Università di Pisa
Dipartimento di Informatica

Technical Report: TR-04-10

A prototype implementation of distributed agreements

Alberto Baragatti Roberto Bruni Hernán Melgratti
Ugo Montanari

22 April 2004

ADDRESS: via F. Buonarroti 2, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

A prototype implementation of distributed agreements ⋆

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

Dipartimento di Informatica, Università di Pisa, Italia.
{baragatt,bruni,melgratt,ugo}@di.unipi.it

Abstract. We present a prototype application for distributed agreements in multi-parties negotiations, where the number of participants is statically unknown, new participants can dynamically join ongoing negotiations and each participant knows only those parties with whom interacted. The application is based on asynchronous communication and it exploits the D2PC protocol for committing or aborting the negotiation. The software architecture consists of three modular layers, with the D2PC compiled in the bottom layer and completely transparent to final users. Our prototype is tailored to the running scenario of Ad-Hoc networks used by rescue teams for disaster recovery.

1 Introduction

Mobile Ad-Hoc networks (MANET) are networks where agent mobility coexists with dynamic infrastructures and net topology. MANET are typical of 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). Although several technologies are currently available neither a standard, nor rigorous theoretical foundations and models have yet emerged.

This paper describes a prototype implementation of a "proof of concept" application that requires distributed negotiations between a set of participants not known statically. The typical running scenario involves a rescue unit composed by several teams. Each team has a *leader* and some *operators*. Leaders and operators can interact by exchanging messages, so to orchestrate the participation to each specific task, also depending on the position of each operator in the disastered area. 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 perfomed. Leaders and operators have often a limited knowledge of the other participants involved in the task.

⋆ Research supported by the Italian Ministry of Education, University, and Research (MIUR) within the framework of the IS-MANET project (Software infrastructures for mobile Ad-Hoc networks in difficult environments).

Intial 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
  1. If $\mathcal{S}_j = \mathcal{C}_j$ then finish with "**commit**".
  2. Otherwise, send the own synchronization set $\mathcal{S}_j$ to every known party in $\mathcal{S}_j$ (message $LOCK$).
  3. for any received message $LOCK - \mathcal{S}_i$ from the participant P_i update the state in the following way:
     - $\mathcal{S}_j = \mathcal{S}_j \cup \mathcal{S}_i$
     - $\mathcal{C}_j = \mathcal{C} \cup \{P_i\}$
  4. if a message $ABORT$ is received, send all $LOCK$ messages and then pass to the state $failed$.
  5. 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 $LOCK$.

Fig. 1. D2PC algorithm.

The prototype implementation we present is based on a general purpose mechanism for orchestrating distributed agreements, whose foundations rely on the Join calculus [7], a reflexive process description language based on asynchronous communication, fresh name creation, and name mobility. The most abstract scenario consists of distributed processes that can start local activities to be executed in the context of a larger negotiation. Nevertheless, the structure of the negotiation in which they can be involved is not described statically, but determined at runt-time. In fact, the global structure of a negotiation is discovered dynamically accordingly to the interaction of the different participants. In other words, any activity running as part of a negotiation can communicate with other processes, possibly involved in another negotiation. When such communication occurs, all the involved parties become part of the same larger negotiation, even if they are not aware of the merging.

In order to implement such kind of agreements in a fully distributed way which is transparent to final users, we exploit the *distributed two phase commit* protocol (D2PC), informally summarized in Figure 1. The D2PC was originally proposed in [2] for implementing a transactional extension of Petri nets, called *zero-safe nets* [5], in Join. Consequently, the D2PC can be straightforwardly coded in any programming language that implements Join features, such as Jocaml [6] or Polyphonic C$^\sharp$ [1]. For these reasons, our prototype implementation is structured in three layers, where the bottom layer provides the middleware for distributed negotiations, hosting the distributed transaction manager, which is written in Jocaml. In order not to tight the development of applications to Jocaml, the code for the D2PC has been developed as an independent process that communicates with the other layers of the application through sockets. In our case, the other two layers (GUI and coordinators) are written in Perl, because of its simplicity for developing prototypes.

Structure of the paper. In § 2 we give an informal description of the D2PC, while in § 3 we describe the architecture of the application and its implementation. In § 4 we give a detailed description of the negotiations that can take place during the assignment of activities to rescue teams.

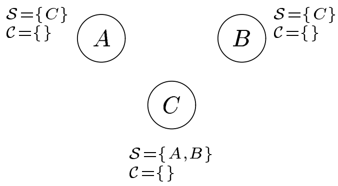
2 The Distributed Two Phase Commit (D2PC)

In this section we provide an informal description of the D2PC proposed in [2]. Roughly, the D2PC implements a distributed agreement protocol among a set of participants (or their *managers*) that have (at the begining) 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 (details can be found in [2]).

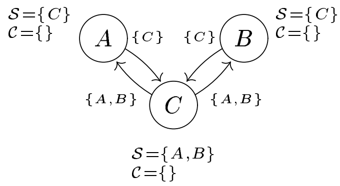
In the D2PC all participants act as transaction managers, all of them having the same behaviour and communicating in an asynchronous way. Any manager maintains a list of all (the ports to communicate with) known parties, called the *synchronization set* ($\mathcal{S}$) and a list of committing parties ($\mathcal{C}$). Both lists are updated during the execution of the protocol. Any participant performs the algorithm described in Figure 1. (We refer to [2] for the formal definition of the protocol in Join).

In Figure 2 we illustrate a run of the D2PC with three participants A, B and C, any of them willing to commit. The initial configuration (Figure 2(a)) shows the partial view that any participant has about the other parties in the agreement (i.e., the synchronization sets $\mathcal{S}$): A and B know only that C is part of the agreement processes, while C knows both A and B. Note that the knowledge relation is symmetric, which is required by the D2PC in order to assure consensus. Moreover, for all participants the set $\mathcal{C}$ of commit confirmations is initialized to the empty set.

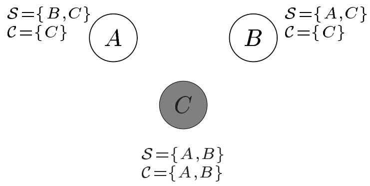
When the protocol starts (Figure 2(b)) every participant sends its ready to commit vote to any known participant together with its synchronization set $\mathcal{S}$. After this round (Figure 2(c)), all participants update their states with the information contained in the received messages. Note that C has received votes from both A and B without information about other participants. In this case both its sets $\mathcal{S}$ and $\mathcal{C}$ coincide and C knows that all parties in the negotiation are willing to commit. Therefore it commits. Differently, A and B have received the commit vote from C containing participants not known previously, thus they update their state and continue the execution of the protocol. In the next step, A and B send their decisions to the recently known participants (Figure 2(d)). After this step, they update their state and commit (Figure 2(e)).



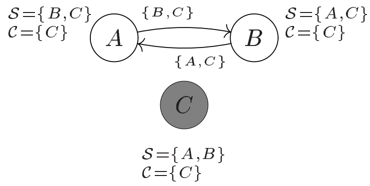
(a) Initial situation.



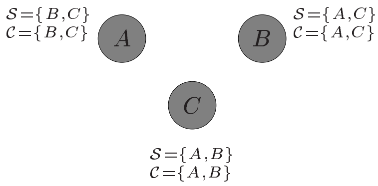
(b) First round.



(c) After first round.



(d) Second round.



(e) Final situation.

Fig. 2. Example of commit.

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 3(a). Note that the synchronization sets of B are never shown in Figure 3 because they are useless when the participant aborts. 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 3(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 3(d)), A sends its vote to B. Instead, B answers the message received in the previous round from C with **abt**, signaling the negotiation must be aborted. After the second round (Figure 3(e)) C aborts because of the message **abt** received from B, while A is still waiting

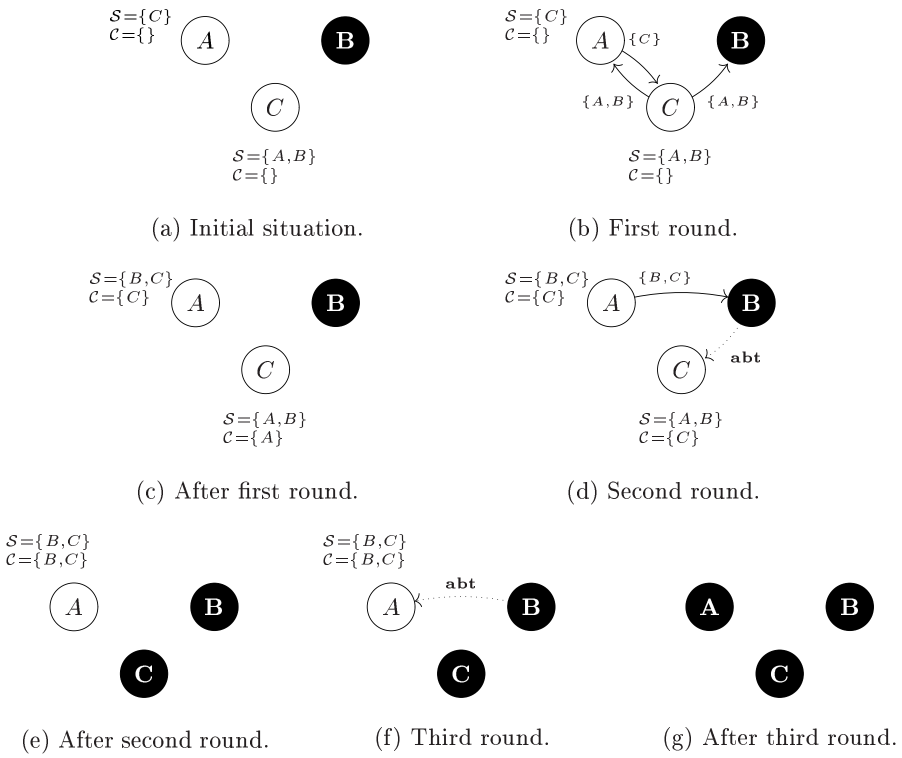


Fig. 3. Example of abort.

the corresponding vote from B. Finally, in the third round (Figure 3(f)), B answers to the commit vote from A with **abt**. After this round (Figure 3(g)) all participants have aborted.

3 Application

We developed a prototype application that implements a minimal set of functionalities for allowing users to communicate by exchanging messages and to reach an agreement in the context of scenarios like those described in the Introduction and § 4. In this section we describe the architecture and the principles that have inspired the design of our implementation.

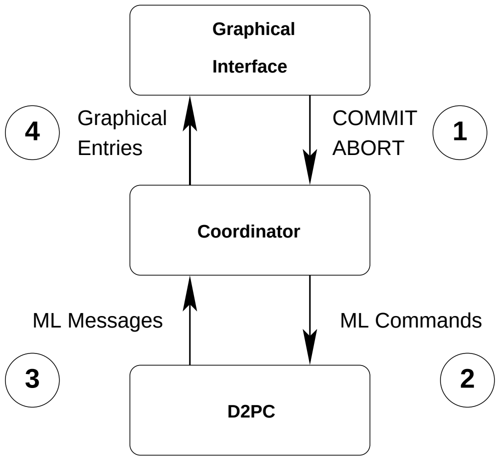


Fig. 4. Application layout.

3.1 Architecture

The application has been implemented as the layered architecture (i.e., a stack of modules) shown in Figure 4. The functionalities of any layer are summarised below.

- **Graphical Interface**, which handles GUI events, allowing a user to send messages to other parties, and to commit or abort the current agreement.
- **Coordinator**, responsible for the distributed execution of the commit protocol, by communicating with other coordinators and using the underlying D2PC algorithm.
- D2PC **algorithm**, which performs the algorithm described in § 2.

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.

Top and middle layers have been implemented in Perl while the bottom one has been written in Jocaml. The communication between layers is realized by exchanging messages asynchronously through TCP (or Unix) sockets. This leaves us the freedom of implementing each module in a different programming language, achieving modularity. At the same time, this allows the interaction with participants running on other platforms and developed in other languages, such as Polyphonic C# applications running on the .NET platform, thus yielding interoperability.

Due to the asynchronous communication between the different layers and users, the coordinator's functionalities have been split into several threads in order to manage all messages coming from other parties and adjacent modules.

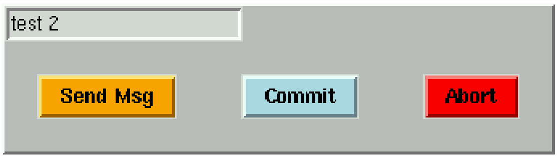


Fig. 5. User actions.

Then the Jocaml core for the D2PC computes a new network status and eventually sends commands back to the coordinator that in turn updates GUI status and forwards messages to coordinators modules running on different nodes.

3.2 User view

To improve usability of our application, the top layer hides all the implementation details to final users. In particular, the D2PC protocol is provided as a middleware service and its execution is transparent to the user.

The prototype abstracts away from the logic of the negotiation, but of course, algorithms or workflows based on message exchanges can be straightforwardly programmed by exploiting the two layers below. Negotiations are expected to finish within a period of time, therefore we assume without loss of generality that each participant will vote commit / abort after a finite amount of time.

Actually, users in our application are allowed to send messages with free textual content, trying to establish some agreement with other reachable users (chosen from a set fixed a priori and loadable from the configuration file). Moreover, at any moment, a user can decide to commit or abort. Figure 5 shows a detail of the graphical interface, which contains the core widgets: a text box for inputting a message, a button to send the message, a button for voting commit and one for voting abort.

When a user aborts, the local graphical interface shows immediately the status *abort*. Moreover, every other user in the agreement that has voted, will eventually reach the abort state (as granted by the D2PC).

When a user tries to commit, the decisions from all other parties must be waited for, because the status of every participant must be *commit* only when every participant in the negotiation has voted commit. The way in which the decision is achieved is not shown to the user, who has just to press the commit button and then wait for the outcome to be displayed.

3.3 Communication between parties

Each user is identified with its own unique ID, which is provided as command line argument when the application is first launched. For simplicity, the application knows how to reach other nodes by reading a static configuration file, which associates an IP address to any ID, e.g. *node-address:*ID. Moreover, the ports in which parties communicate depend exclusively on the node ID. This assures that

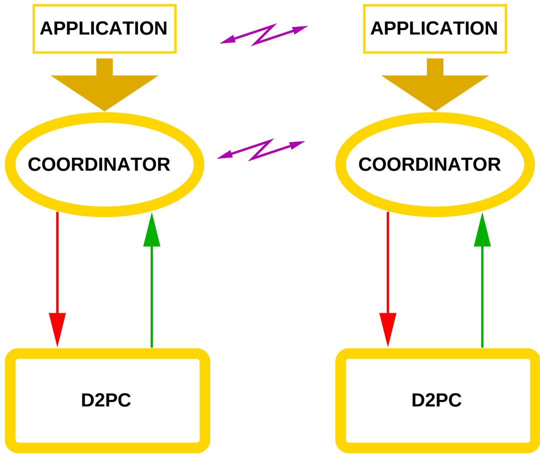


Fig. 6. Communications between parties.

different applications running on the same IP address do not conflict in the use of ports.

The application layer provides a simple discovery tool that dynamically checks peers availability. It has been implemented as a thread that periodically sends icmp messages (*echo request*) toward all peers in the configuration file. Probably a wireless media dependent tool (hwping, l2ping) could be desiderable as a less invasive discovery protocol for the wireless physical channel, but this would tie the application to a specific wireless technology. The actual design instead, is only bound to the IP protocol and proved to work on Bluetooth and 802.11 media in Ad-Hoc mode.

As shown in Figure 6, the communication between different parties (or nodes) occurs at two different levels: (i) the application layer; and (ii) the coordinators. Note that the D2PC algorithm layer is wrapped by the coordinator layer and thus it cannot directly exchange messages with other parties. In fact, bottom layers interact only through their corresponding coordinators, which are responsible for forwarding messages from/to other parties.

We recall that both the application and the coordinator layer requires only the ID of the peer party to communicate with, because they are associated to a particular address and range of ports. We assume the platform 16 bits threshold for port definition (max 65535 ports) does not affect the application behaviour.

The two kinds of inter-party communication that can occur are summarized below, together with the corresponding message format.

Application layer communication. The application layer exchanges messages with the correspondent layer of another node when a user decides to send a message to a different user. In this case, both the sender and the receiver update their synchronization sets with the identity of the other participants, i.e., from

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

Fig. 7. Reachability information

this moment both participants are part of the same negotiation. Messages must have the form:

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

Communication between coordinators. Coordinators exchange the following two messages, which are the messages involved in the D2PC described in § 2 for voting *abort* or *commit*:

- `ABORT-` to notify that the sender has reached the abort.
- `LOCK-l1;l2;...;ln-l1-a1-` to send a commit vote with the synchronization set `l1;l2;...;ln`. The ports `l1` and `a1` are respectively the ports to be used by other participants to send D2PC messages to the local coordinator.

Example. As a running example, let us consider three different nodes participating to the same negotiation. As mentioned before, the different nodes are identified by a name (i.e., an IP or a DNS resolvable name) and the IDs are statically defined into the configuration file of the application. 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 7(a)). Similary, User2 sees reachability information about User1 and User3 (Figure 7(b)) and User3 has information about User1 and User2 (Figure 7(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 8) will show in its list of *Contacted* nodes the addresses of both User2 (i.e., `131.114.2.205:2`) and User3 (i.e.,`131.114.3.110:3`). Moreover, the message "`test1 from User3`" is displayed on the list *Received Mesg.* Similarly, the

Fig. 8. State of User1 after exchanging messages with User2 and User3.

Fig. 9. State of User2 after receiving a message from User1.

interfaces of both User2 (Figure 9) and User3 (Figure 10) will display the address of User1 in the list of *Contacted* nodes and the message "`test2 from User1`" in the list *Received Mesg*.

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 only information they know about each other concerns reachability (they are *in range*). 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 D2PC closes the agreement, and the green *Commit* flag will be displayed in the status bar of each GUI. Figure 11 shows the final state of User1 (the status is updated anologously in the GUIs of the other participants).

3.4 Communication Protocols between layers

In this section we summarise the communication protocols between the different layers (the numbering refers to Figure 4).

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 synchronization set contains the coordinators `l1;l2;...;ln`.
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

Fig. 10. State of User3 after exchanging messages with User1.

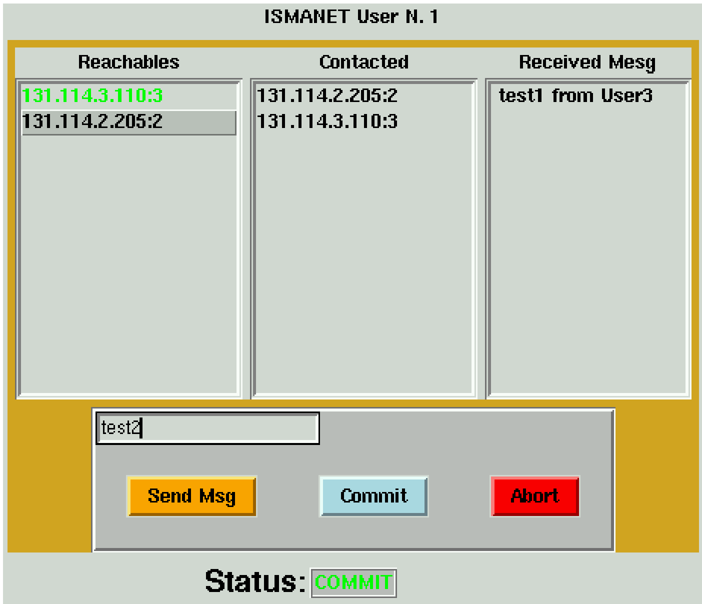


Fig. 11. User1 after the termination of the D2PC.

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 syncronization 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 1):
   - `FWLOCK-l1-l1;l2;...;ln-` to ask the coordinator to forward the commit 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 with 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.

4 Future Scenario

In this section we consider a scenario borrowed from [8] to identify possible applications of the agreement mechanism presented before. The scenario considers rescue teams structured in a hierarchical way (as shown in Figure 12), where different nodes correspond to different computation and communication capabilities. 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* capable of communicating 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, consisting, e.g., of five operators, has a *team leader* that coordinates the team. Any team leader can communicate with the central *Base* and with team operators, which are provided with PDAs. PDAs are able to communicate with the team leader and possibly with other 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 assigment 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.

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 13). This scenario specifies how the *Base* assigns an activity to the team T_1.

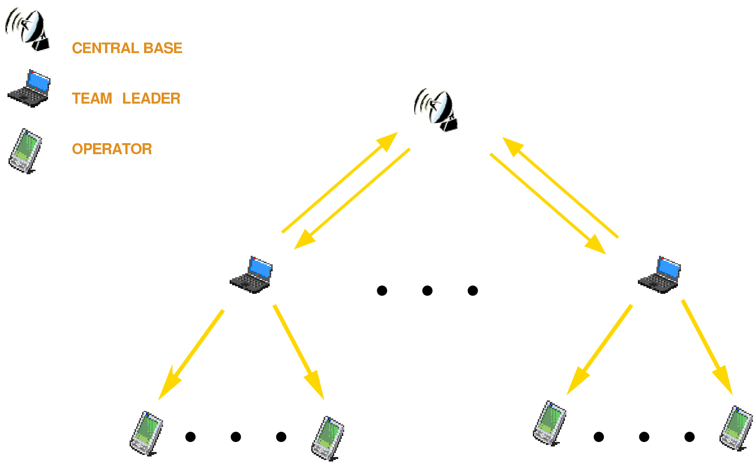


Fig. 12. Structure of a rescue unit.

4.1 Scenario: Assignment of an Activity

Normal Flow:

1. 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.
2. After receiving the request, the leader l_1 decides that two operators will be needed to cover the whole area.
3. Consequently, the leader l_1 selects from T_1 the three operators that are closer to the compromised area, and sends them a message requiring their availability for performing a new task. After that, l_1 waits for operator's answers.
4. Any operator that receives the request will answer the message either by offering her/his availability or by refusing the task. An operator implicitly commits the negotiation when refusing a request, because they are not interested in the result of the agreement.
5. 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 remaining 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 it commits.
   (c) Less than two operators are available for the required task. In this case there are three alternatives:

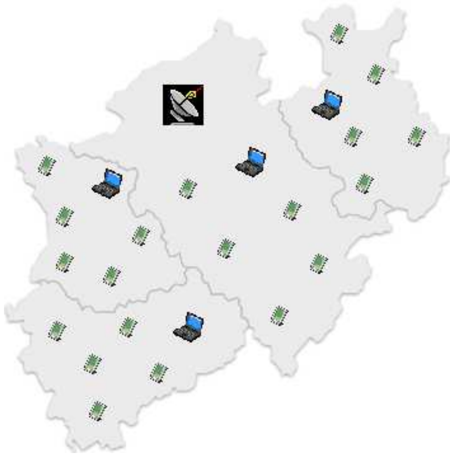


Fig. 13. A rescue unit distributed over a disastered area.

   i. 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.
   ii. l_1 asks the remaining operators of T_1 about their availability. The scenario follows analogously from point 4.
   iii. l_1 requires help from other teams (the scenario follows as described below in § 4.2).
6. If l_1 has managed to assign the task, then the chosen operators receive the specific instructions to perfom the activity. After that, they will commit the agreement.
7. Also the *Base* receives the notification of the successful assignment of the activity to T_1 and commits the agreement.
8. 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 aware all participants 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 realises 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 in § 4.1). 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.

4.2 Scenario: A team requires support from other teams

Normal Flow:

1. The team leader l_1 asks the *Base* to find additional operators from other teams, for instance n operators.
2. The *Base* selects the k closest teams and forwards the request.
3. When a leader receives a request, it follows a protocol similar to that described in § 4.1 to ask for operators availability.
4. After receiving answers from operators, the leader informs the *Base* with the number of available people.
5. 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.
6. After receiving the confirmation, l_1 decides to commit the agreement.
7. Chosen leaders forward the received notification to their operators and commit the agreement.
8. Chosen operators receive the confirmation and then decide to commit.
9. All involved parties are notified when all involved participants have committed.

Exceptions: Analogously to the scenario presented in § 4.1, any participant can withdraw its decision and abort. In such cases, the scenario ends by making aware all participants about the abort.

5 Conclusions

In this paper we have described a prototype implementation of distributed agreements in multi-parties negotiations by taking advantage of the D2PC protocol introduced in [2]. Any party is implemented as a three-layer system (as shown in Figure 4). The application layer is responsible for keeping track of the involved parties and eventually to initiate (locally) the agreement protocol. The two lower layers abstract away the application from the orchestration of the agreement. In this way, they can be reused to provide a negotiation mechanism for the development of new applications. Moreover, as the communication between different coordinators takes place by exchanging textual messages through TCP sockets, it would be possible to interoperate with applications running on different platforms, such as Polyphonic C$^{\#}$ applications running on .NET.

Nevertheless, some limitations should be overcome in order to make the described architecture fully satisfactory for scenarios like that in § 4. 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 static 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 [4]. 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 [3]. Such encoding provides the bases for coding cJoin applications over the presented architecture.

References

1. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for $C^\sharp$. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming, 16th European Conference*, volume 2374 of *Lect. Notes in Comput. Sci.*, pages 415–440, Malaga, Spain, June 10-14 2002. Springer Verlag.
2. R. Bruni, 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*, volume 2421 of *Lect. Notes in Comput. Sci.*, pages 321–336. Springer Verlag, 2002.
3. R. Bruni, 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*, 2003. To appear as Elect. Notes in Th. Comput. Sci.
4. R. Bruni, H. Melgratti, and U. Montanari. Nested commits for mobile calculi: extending Join, 2004. Submitted.
5. R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Inform. and Comput.*, 156(1-2):46–89, 2000.
6. S. Conchon 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.
7. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR'96, 7th International Conference on Concurrency Theory*, volume 1119 of *Lect. Notes in Comput. Sci.*, pages 406–421. Springer Verlag, 1996.
8. IS-MANET. Un possibile scenario per la piattaforma is-manet. On-line documentation of the MIUR Project IS-MANET. Available at `http://zeus.elet.polimi.it/is-manet/Documenti/mi20030924-dei.ppt`, September 2003.