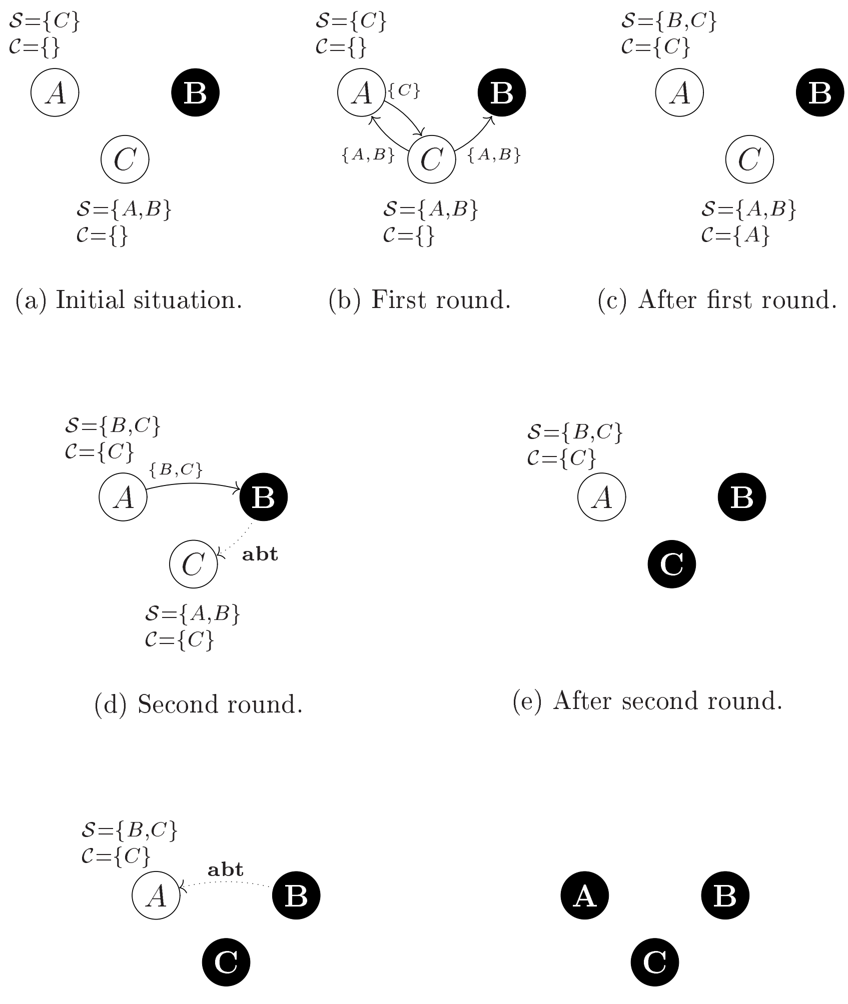
Figure 5. Example of commit.

A and B send their decisions to the recently known participants (Figure 5(d)). After that, they update their state and commit (Figure 5(e)).

Consider a different scenario in which A and C are willing to commit but B decides to abort. The initial situation is shown in Figure 6(a). We do not show the synchronization set of aborted components because it is useless. When the protocol starts, every participant in committing state (i.e., A and C) sends its vote to the known parties. Similarly to the previous case, committing participants update their states (Figure 6(c)). Note that C cannot commit because it has not received the confirmation from B. Neither A can commit because it has received the identity B, discovering a new participant to contact. In the next round (Figure 6(d)), A sends its vote to B. Instead, B answers the message received in the previous round from C with **abt**, signaling the abort of the negotiation. After the second round (Figure 6(e)) C aborts because of the message **abt** received from B, while A is still waiting the corresponding vote from B. Finally, in the third round (Figure 6(f)), B answers to the commit vote from A with **abt**. After this round (Figure 6(g)) all participants have aborted.

4 Implementation

We have developed a prototype application that implements a minimal set of functionalities in the context of scenarios described in the Introduction and



(a) Initial situation. (b) First round. (c) After first round.

(d) Second round. (e) After second round.

(f) Third round. (g) After third round.

Figure 6. Example of abort.

§ 2. It allows users to exchange textual messages and to reach an agreement among the parties that have interacted. In our prototype, parties can be of two different types: (i) Linux components running Jocaml and Perl code; and (ii) .Net components written in Polyphonic C$^\sharp$. Since parties communicates via TCP sockets, components of both types can participate to the same negotiation.

In this section we describe the architecture and the principles that have inspired the design of our implementation. In particular, the functionalities of a component from the user point of view are detailed in § 4.1, while the communication among parties is summarized in § 4.2. Then, § 4.3 and § 4.4 presents the architecture of Jocaml+Perl and Polyphonic C$^\sharp$ components, respectively. Finally, § 4.5 discusses the main differences among the various coding of the D2PC in Join, Jocaml and Polyphonic C$^\sharp$ together with some performance aspects.

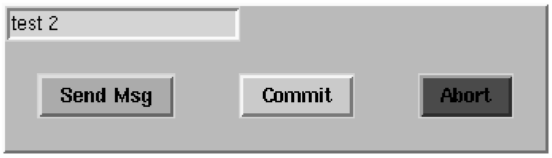


Figure 7. User actions.

4.1 User view

Our application allows users to exchange messages with textual content trying to establish some agreement with other reachable users (chosen from a set of parties fixed a priori and loadable from a configuration file). At any moment users can decide either to commit or to abort. Figure 7 shows the fragment of the graphical interface containing the core widgets: a text box for entering a message, a button for sending the typed message, one button for voting commit and another button for voting abort.

After having sent and received messages to / from other users as part of an agreement, a user will vote commit or abort. We assume that every participant will vote commit / abort after a finite amount of time. If the user votes abort, then the whole agreement is aborted. For this reason the graphical interface shows immediately the status *abort*. Moreover, all remaining users in that agreement will be aware of the abort after voting.

Instead, when a user votes commit all decisions from the other parties are waited for, and the status will be *commit* only when every other participant in the negotiation has voted commit. The way in which the decision is achieved is hidden to users, who can just press the commit button and then wait for the outcome to be displayed.

Additionally, we assume that the structure of a rescue unit is statically fixed and known a priori. For this reason we provide any user with a configuration file that describes all members in a rescue unit. In particular, any user is identified with a unique ID, which is provided as command line argument when the application is launched. In addition, the configuration file associates IDs with IP addresses. Parties know how to reach other nodes by reading the configuration file. Moreover, the ports on which parties communicate depend exclusively on the node ID. This assures that different applications running on the same IP address do not conflict in the use of TCP ports (useful for experimentation). Consequently, communication at both application and coordinator level requires only the ID of the peer partner.

Note that a discovery mechanism could be needed in scenarios where the set of participants cannot be statically defined. These cases require specific protocols for the dynamic discovery of nodes that are out of the scope of this work. Such protocols should be implemented at the physical media access level in order to save as much as possible the wireless media. For instance, we could use hwping like tools available on both Bluetooth and 802.11 technologies (l2ping, etherping), which makes the implementation strongly dependent

(a) User1. (b) User2. (c) User3.

Figure 8. Reachability information.

from the wireless physical layer adopted: any L2 media requires its own implementation of the discovery mechanism.

As an additional functionality our prototype provides a small mechanism for monitoring the reachability of nodes, which is independent from the physical media because it relies on UDP packets. It continuously polls the list of IP addresses by sending a dummy UDP packet to the echo port. A host is considered unreachable when the connection is not possible. This tool does not require any special root privileges (as icmp does), and it reduces the amount of TCP messages (SYN, ACK, PUSH, etc.) potentially flooding the wireless media. We could get rid of the configuration file by implementing an iterated automatic election algorithm similar to the one used by the NTP protocol to elect the master or to solve the Designated Router election problem in OSPF (where an automatic numbering of participants is performed on the basis of their interface MAC address).

Example 4.1 As a running example, we consider a system formed by three nodes. As mentioned before, the different nodes are identified by a name (typically an IP, but we have also used DNS resolvable names in our test bed) and an ID, which are defined into a configuration file. In this case all participants are using the following configuration file:

```
        dotto : 1
131.114.2.205 : 2
131.114.3.110 : 3
```

As soon as the application starts, each user interface will show reachable nodes. For instance, the user with ID 1 (abbreviated as User1) will see the other two users, i.e. User2 and User3 (Figure 8(a)). Similarly, User2 sees reachability information about User1 and User3 (Figure 8(b)) and User3 has information about User1 and User2 (Figure 8(c)).

Now, suppose User3 sends the message "test1" to User1 and, at the same time, User1 sends "test2" to User2 and User3. In this case, the interface of User1 (Figure 9) will show in its list of *Contacted* nodes the addresses of both User2 and User3. Moreover, the message "`test1 from User3`" is displayed on the list *Received Mesg*. Similarly, the interfaces of both User2 (Figure 10) and User3 (Figure 11) will display the address of User1 in the list of *Contacted* nodes and the message "`test2 from User1`" in the list *Received Mesg*.

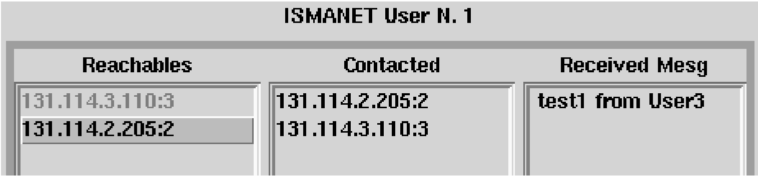


Figure 9. State of User1 after exchanging messages with User2 and User3.

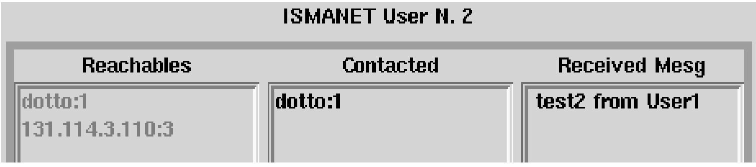


Figure 10. State of User2 after receiving a message from User1.

Figure 11. State of User3 after exchanging messages with User1.

Note that at this point User2 and User3 have never exchanged messages but, nevertheless, they are part of the same negotiation because both have interacted with User1. The information they know about each other concerns only reachability, i.e. they can communicate. Suppose that at this moment all users push the *Commit* button, which will activate the execution of the distributed commit protocol (D2PC) in every node. Since all participants have voted commit, the commit protocol will transparently close the agreement and the status bar of every GUI will eventually display the value *Commit* (in this case the execution of the D2PC will resemble Figure 5). Figure 12 shows the final state for User2 (the status is updated analogously in the GUIs of the remaining participants).

4.2 Communication between parties

The communication between parties (or nodes) occurs at two different levels: (i) the application level; and (ii) the coordinators. At the application level, parties exchange messages corresponding to the logic of agreements, as described in § 3. Instead, messages exchanged by coordinators correspond to the D2PC protocol. The two kinds of inter-party communication that can occur are summarized below, together with the corresponding message format.

Application level communication.
At the application level, two parties exchange messages when a user sends a message to another user. In this case, both the sender and the receiver update

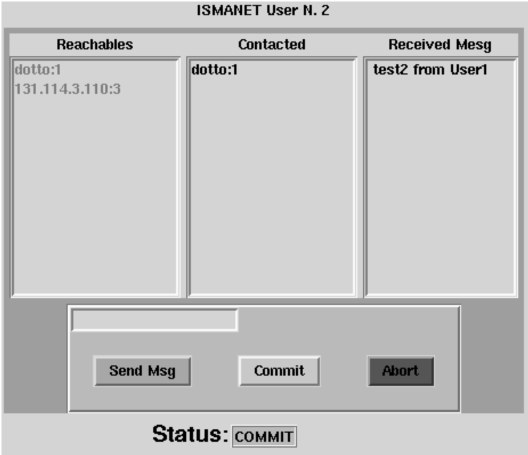


Figure 12. User2 after the termination of the D2PC.

their synchronization sets with the identity of the other participant, i.e., from this moment both participants are part of the same negotiation. Messages at the application level have the following form:

```
[free text] from User<ID>
```

A negotiation identifier should also be included. Without loss of generality, we assume here that each GUI is involved in just one negotiation. (In general, a local progressive numbering of negotiations would suffice.)

Communication between coordinators.
Coordinators exchange messages corresponding to the D2PC described in § 3.1 to vote *commit* or *abort*:

- `LOCK-l1;l2;...;ln-l1-a1-` to send a commit vote with the synchronization set `l1;l2;...;ln`. The logical names `l1` and `a1` denote the ports to be used by other participants to send D2PC messages to the local coordinator. (The corresponding TCP ports are easily derived from `l1` and `a1`, which are logical ids used for convenience.)
- `ABORT-` to notify that the sender has reached the abort.

4.3 Components coded in Jocaml and Perl

Parties have been implemented as the layered architecture shown in Figure 13. The functionalities of any layer are summarized below.

- The **Graphical Interface** handles GUI events to allow a user to send messages to other parties, and to commit or abort the current agreement.
- The **Coordinator** is responsible for the distributed execution of the commit protocol. It communicates with other coordinators and uses the underlying D2PC algorithm.
- The D2PC **algorithm** performs the algorithm described in § 3.1.

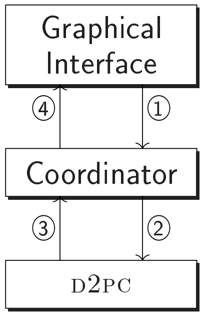


Figure 13. Layered architecture.

All information about the commit protocol is processed locally by the D2PC algorithm, but messages to/from other nodes are managed (and forwarded) by the coordinator layer. Although the communication between components could be wired into the D2PC algorithm, the two functionalities are kept apart to make the D2PC algorithm independent from the communication model used by parties. For instance, components could communicate through UDP sockets instead of TCP sockets only by changing the middle layer.

Top and middle layers have been implemented in Perl (for fast prototyping) while the bottom layer has been written in Jocaml.

Jocaml is an extension of the *Objective Caml* (Ocaml), a functional language with support of object oriented and imperative paradigms, that implements the primitives of Join. Jocaml provides three main abstractions: *process*, *channels*, *join-patterns*. Processes represent communication and synchronization tasks. The simplest process is an asynchronous message. Complex processes are obtained by combining expressions with the parallel composition of other processes. Channels are Jocaml abstractions corresponding to Join names. There are two different kind of channels: *synchronous* and *asynchronous*. The syntax for defining channels is the following

$$\textbf{let def} \quad name[!](args) = P(args)$$

This definition creates a channel (named *name*) and a receiver for it, which will execute the *guarded process* P every time it receives a message. The channel is asynchronous when its name is suffixed with the symbol !, otherwise it is synchronous. Synchronous names must return a value, i.e., P must explicitly define the return value. Finally, join-patterns are used to describe synchronization among different channels. A join-pattern definition creates several channels at the same time and states a synchronization between them: the corresponding guarded process may be executed only when messages on all channels are present. These features are exploited in the definition of D2PC managers. Figure 14 shows a partial view of the code corresponding to D2PC managers in Jocaml, in particular the patterns that handle the beginning of the protocol.

Layers communicates by exchanging messages asynchronously through TCP (or Unix domain) sockets, which provides modularity by allowing modules to

$$\begin{array}{l} \textbf{let def}\ \ \mathit{create\_thread}() = \\ \quad \ldots \\ \quad\ \textbf{or}\ \mathit{state}!\ h \mid \mathit{put}!(l,a,c) = \mathit{commit0}\ (\mathit{remove\ lock}\ l, l, [\mathit{lock}], c, \mathit{union}\ a\ h) \\ \quad\ \textbf{or}\ \mathit{state}!\ h \mid \mathit{abt}!() = \mathit{release}\ h \mid \mathit{failed}!() \\ \textbf{in reply}\ \mathit{lock}, \mathit{put}, \mathit{state}, \mathit{abt}\ \textbf{;;} \end{array}$$

Figure 14. Partial view of D2PC managers in Jocaml.

be implemented in different languages (e.g., Java instead of Perl).

The communication protocols between the different layers are summarized below (numbers refers to Figure 13).

(1) *Application → Coordinator.* The application layer sends a message to a coordinator in order to start the commit protocol, in particular it can send one of the following two messages, depending on the button pressed by the user:
   - `ABORT-` to start the commit protocol voting "*abort*".
   - `PUT-l1;l2;...ln-` to start the commit protocol voting "*commit*". The message includes the list of contacted parties `l1;l2;...;ln`, which is forwarded to the D2PC layer. The list will be used to set up the synchronization set before the start of the commit protocol. The name `PUT` for this kind of messages originated from the centralized version of the D2PC presented in [3] (based on the non reflective fragment of Join), where the message was meant to "put back" suitable tokens in the repository associated with that negotiation.

(2) *Coordinator →* D2PC. The coordinator forwards messages to the D2PC layer when it receives the vote from the user (one of the two messages described above) or when it receives votes coming from other parties as part of the D2PC protocol (inter-party messages between coordinators). More precisely, the coordinator can send the following messages to the D2PC layer in order to start the commit protocol or to update the status of algorithm:
   - `ABORT-` to start the commit protocol voting "*abort*" (corresponds to the abort message generated by the application layer) or to notify the reception of an abort message from a party.
   - `PUT-l1;l2;...;ln-` to start the commit protocol voting "*commit*". The synchronization set contains the coordinators `l1;l2;...;ln`. This message corresponds to the `PUT` generated by the application.
   - `LOCK-l1;l2;...;ln-l1-a1-` to notify a commit vote from `l1`, with the synchronization set `l1;l2;...;ln`. The ports `l1` and `a1` refers to the ports *lock* and *abort* of the sender.

(3) D2PC *→ Coordinator.* The D2PC algorithm generates the following messages to notify the coordinator about the actions it must take (see Figure 4):
   - `FWLOCK-l1-l1;l2;...;ln-` to ask the coordinator to forward the com-

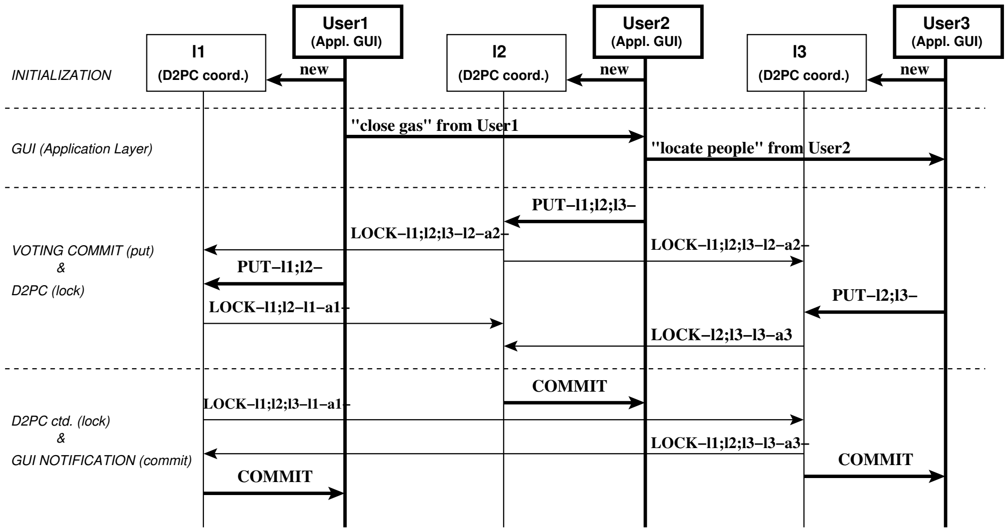


Figure 15. Sample timing diagram of an agreement.

mit vote to the coordinator `l1` with the synchronization set `l1;l2;...;ln`.

- `FWCOMMIT-COMMIT-` to ask the coordinator to inform the user that an agreement has been reached.
- `FWABT-ABORT-` to notify the coordinator that current negotiation has been aborted.
- `FWABT-a1-` to ask the coordinator to forward the abort message to the port `a1` corresponding to the port *abort* of a coordinator in the negotiation.

(4) *Coordinator* → *Application.* The coordinator informs the application about the success or abortion of the negotiation:

- `ABORT-` to inform that the running negotiation has been aborted.
- `COMMIT-` to inform that the running negotiation has been committed.

When one of the two messages above is received by the application, then the content of the status box in the user interface is updated correspondingly.

The sequence diagram in Figure 15 shows a sample interaction between the different layers and among participants. The coordinator layer is omitted for readability. In particular, when the application layer of a participant decides to start a new agreement, it locally creates a fresh D2PC manager for that agreement (INITIALIZATION phase). The GUI phase corresponds to the logic of the application. In the example, User1 sends a textual message to User2, who sends a message to User3. (Note that textual messages have an extra parameter for the identifier of the local D2PC manager, not reported in the diagram). In this way applications acquire the knowledge of cooperating managers (to whom messages are sent, or from whom messages are received). Eventually, each application layer will decide whether to commit or abort, starting the D2PC protocol. In this diagram we show the case in which all applications decide to commit: first User1 press the COMMIT button, then

```
public class d2pc{
    ...
    //declaration of asynchronous methods
    public async abt();
    public async put (lHost l, port a, port c);
    private async state (port h);
    private async commit0(lHost l,lHost l1,lHost l2,port c,lPort a);
    ...
    //a sample chord definition
    when state(port h) & put (lHost l, port a, port c){
       port localHost=l.element(0);
       lHost l1 = l.Clone();
       l1.remove(localHost);
       lHost l2 = new lHost(localHost);
       commit0(l1, l, l2, c, union(a,h));
    }
   ...
}
```

Figure 16. The class *d2pc*.

User2 and finally User3 do the same (see the order of PUT messages). Note that each application starts locally the protocol sending the PUT message to the manager. The parameters of PUT messages correspond to the list of contacted parties during the GUI phase. The LOCK messages are sent by the managers according to the D2PC algorithm. When the execution of the D2PC concludes, every manager will inform its application layer with the final decision (COMMIT, in this case).

4.4 .Net Components

Parties have been also implemented in the object oriented language Polyphonic C# [1]. Polyphonic C# extends C# with asynchronous methods (declared with the keyword **async** and synchronization patterns, called *chords* (defined by keyword **when**. A call to an asynchronous method is guaranteed to complete almost immediately, i.e., the caller never blocks. A chord is defined by a header (i.e., a set of method declarations) and a body. The body is only executed once all the methods in the header have been called.

Consider the class *d2pc* in Figure 16, whose instances are responsible for executing the commit protocol. The public asynchronous methods `put`, `lock` and `abort` respectively initiates the protocol, receives a ready to commit vote from a partner, and receives an abort. The private asynchronous methods `state` and `commit0` represent internal states of managers. The following chord

when `state(port h) & put(lHost l,port a,port c)...`

handles the activation of the commit protocol. In particular, its body is executed when both `state` (coding the initial state of the manager) and `put` (i.e., the commit vote from the application) are called.

The classes of Polyphonic C# components are organized as in Figure 17. The utility classes *Sender* and *Receiver* provide methods for sending messages to and receiving messages from other parties. The class *User Interface* handles

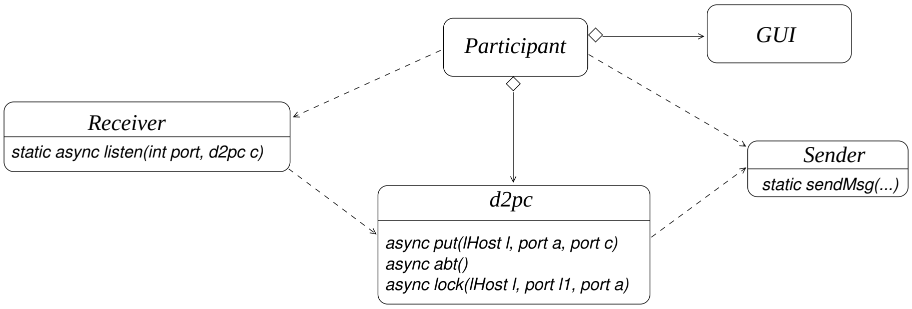


Figure 17. Structure of Polyphonic C# components.

the interactions with the user and the instances of *d2pc* execute the commit protocol. Note that the communication between classes inside a component is achieved by method invocation instead of socket communication.

4.5 Discussion

The main differences between the implementations of the D2PC in Jocaml and in Polyphonic C♯ are:

- *Nondeterministic abort.* The original Join coding allows a manager to autonomously initiate at any time the commit protocol voting abort while it has not received the `PUT` message that initiates the commit protocol with vote commit (nondeterministic simulation of abort decision). This rule, which guarantees the termination of any instance of the protocol, has the disadvantage that can be fired as soon as the manager is created, forbidding in this way the possibility to wait for a commit. Instead, in both the Jocaml and Polyphonic C♯ implementation, the manager starts the commit protocol voting for abort only when it receives the abort vote (e.g. from the associated user). This implementation choice does not compromise the correctness and completeness of the D2PC as far as every participant in the agreement vote after a finite amount of time. As we are assuming that all users will eventually vote, this modification does not affect the properties of the protocol.
- *Non-linear pattern matching.* Neither Jocaml nor Polyphonic C♯ provide mechanisms of non-linear pattern matching, although an extension of the Join calculus with linear pattern matching has been proposed in [10]. In the Join formulation of the D2PC, non-linear pattern matching is used for convenience on port `commit`, which represents an internal state of a manager. There are two cases, one in which there are managers to be notified, and the other when all known managers have been already notified. The D2PC allows both sending of notification and vote receptions from other managers to be interleaved freely. In our implementation we impose all notifications to be sent before accepting a vote from a manager. Clearly, this is a particular interleaving of the original specification, and therefore it satisfies all the properties of the protocol. In our encoding this is achieved by using an

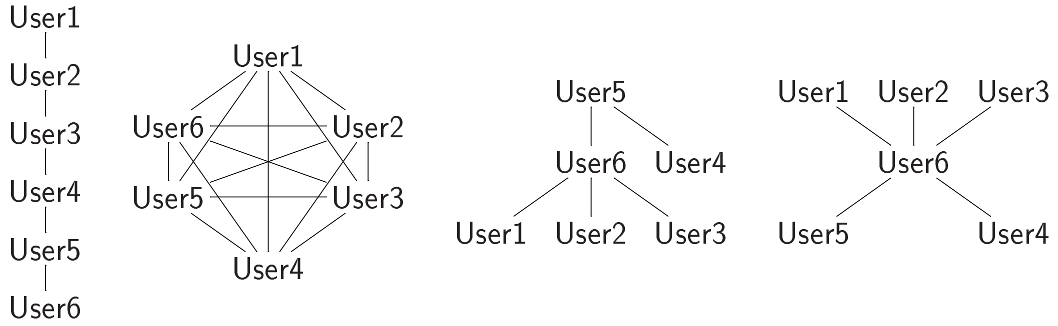


Figure 18. Experimentation patterns.

auxiliary port so to avoid non-linear pattern matching.

- *Operations over sets.* We have implemented in Jocaml the functions union, remove, equivalence and difference over (lists representing) sets. In Polyphonic C♯ we have implemented the class *lHost* that provides the operations corresponding to a set of host addresses.

To give a taste of the overhead of the D2PC in terms of total amount of bytes exchanged on the network and the maximum time delay needed to inform all users about the commit, we performed some tests involving six users over a simple MANET configuration. Users were numbered from 1 to 6 and the MANET was composed of two nodes (laptops) linked by a Bluetooth connection. Each node was running three different users (even numbered users and odd numbered ones running on different nodes). Users exchanged messages according to four different patterns (see Figure 18): (i) a *linear* pattern, where user number i sends textual messages to user $i+1$; (ii) a *clique* (complete graph) pattern, each user sending and receiving from all the others; (iii) an *unbalanced tree*-like pattern; (iv) a *star*-shaped pattern, with one user exchanging messages with all the others. The different patterns influence the initial knoweledge of each participant when the protocol starts.

In all our experimentations, the time measuring the performance of the commit protocol was calculated as the time interval between the last push of commit button (always from user number 6) and the first / latest flashing of the COMMIT flag on user interfaces. Note that when the button is pressed by the last user, the other managers have already started the protocol. The number of bytes sent during the execution of the protocol is almost constant across the configurations, being minimum in the linear pattern and maximum for the clique (this is because in the clique pattern the initial synchronization sets are larger). Differently, the execution time is minimum for the clique pattern, and it varies up-to 5 times for the linear case and up-to 10 times for the remaining cases. Of course, these data strongly depend on the number of participants, on the way in which they exchanged information and on the order in which users voted for commit. Scalability issues can be hardly inferred from this simple experimentation.

5 Conclusions

We have described a prototype implementation of distributed agreements in multi-parties negotiations that takes advantage of the D2PC protocol introduced in [3]. Parties have been implemented both in Polyphonic C$^{\#}$ running on .Net and in Jocaml + Perl running on Linux. Since the communication among parties takes place by exchanging textual messages on TCP sockets, components running in different platforms can interoperate.

Nevertheless, some limitations should be overcome in order to make the described architecture fully satisfactory for scenarios like the one in § 2. In particular, the D2PC should be extended to handle failures, for instance by using a suitable version of the *three phase commit protocol*. Moreover, taking into account the hierarchical organization of rescue units and the way in which decisions are taken, it would be interesting to analyze the combination of the D2PC with some traditional commits protocols that optimize the number of exchanged messages. Additionally, the inclusion of some mechanisms for the dynamic discovering of participants instead of the configuration files used in the presented implementation would be desirable.

As an additional contribution, the proposed architecture seems suitable to implement (in an ad hoc manner) applications written in cJoin [5]. The cJoin calculus is an extension of the Join calculus with nested, compensatable negotiations, where processes in different transactions can interact by joining their original negotiations into a larger one. In particular, the subcalculus of *flat* negotiations has been encoded into Join by applying the D2PC [4]. Such encoding provides the bases for coding cJoin applications over the presented architecture.

References

[1] Benton, N., L. Cardelli and C. Fournet, *Modern concurrency abstractions for C$^\sharp$*, in: B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming, 16th European Conference*, Lect. Notes in Comput. Sci. **2374** (2002), pp. 415–440.

[2] Bernstein, P., V. Hadzilacos and N. Goodman, "Concurrency, Control and Recovery in Database Systems," Addison-Wesley Longman, 1987.

[3] Bruni, R., C. Laneve and U. Montanari, *Orchestrating transactions in join calculus*, in: L. Brim, P. Jancar, M. Kretinsky and A. Kucera, editors, *Proceedings of CONCUR 2002, 13th International Conference on Concurrency Theory*, Lect. Notes in Comput. Sci. **2421** (2002), pp. 321–336.

[4] Bruni, R., H. Melgratti and U. Montanari, *Flat Committed Join in Join*, in: F. Honsell, M. Lenisa and M. Miculan, editors, *Proceedings of CoMeta 2003, Final Workshop of the CoMeta Project*, Elect. Notes in Th. Comput. Sci, 2004, pp. 39–54.

[5] Bruni, R., H. Melgratti and U. Montanari, *Nested commits for mobile calculi: extending Join*, in J-J. Lévy, E. W. Mayr and J. Mitchell: *Proceedings of the 3rd IFIP-TCS 2004*, 2004, pp. 569–582.

[6] Conchon, S. and F. Le Fessant, *Jocaml: Mobile agents for Objective-Caml*, in: *1st International Symposium on Agent Systems and Applications (ASA'99)/3rd International Symposium on Mobile Agents (MA'99)*, 1999, pp. 22–29.

[7] Fournet, C. and G. Gonthier, *The reflexive chemical abstract machine and the Join calculus*, in: *Proceedings of POPL'96, 23rd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages* (1996), pp. 372–385.

[8] Lynch, N., "Distributed Algorithms," Morgan Kaufmann Publishers, 1996.

[9] IS-MANET, *Un possibile scenario per la piattaforma* IS-MANET, Online documentation of the MIUR Project IS-MANET. Available at http://zeus.elet.polimi.it/is-manet/ (2003).

[10] Ma, Q. and L. Maranget, *Compiling Pattern Matching in Join-Patterns*, in: P. Gardner and N. Yoshida, editors, *Proceedings of CONCUR 2004, 15th International Conference on Concurrency Theory*, Lect. Notes in Comput. Sci. **3170** (2004), pp. 417–431.