Computing and Informatics, Vol. 25, 2006, 571–595

LACOLLA: A MIDDLEWARE TO SUPPORT SELF-SUFFICIENT COLLABORATIVE GROUPS

Joan Manuel Marquès

Universitat Oberta de Catalunya, Department of Computer Sciences
Av. Tibidabo, 39-43, 08035
Barcelona, Catalonia, Spain
e-mail: jmarquesp@uoc.edu

Leandro Navarro

Universitat Politècnica de Catalunya, Department of Computer Architecture
Jordi Girona, 1-3, D6-105, Campus Nord
Barcelona, Catalonia, Spain
e-mail: leandro@ac.upc.edu

Revised manuscript received 19 June 2006

Abstract. In a decentralised and distributed environment, collaboration requiring the sharing and building of applications is a complex task. For this reason, we propose LaCOLLA, a fully decentralised peer-to-peer middleware that aims to simplify the process of incorporating collaborative functionalities into any application. It provides applications with certain essential collaborative functionalities: dissemination of information, storage, presence and transparency of location, management of members and groups, and execution of tasks. A distinguishing feature of LaCOLLA is that participants provide resources for the benefit of the group. This enables collaboration activities to take place in a collective environment using only the resources provided by participants in the collaboration (self-sufficiency). In this paper we present and evaluate the architecture of LaCOLLA, its API, and key aspects of its implementation.

Keywords: Collaborative middleware, peer-to-peer collaborative systems, peer-to-peer middleware, LaCOLLA

1 INTRODUCTION

One of the most significant benefits of the Internet has been the improvement in people's interaction and communication. E-mail, Usenet News, Web and Instant Messaging are four of the most well-known and successful examples of this. Internet has allowed for the creation of asynchronous virtual communities where members interact on a many-to-many basis despite organisational and technical barriers. Many-to-many interaction, uncommon in the physical world, has transformed the way people learn, work together, find other people with common interests and share information, etc. In particular, collaboration has benefited from this many-to-many communication in ways hard to imagine only fifteen or twenty years ago [1]. Nevertheless, non-specific collaborative applications include only a few collaborative functionalities due to the complexity derived from:

The nature of the interaction: participants are dispersed, many-to-many collaboration, people participate in the collaboration at different times, the same person connects from different locations at different times of the day (home, work).

Idiosyncrasy of groups: several issues are involved, such as flexibility, dynamism, decentralisation, autonomy of participants, different kinds of groups (long-term, task-oriented, weak commitment groups, etc.), ownership of resources, etc.

Technical and administrative issues: participants use different devices (desktop computer, notebook, PDA, mobile phone, etc.). Guarantees for the availability of information of the group, interoperability among applications, security aspects (authorisation, access rights, firewalls), participants belonging to different organisations or departments with different authorities that impose rules and limits to aid administration, internal work and individual use, etc.

Development of applications that take into account these requirements is complex and costly; therefore applications that incorporate collaborative functionalities focus only on a few key aspects while neglecting others.

The most widely used model is the client/server centralised solution with resources used during the collaboration provided by a third-party agent (e.g. a service provider) or by one of the members of the group (i.e. a participant offers their computer as a server). Client/server solutions, or more generally speaking, all solutions that require some sort of centralisation, are simple to implement but impose technical, administrative and economic restrictions on use that interfere with the interactive nature and idiosyncrasy of the groups.

In contrast, Peer-to-Peer (P2P) systems are distributed systems formed only by the networked PCs of the participants, where all the machines share their resources (computation, storage and communication), all act both as servers and clients and there is no special node in charge of coordinating the network: coordination is the result of the collective behaviour. P2P systems are self-sufficient and self-organising, applying protocols in a decentralised way to search and locate, and sharing the

burden of object transfers. As resource provision and coordination is not assigned to a central authority, all participants have similar functionalities and there is no strict dependency on any single participant [2]. Though most popular P2P applications look only to satisfy the needs of the sum of individual interests and are focused on sharing, their decentralised behaviour has helped many people realise the potential of this model in groups of communities that want to operate without depending on third-party agents.

In this paper we present LaCOLLA, a fully decentralised P2P middleware for building collaborative applications that provides general-purpose collaborative functionality using only the resources provided by participants.

LaCOLLA pays special attention to the autonomy of its members and to the self-organisation of its components. Another key aspect is that resources and services are provided by its members (avoiding dependency on third-party agents). This provision of resources and services for the benefit of the group enables collaborative activities to take place in a collective environment only using resources provided by participants in the collaboration. The management of resources in this collective environment has three main characteristics: a) resources are provided by participants, b) each participant can decide to disconnect a resource from the group at any moment, and c) while a resource is connected to a group, the group is in charge of managing the resource and group policies are in control of the resource. This cession of resources for the benefit of the group makes sense in environments where participants share some collective motivations, as in student projects and practicals, non-profit organisations (NGO, society, club, etc.), research groups with researchers from different organisations, or groups within a company. Conversely, they are not conceived to support groups that do not share a spirit of mutual collaboration. Therefore, some additional mechanisms should be implemented to avoid free-riding and provide security guarantees.

LaCOLLA is available on the Web at `http://lacolla.uoc.edu/lacolla/`. It has an open-source licence and is written in Java.

The rest of the paper is organised as follows: Section 2 presents the requirements that a middleware for applications that support asynchronous and synchronous-like collaborative activities should satisfy. Section 3 describes the functionality and architectural aspects of LaCOLLA, along with its API. Section 4 describes the implementation of the main internal mechanisms. Section 5 presents the experimental results. Section 6 relates LaCOLLA to other works, and the paper concludes with Section 7.

2 REQUIREMENTS FOR A MIDDLEWARE TO SUPPORT ASYNCHRONOUS COLLABORATIVE ACTIVITIES

Collaborative systems have often been based on a temporal aspect of sharing: applications in which users share some "thing" at the same time are called synchronous. Applications in which the users share that thing at different times are called asyn-

chronous. Most applications fall in between. LaCOLLA is mainly designed to support asynchronous applications. However, because of how it is implemented, it can support low- or medium-interactivity synchronous applications. In this section we present the basic requirements that a middleware system should satisfy to facilitate the development of asynchronous and synchronous-like applications [3]:

Decentralisation: no component is responsible for coordinating other components. No information is associated to a single component. Centralisation leads to simple solutions, but critical components restrict participants' autonomy.

Self-organisation of the system: the system should be able to function automatically without requiring external intervention. This requires the ability to reorganise its components spontaneously when faced with failures or dynamism (connection, disconnection or mobility).

Oriented to groups: the group is the unit of organisation.

- **Group availability:** ability of a group to continue operating with some components malfunctioning or unavailable. Replication (of objects, resources or services) can be used to improve availability and quality of service.
- **Individual autonomy:** members of a group freely decide what actions to carry out, what resources and services to provide, and when to connect or disconnect. Our characterisation of individual autonomy does not mean that a member has a local copy of information for use in disconnected mode.
- **Group's self-sufficiency:** a group must be able to operate with resources provided by its members. In future versions, we plan to extend the self-sufficiency concept to allow groups to share resources among groups or to connect to commercial cluster providers when needed.
- **Allow sharing:** several applications should be able to use information belonging to a group (e.g. events, objects, presence information, etc.).
- **Security of the group:** guarantee the identity and the selective and limited access to shared information (protection of information, authentication).
- **Availability of resources:** provide mechanisms to use resources (storage, computation, etc.) belonging to other groups (public, rented, interchange between groups to improve availability, etc.)

Internet-scale system: formed by several components (distributed). Members and components can be at any location (dispersion).

- **Scalability:** in number of groups, guaranteed because each group uses its own resources.

Universal and transparent access: participants can connect from any computer or digital device, regardless of the connection location or device (e.g. a web browser).

Transparency of location of objects and members: applications do not have to worry about where the objects or members of the group are. Applications use a location-independent identifier and can access different instances as people move, peers join and leave, or any other condition changes.

Support disconnected operational mode: be able to work without being connected to the group. Very useful for portable devices.

3 LACOLLA

Four main abstractions have inspired the design process for LaCOLLA: orientation to groups; members knowing what is happening in the group; storage using resources provided by participants, and tasks that can be executed using the computational resources provided to the group by its members. These abstractions take shape in the functionality described in 3.1.

3.1 Functionality

LaCOLLA provides the following functionality to applications [3]:

Communication and coordination by "immediate" and consistent dissemination of events: information about what is occurring in the group is spread among members of the group as events. All connected applications (and, thus, members) receive them immediately as they occur or, in the worst case, within a time limit seen to be sufficiently immediate. Disconnected members receive this information during the re-connection process. This immediate and consistent dissemination of events helps applications to keep members up-to-date. It also helps applications coordinate and communicate in a decentralised and distributed manner.

Virtual strong consistency in the storage of objects: components connected to a group have access to any object. Objects are replicated in a weakly-consistent optimistic manner. Therefore, when an object is modified, different replicas of the object will be inconsistent for a while. However, LaCOLLA has an internal mechanism to resolve the location of objects. The latest version of the object will almost always be provided (see the validation section).

Execution of tasks: members of a group (or the applications they use) can submit tasks to be executed using the computational resources provided to the group. This functionality is in an initial stage. In the current version tasks are Java programs. It allows members to benefit from unused partners' computational resources. We are currently extending it to support the deployment of nomadic group services. Examples of this kind of service include a service to coordinate dynamic and volatile aspects of a synchronous collaborative activity, a group-level session awareness service, a publish/subscribe service, or any other service that can add value to groups.

Presence: know what components and members are connected to the group.

Location transparency: applications do not have to know the location (IP address) of objects or members. LaCOLLA resolves them internally.

Instant messaging: send a message to a subgroup of group members.

Management of groups and members: The system can administrate groups and members by adding, deleting or modifying information about them.

Disconnected mode: allow applications to operate offline. During re-connection, the middleware automatically propagates the changes and synchronises them. This functionality is not implemented in the current version.

3.2 Architecture

The architecture of LaCOLLA [3] is organised into five kinds of components (Figure 1). Each member decides to instantiate any number of the following components in the peer used:

User Agent (UA): interacts with applications (see Section 3.4 for a more detailed explanation). Users (members of the group) are represented in LaCOLLA through this interaction.

Repository Agent (RA): stores objects and events generated in the group persistently.

Group Administration and Presence Agent (GAPA): in charge of the administration and management of information about groups and their members. It is also in charge of the authentication of members.

Task Dispatcher Agent (TDA): distributes tasks to executors. If all executors are busy, the TDAs queue tasks. It also ensures that tasks are executed even if the UA and member disconnect.

Executor Agent (EA): executes tasks.

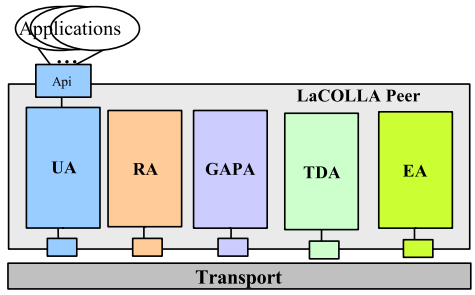

Fig. 1. LaCOLLA peer

Components interact with each other autonomously, i.e. each component behaves according to the local information and no component coordinates other components. The coordination among the components connected to a group is achieved through internal mechanisms. Internal mechanisms [3] are grouped into: events, objects, tasks, presence and location, groups, members and instant messaging. Table 1 describes which components are involved in each type of mechanism. In Section 4 we provide a more detailed explanation of how LaCOLLA works.

Types of Mechanism	UA	RA	GAPA	TDA	EA
Events	X	X	–	–	–
Objects	X	X	–	–	X
Tasks	X	–	–	X	X
Presence and Location	X	X	X	X	X
Instant Messaging	X	–	X	–	–
Groups	X	X	X	X	X
Members	X	–	X	–	–

Table 1. Types of mechanism implemented by each kind of component

3.3 An Example of a LaCOLLA Group

Figure 2 is a snapshot of a collaborative group using applications connected to LaCOLLA. Each member provides the group with whatever resources she/he chooses. This decision depends on the capacity and connectivity of the computer the member is using and members' degree of involvement in the group. In this example, two members (C and D) provide all possible components (RA, GAPA, EA and TDA). Another two members (B and F) provide all components except execution components (RA and GAPA). Three of the members (A, E and G) provide no resources to the group.

Member D (represented by dotted lines) is not connected to the group at this moment in time. However, her/his peer is connected to the group, providing it with all its resources. This means that all the events and some of the objects generated could be stored on her/his LaCOLLA peer (RA), tasks could be executed or planned using its resources (EA, TDA), users could be authenticated by their peer (GAPA), and member and group information could be also stored on it.

Applications connected to the group share presence, members and group information. Thus, users need not register for each application. Presence information is provided even if they are using different applications. LaCOLLA middleware also aids the sharing of information among applications (if compatible formats are used) due to the fact that information, events and objects are stored in LaCOLLA storing resources (RA).

A collaborative learning group in a virtual university such as the Open University of Catalonia (UOC) could take the form shown in the figure. The learning project

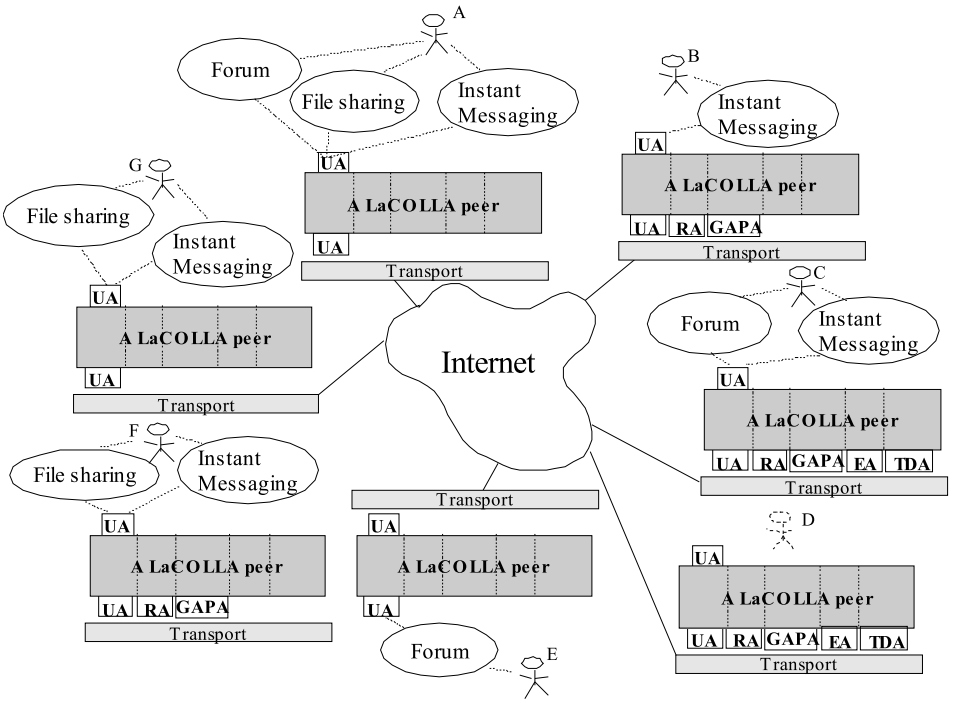

Fig. 2. Snapshot of a collaborative group that collaborates using three applications: file sharing, asynchronous forum and instant messaging

could be software development or a case study. In such a case, a member of the group initiates the group (by providing at least one RA and one GAPA) and invites other members (who contribute more resources and components to the group). From that point on, the group operates using the resources provided by its members. When any member (including the initiator) disconnects its resources or is removed from the group, the group remains operational. The group exists as long as members provide resources, and ceases to exist when they stop doing so.

LaCOLLA is independent of the applications that use its functionality. Many applications involved in collaborative tasks (not just the kind presented in the figure) could benefit from the collaborative functionality that LaCOLLA provides.

3.4 LaCOLLAs API

As can be seen in Figure 3, the LaCOLLA API is divided into two parts. One part is used by applications to invoke LaCOLLA functions on the UA where they are connected (invoking functions in `UASideApi`). The other part is used by LaCOLLA to make notifications to applications (invoking functions in `ApplicationSideApi`). Table 2 and Table 3 contain the list of functions. In the current version, Java RMI

is used to publish and invoke each part of the API. However, to use the API from an application not developed in Java simply requires building a module that translates parameters and results to and from Java. For instance, if the application is written in C/C++, JNI (Java Native Interface) can be used.

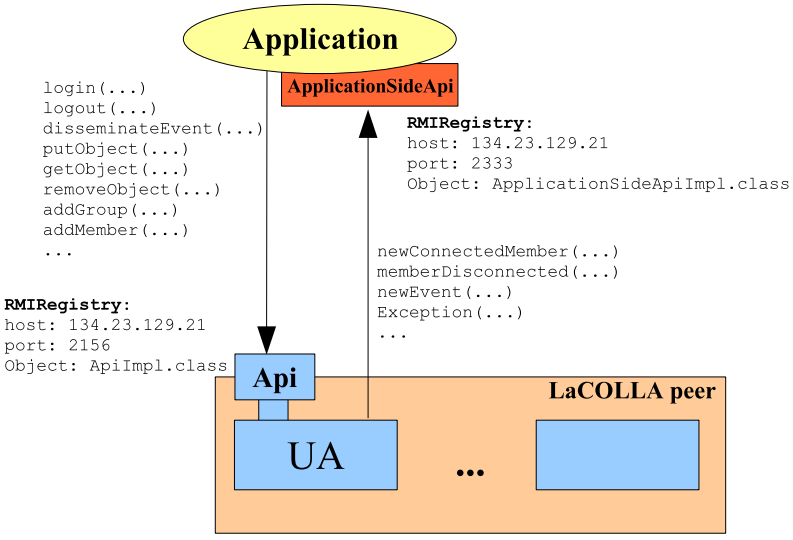

Fig. 3. The LaCOLLA API has two parts. Applications use the UA's API to ask LaCOLLA to carry out actions. The other part of the API is provided by the applications to the UA and is used to notify applications of events or information. The UAs invoke functions in the `ApplicationSideApi` class. This class is provided with the middleware and must be extended to any application that wants to use LaCOLLA.

4 LACOLLA INTERNAL IMPLEMENTATION

This section outlines the basics for the implementation of LaCOLLA's internal mechanisms. They offer the collaborative functionality described in Section 3.1 in a decentralised and self-sufficient way. Internal mechanisms are implemented using optimistic replication techniques. Optimistic techniques *allow users to access any replica at any time, based on the optimistic presumption that conflicting updates are rare, and that the contents are consistent enough with those on other replicas* [4]. These techniques are especially interesting in our case because they provide availability, flexibility, autonomy of components and quick feedback in dynamic and decentralised environments. In the current implementation, components communicate by passing messages. Messages are serialised Java objects sent using TCP sockets.

Category	Function	Description
Presence	`login`	Connects user to group.
	`logout`	Disconnects user from group.
	`whoIsConnected`	What members are connected to the group?
Events	`disseminateEvent`	Sends an event to all applications belonging to group.
	`eventsRelatedTo`	What events have occurred to a specific object?
Objects	`putObject`	Stores an object into LaCOLLA.
	`getObject`	Obtains an object stored into LaCOLLA.
	`removeObject`	Removes an object stored in LaCOLLA.
Tasks	`submitTask`	Submits a task to be executed by computational resources belonging to group.
	`stopTask`	Stops a task.
	`getTaskState`	What is a task's state?
Instant Messaging	`sendInstantMessage`	Sends a message to specified members of the group.
Groups	`addGroup`	Creates a new group.
	`removeGroup`	Removes a group.
	`modifyGroup`	Modifies the properties of a group.
	`getGroupInfo`	Gets information about the properties of a group. (Asynchronous function. See groupInfo function in Table 3).
	`getGroupInfoSync`	Gets information about the properties of a group in synchronously. This function does not return until the operation is completed and a result is available.
Members	`addMember`	Creates a new member.
	`removeMember`	Removes a member.
	`modifyMember`	Modifies the properties of a member.
	`getMemberInfo`	Gets information about the properties of a member.

Table 2. API functions that User Agents offer to applications

4.1 Main Techniques and Protocols Used in LaCOLLA

To provide self-sufficiency and availability in a dynamic and decentralised manner we decided to use a multi-master optimistic approach [4]. Multi-master systems enable updates to be issued to multiple replicas independently and exchanged in the background. These kinds of systems offer more availability than single-master systems, where all updates originate in one replica and are then propagated to other replicas, but are significantly more complex. In multi-master systems, updates can be propagated in two ways: state-transfer (sites exchange object contents), and operation-transfer (sites exchange operations). We used both ways of propagation.

Category	Function	Description
Presence	`newConnectedMember`	Notifies the connection of a member.
	`memberDisconnected`	Notifies the disconnection of a member.
Events	`newEvent`	Reception of an event occurring in the group.
Tasks	`taskStopped`	Notifies that the task has been stopped correctly.
	`taskEnded`	Notifies the ending of a task.
Instant Messaging	`newInstantMessage`	Reception of an instant message.
Groups	`groupInfo`	Reception of the group information. (See getGroupInfo in Table 2).
Other	`exception`	Notifies that an internal exception or anomalous situation has occurred.
	`appIsAlive`	UA queries the state of the application. Used to know if an application is still alive and connected to group.

Table 3. API functions that applications should offer to User Agents

Components (masters) coordinate in an optimistic manner combining multicast and epidemic propagation of information [7].

The Time-Stamped Anti-Entropy (TSAE) protocol [5] was originally designed as a group communication service, although it is also used as an optimistic replication service. We used the TSAE protocol together with multicast of events. Multicast of events brings new events to connected components: *"Quick propagation is the best way to avoid conflicts"* [4]. It provides immediate awareness to connected members and reduces divergence among replicas. TSAE consistency sessions complement multicast, providing the guarantee that all components will eventually receive all events. It works by propagating data in the background in an epidemic fashion, allowing disconnected components or components that did not receive the event when it was multicasted to receive it (e.g. due to a failure during the sending or because the sender did not know that the component was connected).

In the TSAE protocol, each site periodically contacts another site and the two sites exchange messages from their logs until both logs contain the same set of messages. To guide the exchanges, each site maintains a data structure that summarises the messages each site has received. The sender timestamps each message using a local timestamp. The summary is built by selecting the last consecutive timestamp received from each site. When the anti-entropy session starts, each site sends the summary of received messages to the partner site. Each site determines if it has messages the other has not received, by seeing if some of its summary timestamps are later than the corresponding ones of its partner. These messages are retrieved from the log and sent to the other site.

We used TSAE in the events mechanism to guarantee the consistency of events. We adapted unsynchronised-clocks TSAE protocol [6] because it allows sites to be far out of synchronisation without compromising the progress of the system. In our variant, a site accepts (stores in the log) and propagates an event although the site misses previous events issued by the same originator. This allows for faster propagation of information. The log summary has two data structures. The first one, *all_consecutive*, is the common timestamp matrix, as used in Golding's TSAE, where the consecutive events received from each of the sites of the group are summarised. The second one, *non-consecutive*, is used to store the timestamps of out-of-order events received (i.e. events stored in the log without having all previous events from the same source). A component can receive a non-consecutive event if it missed a previous one due to the fact that the sender did not know the component was connected, or because a failure occurred while sending it. When a site receives a consecutive event, it summarises (in *all_consecutive*) the events contained in the non-consecutive summary.

In the anti-entropy sessions, the two sites exchange both summaries: summary matrix (consecutive events) and the non-consecutive summary.

Several of the mechanisms implemented in LaCOLLA use anti-entropy sessions to achieve consistency of information: events, presence and location, groups, members and execution.

4.2 Internal Mechanisms

There are seven kinds of internal mechanisms: events, objects, presence and location, execution, instant messaging, groups, and members. Each kind of mechanism is divided into several sub-mechanisms. It is beyond the scope of this paper to present a fully detailed description of the implementation of all of them. This description can be found on [3, 8]. Alternatively we are going to explain the general behaviour of internal mechanisms that characterise LaCOLLA middleware and outline the key aspects to understand its functioning.

Typically collaborative activities involve a small number of participants. Therefore, groups in LaCOLLA are considered to have a small number of members and components (as stated in the validation section, LaCOLLA can deal with big groups of 100 or more components but, to be realistic, collaborative groups should typically have 5, 10 or 20 members, not more).

4.2.1 Presence and Location Mechanism

The presence and location mechanism is one of the two keystones to the LaCOLLA middleware. Apart from informing applications about which members are connected to the group, the presence and location mechanism is used by other mechanisms to know what components are connected to the group. Due to the decentralised, autonomous and self-organising nature of LaCOLLA, it is not easy to know what components are connected to the group. In fact, no component can ever claim to

know all the components connected, even though the optimistic techniques used guarantee that all components have a good enough perception of what components are connected, and these perceptions eventually converge (see the validation section).

The presence and location mechanism is implemented in a decentralised multi-master state-transfer optimistic approach. When a component wants to connect to a group, it sends its authentication information to a GAPA. If the component is authenticated, the GAPA facilitates the connecting component those components the GAPA knows are connected to the group (transfers the presence and location local state). Then, the connecting component multicasts a message informing of its connection to the group to all other components that it knows are connected to the group (those that the GAPA informed it as being connected to group). This message, like any other message sent by any LaCOLLA component, includes information about the components the sender knows are connected to the group (sender's state). Comparing the sender's state with its local state, receivers can learn about unknown connected components.

To reinforce this epidemic propagation of information about connected components, from time to time, a component randomly selects N components[1] and performs an anti-entropy consistency session with any of them. During a consistency session between two sites, say A and B, A tells B its state (what components A knows are connected to the group) and B informs A of its own knowledge. Both learn from each other's state. This epidemic transfer of senders' states regarding presence and location information allows LaCOLLA to achieve a consistent view of the connected components.

Prior to an ordered disconnection, the disconnecting component informs other components about its intention and each component updates its local state.

Finally, there is a sub-mechanism to detect components that are no longer connected to the group. Component A suspects that another component B is disconnected either when a) A has not received a message from B after a long period of time[2], or b) when components from whom A receives messages have not received a message from B for a long period of time. In that case, A sends a message to B. If B does not answer, A decides that B is disconnected and removes B from its connected components list.

Components decide that a component has not sent a message for a long period of time as follows: components have a counter for every component that they know to be connected to group. This counter is created and initialised by any component when it receives the *newConnectedComponent* (message that every component multicasts when connecting) and destroyed when the component sends the message informing of its disconnection. It is initialised to a given value[3].

[1] $N = Max(2, \log_2(\mathit{numberConnectedComponents}) + 1)$. This number was adjusted by simulation.

[2] Configuration parameter.

[3] Configuration parameter. Current value adjusted by simulation.

This counter has an associated timer that reduces it every time the timer expires. When a component receives a message from another component (the message contains sender's information about the known components and its originator's value for the counter), it compares presence information from the sender with local information. For every component, it selects the maximum between the local and remote counter. The receiver also initialises the counter for the sender of message (meaning that the sender is alive). This provides an optimistic approach to connected members. When the counter reaches zero it means the component is thought to be disconnected.

The presence information also includes the location of known components. Every time a component changes its location, it multicasts a message informing of its new location. Components that do not receive the message will eventually learn of it from information included in received messages (epidemic propagation) or through an anti-entropy presence consistency session. The receiver will know that the component has changed location because the received timestamp associated to the location information for the component will be newer than the local timestamp for the same component. Each component generates its location timestamp when connecting or each time it changes location dynamically. The timestamp is initialised when connecting and increases monotonically with every dynamic change of location.

The presence and location mechanism implements the presence and location functionality from Section 3.1.

4.2.2 Events Mechanism

The events mechanism is the other keystone in the LaCOLLA middleware. Almost all other mechanisms (except the presence and location mechanism) rely on the events mechanism to deliver information in an immediate and consistent way. The objects mechanism, in particular, depends on the events mechanism to achieve its consistency. The events mechanism is implemented with a multi-master operation-transfer optimistic approach. When an action occurs, an event is created to inform of this action. Events can be provided by applications or automatically generated by LaCOLLA as a consequence of an internal action related to the functionality provided by the middleware (e.g. new document, new member, document read, etc.). In LaCOLLA, events are used by each component to autonomously determine the state of the system.

When a component has a new event, it multicasts it to all the UAs and RAs that it knows to be connected. All the RAs store all received events persistently.

Events are sent by applications to the UA the application is connected to or are internal events. In the first case, the UA is responsible for multicasting the event to all connected components. Internal events are sent by the component where the action originates. Multicasting events as soon as they occur by the component where the event has originated provides members with the perception and knowledge of what is occurring in the group while it is occurring, i.e. all connected members receive information about what is going on in the group in a time that they perceive

as immediate. Therefore, to provide immediacy, the performance of this mechanism is closely related to the presence mechanism.

Components not connected to the group or components that the sender of the event does not know to be connected to the group will not receive the event. Therefore, we implemented an epidemic sub-mechanism that complements the multicast. It uses the variant of Golding's TSAE protocol explained in Section 4.1. Consistency sessions are performed between two RAs and synchronisation sessions between a UA and an RA. Consistency sessions work as follows: from time to time, an RA (e.g. A) randomly selects another RA among the RAs that A knows to be connected to the group and they perform an anti-entropy session.

Similarly, the synchronisation sessions between UAs and RAs work like consistency sessions but asymmetrically; i.e. UAs learn events from RAs, but not the other way around.

This implementation of the events mechanism allows members to receive information about what is going on in the group in a time that they perceive as immediate (see the validation section). Multicast provides this immediacy. In addition, TSAE consistency sessions guarantee that all components will eventually receive the information whether or not they were connected to a group when the information was multicasted, or whether or not there has been any failure during the multicast of information.

On the other hand, no guarantees in terms of ordering are provided. In the case where some ordering guarantees are required, applications should provide them. For example, in the case where an application needs to process events from the same originator in the same order as they were issued, the application should not process the non-consecutive events until it has received all previous ones. As future work we are considering the implementation of some event-ordering policies that could be added on top of LaCOLLA and provided to the applications.

The events log contains all events sent in the group. Some events are related to the state of the system (e.g. new object, remove object, etc.). To prevent logs from growing indiscriminately, we have implemented a selective purge mechanism that allows components to autonomously purge events, without modifying the state of the system. This mechanism implements communication and coordination by immediate and consistent dissemination of events, as seen in Section 3.1.

4.2.3 Objects Storage Mechanism and Virtual Synchronism

In the events mechanism section, we saw that LaCOLLA guarantees applications that all events will be received almost immediately by connected applications (and members). Likewise, that disconnected members will receive the events during the re-connection process. This provides users and applications with an up-to-date view of the collaboration that is taking place in a group. It also has an important side effect that we have exploited in our middleware: since all components know the location of all replicas of any object (because they have received an event informing them of such), components can access them directly (without a resolver informing

about the location of objects). This allows LaCOLLA to have an autonomous and decentralised policy for handling objects and their replicas at the same time as it guarantees immediate access to the 'most likely' latest version of stored information. We say the 'most likely' latest version because an application may miss the multicasted event informing about a new version of the object and ask for the object before the anti-entropy mechanism has propagated the existence of the new version. As can be seen in the validation section, this divergence lasts only a few seconds. We assumed that the probability of a user wishing to access the object during this period of divergence is very low. We refer to the sum of immediate and consistent dissemination of events (knowing what is happening in the group) and the ability to access the likely last version of any object as virtual synchronism. (See the validation section for more details.)

As mentioned above, the object storage mechanism depends on the events mechanism. UA and RA components use the events that inform about stored objects (e.g. new object, removed object, new replica, etc.) to build the state of stored objects autonomously. Each UA and RA knows all stored objects and their locations (as objects are replicated, they will have multiple locations). When a component needs a specific object it queries its local information about stored objects and randomly selects a location from among the connected RA that have the object. In more detail: when an object is provided to a UA to be stored in LaCOLLA, the UA randomly selects an RA connected to the group and sends the object to it. Then, the RA disseminates an event to all components (UAs and RAs) to inform them of the new object and its location. This event is used by all components to know where the object is located. Therefore, when an UA wants to retrieve an object, it can be obtained from any of its locations. Finally, to guarantee the availability of objects, they are automatically replicated in an autonomous and decentralised manner: from time to time, an RA copies to other RAs the objects it has that have less replicas than the replication factor for the group. The destination RAs are selected randomly among the RAs connected that do not have a replica of the object. Every time a new replica is created, an event is disseminated to inform of the availability of the object in its new location.

We used random decision techniques to locate objects and their replicas because this technique has proved a good trade-off between efficiency and simplicity [9]. The location algorithm can be improved in future work by means of a more in-depth study into the components' behaviour in real scenarios.

LaCOLLA does not have a modify operation. To modify objects, applications must create a new object. The event informing about the new object must contain information indicating that it is a new version of another object. Applications have to detect and resolve conflicts if one object is modified simultaneously by more than one user. In the current version, LaCOLLA only provides for the storage of objects. We are currently working on the development of a file system module. This module will be deployed on top of LaCOLLA and will help applications to deal with conflicts and their resolution.

This mechanism implements the storage mechanism as seen in Section 3.1.

4.2.4 Summary

To sum up, although push is frequently used in all the mechanisms, neither components nor the network are saturated because groups are small. The combination of autonomy of components and direct communication among them (peer-to-peer), along with the collective ownership of resources, provide a flexibility that suits the idiosyncrasy of collaborative groups.

Finally, Table 4 summarises the main techniques used to implement all internal mechanisms.

5 VALIDATION

LaCOLLA was validated in two steps. In the first step we implemented the basic mechanisms that characterise LaCOLLA (presence and location, events and storage of objects) in a simulated environment. The second step was to implement LaCOLLA as real middleware. We implemented some collaborative applications and used them.

The first step helped us prove that LaCOLLA provides virtual synchronism to participants, that it is able to self-organise and that it behaves in an autonomous, decentralised and self-sufficient manner. This section will be entirely dedicated to this step. The implementation and adaptation of collaborative applications (an instant messaging tool, an asynchronous forum, and a document sharing tool) carried out in the second step reinforced our belief about the usefulness of LaCOLLA. In addition, it helped us improve its architecture and implementation. After these two successful steps, we are working on the extension and improvement of current functionality with the aim of being able to use LaCOLLA and the implemented collaborative applications in regular university courses at the UOC (Open University of Catalonia).

To validate LaCOLLA we used J-Sim [10] as the network simulator. We implemented the UA, RA and GAPA components and presence, events and storage internal mechanisms.

Several experiments were done with synthetic workloads with different degrees of dynamism (failures, connections, disconnections or mobility), with different sizes of groups (from 5 to 100 members) and with different degrees of replication (number of RAs and GAPAs). All components were affected by dynamism. Full information about experiments can be found on [3].

The simulations involved two phases. The first phase simulated a realistic situation. In this phase all internal mechanisms were operative. During this phase members' activity was simulated. Components connected, disconnected, moved or failed. The second phase was named the repair phase and only internal mechanisms were active. This second phase was used to evaluate how long it would take to achieve:

Mechanism	Techniques used to implement the mechanism
Events	Multi-master operation-transfer. Multicast: the component where the event originates sends the event to all UAs and RAs that it knows to be connected. TSAE: anti-entropy consistency sessions among components to obtain events not received when they were multicasted (due to disconnection, mobility or failure). Works in epidemically. Sessions carried out in background. Complements multicast.
Storage of objects	Random decision technique: to select location of an object when it is stored in LaCOLLA. UA randomly selects an RA from among the RAs connected. Also used by an RA to select the RA to store a new replica of an object. Push update: the RA with the object originates the replication of the object and sends it to the target RA.
Presence and location	Multi-master state-transfer. Multicast: an event is sent every time a component connects to or disconnects from LaCOLLA. Epidemic: every message includes information about known connected components and their location (optimistic. Receiver can learn of unknown components). Anti-entropy: consistency sessions among components to learn about unknown connected components.
Execution	Multi-master state-transfer. Multicast: the UA sends information about the task to be executed to all TDAs. TSAE: between TDAs to make the information about planned tasks consistent.
Instant messaging	Multicast: to destination components.
Groups	Multi-master state-transfer. Multicast: to disseminate a modification. TSAE: consistency sessions among GAPAs to guarantee that all GAPAs have the same information about groups.
Members	Multi-master state-transfer. Multicast: to all GAPAs when a new member is added. Also for modifications. TSAE: consistency sessions among GAPAs to guarantee all GAPAs have the same information about members.

Table 4. Summary of the main techniques used to implement each kind of internal mechanism

a) self-organisation: all connected components having consistent information about all internal mechanisms;

b) virtual synchronism: all connected components having all events and consistent information about available objects (in our analysis we have evaluated separately the performance of the events mechanism and objects mechanism); and

c) presence and location: all connected components having consistent information about presence and location.

Experiments showed that, despite the dynamism and the autonomous and decentralised behaviour of components, LaCOLLA required a short amount of time to obtain up-to-date information on all components. Experiments also showed that members knew what was happening in the group and that they had access to objects in a time they perceived as immediate [3].

Component	Failure		Disconnection		Mobility
UA	0.0005	[15,120]	0.0025	[60,540]	0.00035
RA and GAPA	0.000125	[12,60]	0.0005	[15,120]	0.0001

Table 5. Degree of dynamism. For each kind of component, we show the probability (per iteration) that each type of dynamism occurs. For failures and disconnections, an interval is provided. The application remains in failure or disconnection for a random time within the interval (in iterations). Iteration: 10 seconds.

Considering iterations every 10 seconds, Figure 4 shows the cumulative probability that in N seconds ($N/10$ iterations) LaCOLLA:

a) has consistent information about events in all components;

b) has consistent information of objects in all components;

c) is self-organised; and

d) information about presence and location is consistent in all components.

The data for the figures comes from experiments with 10 RAs and 10 GAPAs that connect, disconnect, fail and move in a moderate degree (Table 5 contains information about dynamism. Information about parameters related to synthetic members' activity can be found on [3]). Note that, for groups of a typical size (10 members), LaCOLLA offers good performance: in 95 % of experiments, all components had consistent information about events, objects and presence and location at the end of the first phase; and in less than 10 seconds all this information is consistent in all components. This is the case of components that the sender did not know were connected at the moment the information was multicasted or that did not receive the information due to some failure. Having all information consistent in all components, "self-organisation" requires more time (up to 80 seconds). This is due to the decentralised implementation of internal mechanisms and to the fact that non-key mechanisms have long-term consistency policies. Special attention should be given

to the fact that, even though all components do not have completely consistent information about internal mechanisms (self-organisation), connected members know all that is happening in the group (events) and have access to objects in a time that they perceive as immediate. In Section 4, we named this up-to-date view virtual synchronism. Provision of virtual synchronism is what allows us to say that LaCOLLA is useful for collaborative environments.

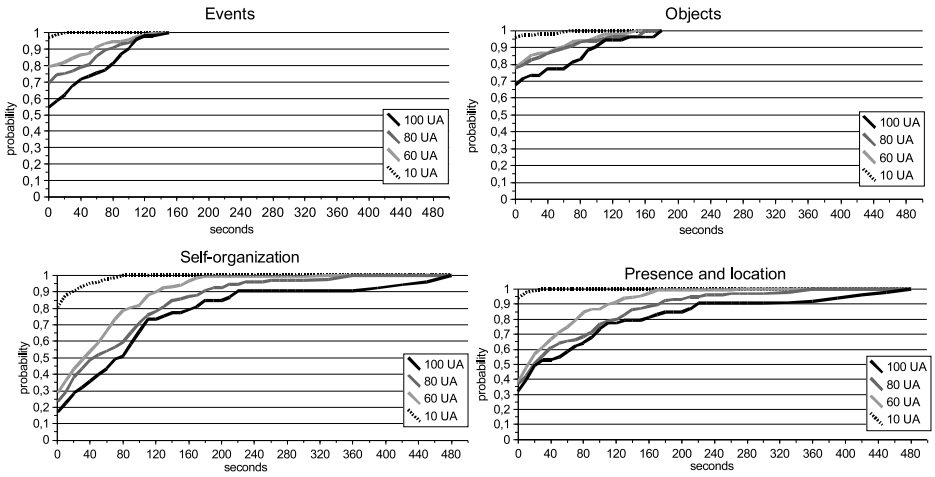

Fig. 4. Cumulative probability that in N seconds a) information about events is consistent in all components (events), b) information about objects in all components is consistent and objects are replicated at least replication_factor times (objects), c) all components of LaCOLLA have a consistent view (self-organisation), and d) information about presence and location is consistent in all components (presence and consistency).

In Figure 4, the effect of scaling the size of groups can also be seen. In groups of up to 100 members, virtual synchronism (the worst case between events and objects mechanisms) is provided in a time that participants perceive as immediate. With 100 members, the worst case is 180 seconds. In contrast, in 90 % of experiments, the components require 90 seconds to have consistent events and objects mechanisms, which still provides a feeling of immediacy to members. On the other hand, the presence and location mechanism and the self-organisation of LaCOLLA perform poorly when group size increases. Nevertheless, as stated above, what is most important for collaborative applications is the provision of virtual synchronism.

When the size of the group decreases, all mechanisms converge faster to a consistent situation. The only consideration is that dynamism has more influence. If there is not at least one RA and one GAPA connected to group, the group cannot operate.

In conclusion, LaCOLLA provides immediacy that is good enough for small-size collaborative asynchronous applications and for small-size synchronous-like collaborative applications that can bear some users not being momentarily up-to-date. LaCOLLA also provides good support for medium-scale asynchronous collaborative applications.

6 RELATED WORK

A number of research and commercial systems have used optimistic replication techniques. Each of the features of LaCOLLA's internal mechanisms related to propagation and synchronisation of information have almost certainly appeared in previous systems in some form. Interesting differences lie in the implementation details used in LaCOLLA that allow LaCOLLA to have self-organised and self-sufficient behaviour using the connected resources provided for the benefit of the group. Furthermore, we present some systems that have influenced the design of LaCOLLA.

BSCW system [11] is a web-based application for collaborative information sharing based on the metaphor of shared folders as a repository for group information. BSCW provides awareness information about all the objects within the system (events). It has two limitations:

1. it does not take into account events results of local actions,
2. it can be used from any web browser, but it is a centralised system.

BSCW is an example of an application that could benefit from the functionality provided by LaCOLLA.

Coda [12] and Ficus [13] are general-purpose replicated distributed file systems intended to facilitate distributed collaboration in a highly reliable and scalable fashion. Both file systems allow updates so long as at least one replica of a data object is available. While Coda has clients and file servers, in Ficus each machine, including workstations, portable computers and servers, should be empowered with full functions where replication, file service and reconciliation are concerned. In this sense, all machines are peers. In contrast, LaCOLLA provides decentralised storage in resources provided for the benefit of the group.

Lotus Notes [14] and Bayou [15] are replicated database systems based on epidemic update propagation. DOORS [16] is a replicated object store. All three offer a disconnected operational mode. The main difference with LaCOLLA is that in LaCOLLA objects are stored using the resources provided by members.

Groove is a platform that partially covers some of the ideas behind the LaCOLLA approach. Groove is defined [17] as a system that enables users to create shared workspaces on their local PCs, collaborating freely across corporate boundaries and firewalls, without the permission, assistance, or knowledge of any central authority or support group. Groove allows transparent synchronisation among workspaces, but depends on relay servers to provide offline queuing, awareness, fan-out and transparency (to overcome firewall and NAT problems). These relay servers are

provided by third parties. The main differences between Groove and LaCOLLA are that

1. Groove emphasises transparent synchronisation of collaborating PCs, along with direct communication among them, and
2. the fact that Groove provides third-party relay servers.

In contrast, LaCOLLA allows members to provide resources for the benefit of the group and articulates all the collaboration using these resources only; i.e. participants are not obliged to provide resources to groups, but groups only work with resources provided by their members. All resources connected to a group are synchronised transparently. Unfortunately, due to the fact that Groove is a closed source commercial product, we were not able to find more details as to how it works.

Finally, JXTA [18] is a P2P platform to support the development of P2P applications. The main differences with LaCOLLA are that JXTA is lower level middleware than LaCOLLA. JXTA also uses generic mechanisms to discover and connect components. In contrast, LaCOLLA is designed to support collaborative environments. Its mechanisms are specifically designed for decentralised and self-sufficient collaborative environments.

7 CONCLUSIONS AND FUTURE WORK

LaCOLLA is a middleware that aims to simplify the process of incorporating collaborative functionalities in any application. Our proposal allows the building of applications that enable groups of people to join together and self-organise in an autonomous, decentralised and self-sufficient manner as they are used to doing in non-virtual environments. Self-sufficiency (i.e. operating using only resources provided by participants), presents a promising opportunity to build collective environments where participants provide resources for the benefit of the group. This cession of resources for the benefit of the group makes sense in environments that share collective motivations.

In this paper we described the architecture, API and internal implementation of LaCOLLA, a fully decentralised peer-to-peer middleware. In the validation section we proved that LaCOLLA provides participants with the virtual synchronism property: participants know what is happening in the group and have access to objects stored in the group.

From both building collaborative applications that use LaCOLLA, and from using the developed applications, we have obtained valuable ideas and improvements to introduce into future releases of the software. These new versions will pay special attention to security issues (which were not prioritised in the first version); to the deployment of services at group level using the computational resources provided to the group by participants; to introducing new components and mechanisms that will allow mobile devices (PDAs, mobile phones, sensors, etc.) to become LaCOLLA peers, emphasising the capabilities related to the disconnected mode of

operation, and to introducing new components and mechanisms to allow groups to share resources.

We also plan to use LaCOLLA in real collaborative settings. Thus, we intend to use collaborative applications that employ LaCOLLA middleware in some collaborative learning practicals at the UOC (Open University of Catalonia). The UOC is a virtual university that mediates all relations between students and lecturers over the Internet. We believe that these kinds of collaborative environments where participants never physically meet each other will benefit from approaches like the one provided by LaCOLLA, especially in terms of the degree of autonomy and self-sufficiency that can be achieved. These real experiences will be of great value for us to further refine the architecture and adjust the implementation of the middleware.

Acknowledgements

This work was partially supported by TSI2005-08225-C07-05 and TIC2002-04258-C03 projects.

REFERENCES

[1] Daradoumis, T.—Martínez, A.—Xhafa, F.: An Integrated Approach for Analysing and Assessing the Performance of Virtual Learning Groups. Lecture Notes in Computer Science (LNCS 3198). Groupware: Design, Implementation, and Use. G.-J. de Vreede et al. (Eds.). Heidelberg-Berlin: Springer-Verlag, pp. 289–304, ISBN: 3-540-23016-5.

[2] Navarro, L.—Marquès, J. M.—Freitag, F.: On Distributed Systems and CSCL. The First International Workshop on Collaborative Learning Applications of Grid Technology. Held in conjunction with the IEEE International Symposium CCGrid 2004. April 19–22, 2004, Chicago, USA.

[3] Marquès, J. M.: LaCOLLA: An Autonomous and Self-Organised Infrastructure to Aid Collaboration. 2003. Ph. D. thesis. Available on: `http://people.ac.upc.es/marques/LaCOLLA-tesiJM.pdf` (in Catalan).

[4] Saito,Y.—Shapiro, M.: Optimistic Replication. ACM Computing Surveys (CSUR). Vol. 37, 2005, No. 1, p. 42–81.

[5] Golding, R. A.: Weak-Consistency Group Communication and Membership. Doctoral Thesis, University of California, Santa Cruz, 1992.

[6] Agrawal, D.—Malpani, A.: Efficient Dissemination of Information in Computer Networks. Computer Journal, Vol. 34, 1991, No. 6, pp. 534-541.

[7] Demers, A. J.—Greene, D. H.—Hauser, C.—Irish, W.—Larson, J.: Epidemic Algorithms for Replicated Database Maintenance. In 6th Symp. On Princ. Of Distr. Comp. (Vancouver, Canada, Aug. 1987), pp. 1–12.

[8] LaCOLLA. Avaliable on: `http://lacolla.uoc.edu/lacolla`.

[9] Carter, R. L.: Dynamic Server Selection in the Internet. In Proceedings of the Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems, 1995.

[10] J-Sim. Avaliable on: `http://www.j-sim.org`.

[11] Bentley, R.—Horstmann, T.—Sikkel, K.—Trevor, J.: Supporting Collaborative Information Sharing with the World Wide Web: The BSCW Shared Workspace System, in The World Wide Web Journal. Proceedings of the 4th International WWW Conference. Issue 1, Dec. 1995, pp. 63–74. O'Reilly.

[12] Kistler, J. J.—Satyanarayanan, M.: Disconnected Operation in the Coda File System. In ACM Transactions on Computer Systems, Vol. 10, 1992, No. 1, pp. 3–25.

[13] Page, T. W.—Guy, R. G.—Heidemann, J. S.—Ratner, D. H.—Reiher, P. L.—Goel, A.—Kuenning, G. H.—Popek, G. J.: Perspectives on Optimistically Replicated Peer-to-Peer Filing. Software – Practice and Experience, Vol. 27, 1997, No. 12.

[14] Kawell, L.—Beckhardt, S.—Halvorsen, T.—Raymond, O. R.—Greif, L.: Replicated Document Management in a Group Communication System. In Proceedings of the 2nd International Conference on CSCW, pp. 205–216, Sep. 1988. Also: IBM lotus notes. Available on: `http://www.lotus.com/notes`.

[15] Edwards, K.—Mynatt, E.—Petersen, K.—Spreitzer, M.—Terry, D.—Theimer, M.: Designing and Implementing Asynchronous Collaborative Applications with Bayou. ACM Symposium on UIST '97, pp. 119–128, 1997.

[16] Preguia, N.—Martins, L.—Domingos, H.—Duarte, S.: Data Management Support for Asynchronous Groupware. CSCW 2000, pp. 69–78. 2000.

[17] Hurwicz, M.: Groove Networks: Think Globally, Store Locally. Network Magazine. May 2001. LaCOLLA.

[18] JXTA web site. Available on: `http://www.jxta.org/`.

Joan Manuel Marquès is lecturer on computer science studies at the Open University of Catalonia since 1997. He is also part-time lecturer at the Computer Architecture Department of UPC since 1995. He received his Ph. D. from UPC under professor Leandro Navarro in 2003. He also received his degree in computer sciences from UPC. His research interests include the design of scalable and cooperative Internet services and applications. He is member of the ACM (Association for Computing Machinery).

Leandro Navarro joined the Computer Architecture Department of UPC as an associate professor in 1988, after receiving his graduate degree on telecommunication engineering from UPC. He received his Ph. D. from UPC under Professor Manuel Medina in 1992. Since 1985, he is working at the Computer Architecture Department. His research interests include the design of scalable and cooperative Internet services and applications. He is member of the ACM (Association for Computing Machinery), APC (Association for Progressive Communications) (Council), IFIP TC6 WG6.4 (International Federation of Information Processing), CCD (Centre de Cooperaci per al Desenvolupament-UPC) (council), CPSR (Computer Professionals for Social Responsibility), and IEEE (Institute for Electrical and Electronic Engineers).