What is and How to use LaCOLLA.

An introduction to LACOLLA and its API.

Authors: *Joan Manuel Marquès*

Xavier Vilajosana

Date:February 2005

Index

1.Introduction

What is LaCOLLA?

LaCOLLA is a fully decentralized infrastructure for building collaborative applications and providing them general purpose collaborative functionalities.

Key aspects of LaCOLLA:

- Avoids applications to deal with complexities derived from groups/members dispersion all over the Internet.
- Resources are provided by members of the group.
- Each member can use the resources belonging to group, what augments capacity and availability
- Decentralized. Autonomy of members.

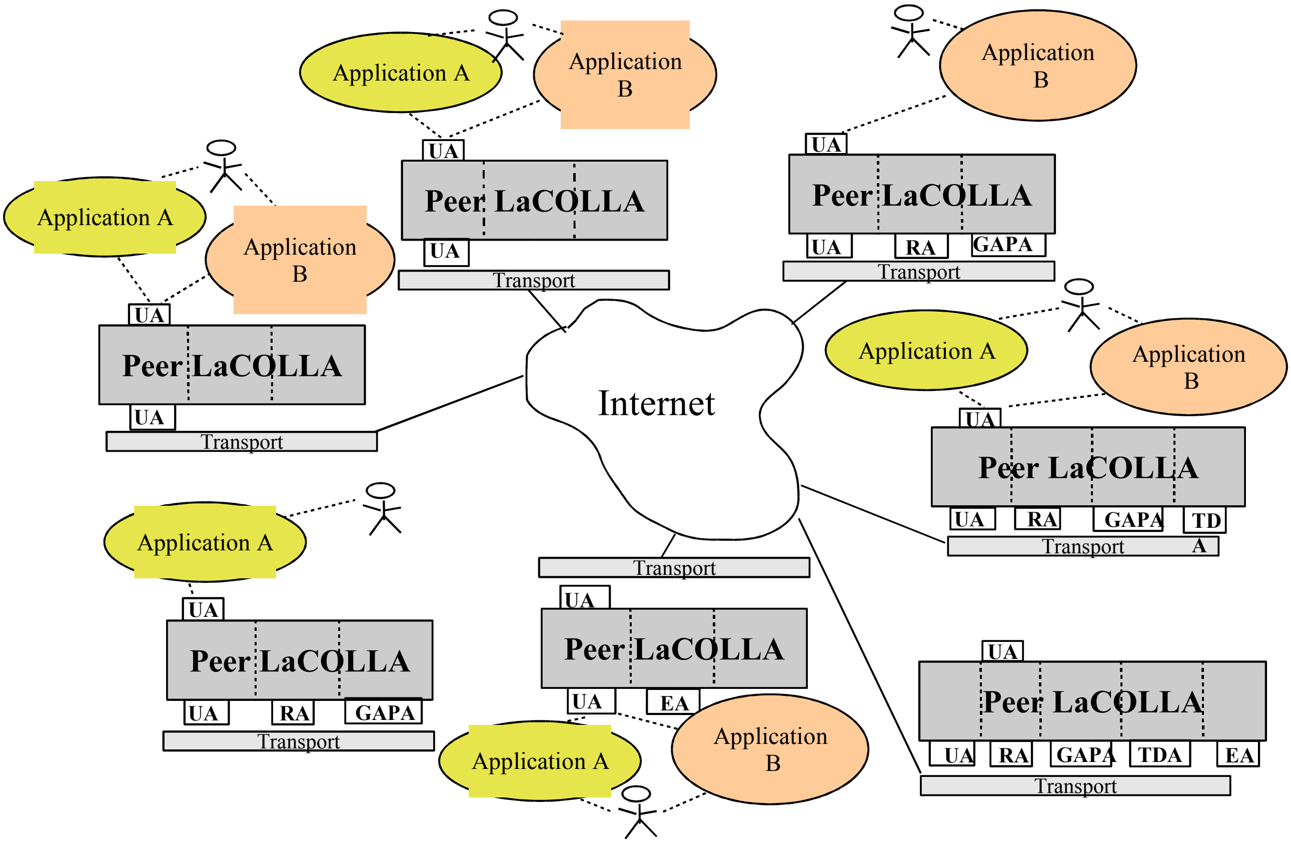

Figure 1. Example of group using application A and B in top of LaCOLLA.

Functionalities

- **Dissemination of events** (immediate & consistent): information about what is occurring in the group is spread among members of the group as events. All connected members receive this information right after it occurs. Disconnected members receive it during the re-connection process.
- **Storage** (virtually strong consistency) of objects: components connected to a group can access the latest version of any object. Since objects are replicated, when it is modified, if an application asks for that object LaCOLLA guarantees that the last version will be provided (even thought all replicas were not consistent and it will require some time to have all of them consistent).
- **Execution of tasks**: members of a group (or the applications these members use) can submit tasks to be executed using computational resources belonging (or available) to the group.

- **Presence**: know which components and members are connected to the group.
- **Location transparency**: applications don't have to know the location (IP address) of objects or members. LaCOLLA resolves them internally (similar to domain name services like DNS).
- **Instant messaging**: send a message to a subgroup of members of the group.
- **Management of groups and members**: add, delete or modify information about members or groups.
- **Disconnected mode**: allow applications operate offline. During re-connection, the infrastructure automatically propagates the changes.

Architecture

- **User Agent (UA)**: interacts with applications. Through this interaction, it represents users (members of the group) in LaCOLLA.
- **Repository Agent (RA)**: stores objects and events generated inside the group in a persistent manner.
- **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. In case any are busy, the TDAs queues them. Guarantees that tasks will be executed even though the UA and the member disconnects.
- **Executor Agent (EA)**: Executes tasks.

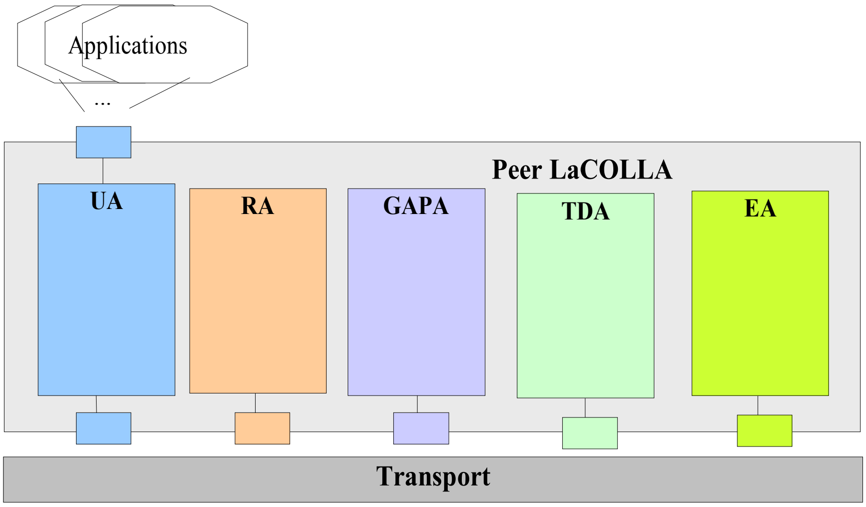

Figure 2. LaCOLLA Peer.

Components interact one to each other in an autonomous manner. The coordination among the components connected to a group is achieved through internal mechanisms. Internal mechanisms have been grouped in:

- events
- objects
- tasks
- presence

- location
- groups
- members
- instant messaging.

They are implemented using weak-consistency optimistic protocols and random decision techniques.

Requirements

- **Decentralization**: no component is responsible of coordinating other components. No information is associated to a single component. Centralization leads to simple solutions, but with critical components conditioning the autonomy of participants.
- **Self-organization of the system**: the system should have the capability to function in an automatic manner without requiring external intervention. This requires the ability of reorganizing its components in a spontaneous manner in presence of failures or dynamism (connection, disconnection, or mobility).
- **Oriented to groups**: group is the unit of organization.
  - **Group availability**: capability of a group to continue operating with some malfunctioning or not available components. Replication (of objects, resources or services) can be used to improve availability and quality of service.
  - **Individual autonomy**: members of a group freely decide which actions perform, which resources and services provide, and when connect or disconnect.
  - **Group's self-sufficiency**: a group must be able to operate with resources provided by its members (ideally) or with resources obtained externally (public, rent, interchange with other groups, ...)
  - **Allow sharing**: information belonging to a group (e.g. events, objects, presence information, etc.) can be used by several applications.
  - **Security of 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, computational, 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, with a connection independent view (e.g. as a web browser).
- **Transparency of location of objects and members**: applications don't have to worry about where are the objects or members of the group. Applications use a location independent identifier.
- **Support disconnected operational mode**: work without being connected to the group. Very useful for portable devices.

2.How to connect an application to an UA:

The API:

LaCOLLA API is a bidirectional API, this means that the application can invoke methods from the API in order to get a service of LaCOLLA. In the other hand LaCOLLA notifies to applications different events that are happening in the group/s which we are connected to.

LaCOLLA provides different set of services as mentioned in the chapter before. Also the notifications of LaCOLLA are related to events happened in LaCOLLA due to many reasons. The main reasons are related to the activity of the rest of the members connected to the group of LaCOLLA whose we are connected.

LaCOLLA works in an asynchronous way, for this reason when ever an occurrence happens it is notified directly to the application by invoking an application defined method. LaCOLLA knows where and how many applications are connected.

The API is distributed in two parts. That architecture permits

1. **LaCOLLA Part:** Services offered by LaCOLLA to applications connected to an UA. The set of services that an application can ask to LaCOLLA. For instance *login, logout, disseminateEvent, sendInstantMessage, executeTask,...*
2. **Application Part:**Notification services that LaColla offers to the applications, these services are invoked by LaCOLLA, due to its asynchronous behaviour. The application developer must define the response of the application for each of that notifications. For instance *newConnectedMember, newEvent, notifyTaskException,...*

- **How does API work?**

The UA publishes its API into a RMIRegistry defined by LaCOLLA. The application must resolve the API location and get a remote instance of it.

Henceforth, the application can use all the public methods published in the API.

When an application starts,publishes its part of the API into a local RMIRegistry (it may be remote) in the host and the port specified by application's developer.

Every application must redefine the methods from the class ApplicationsSideApi. These methods are invoked by the UserAgent when LaCOLLA wants to notify something to an application. We must note the asynchronous execution of this methods.

LaCOLLA provides the interface ApplicationSideApi. That interface contains methods that the application must implement. During the development of the application the methods from the ApplicationSideApi interface must be redefined in a class named ApplicationSideApiImpl. That class must *implement* the methods from the class mentioned before. The implementation class must be set in the package of the application, inside a folder named API.

Example : package Apps.MessageServer.API

The application must implement a class that resolves the LaCOLLA API. In a class implemented by the application may be resolved(using standard RMI) the API remote object from LaCOLLA and invoked the desired methods from it. More information could be found in the documentation attached to LaCOLLA install version.

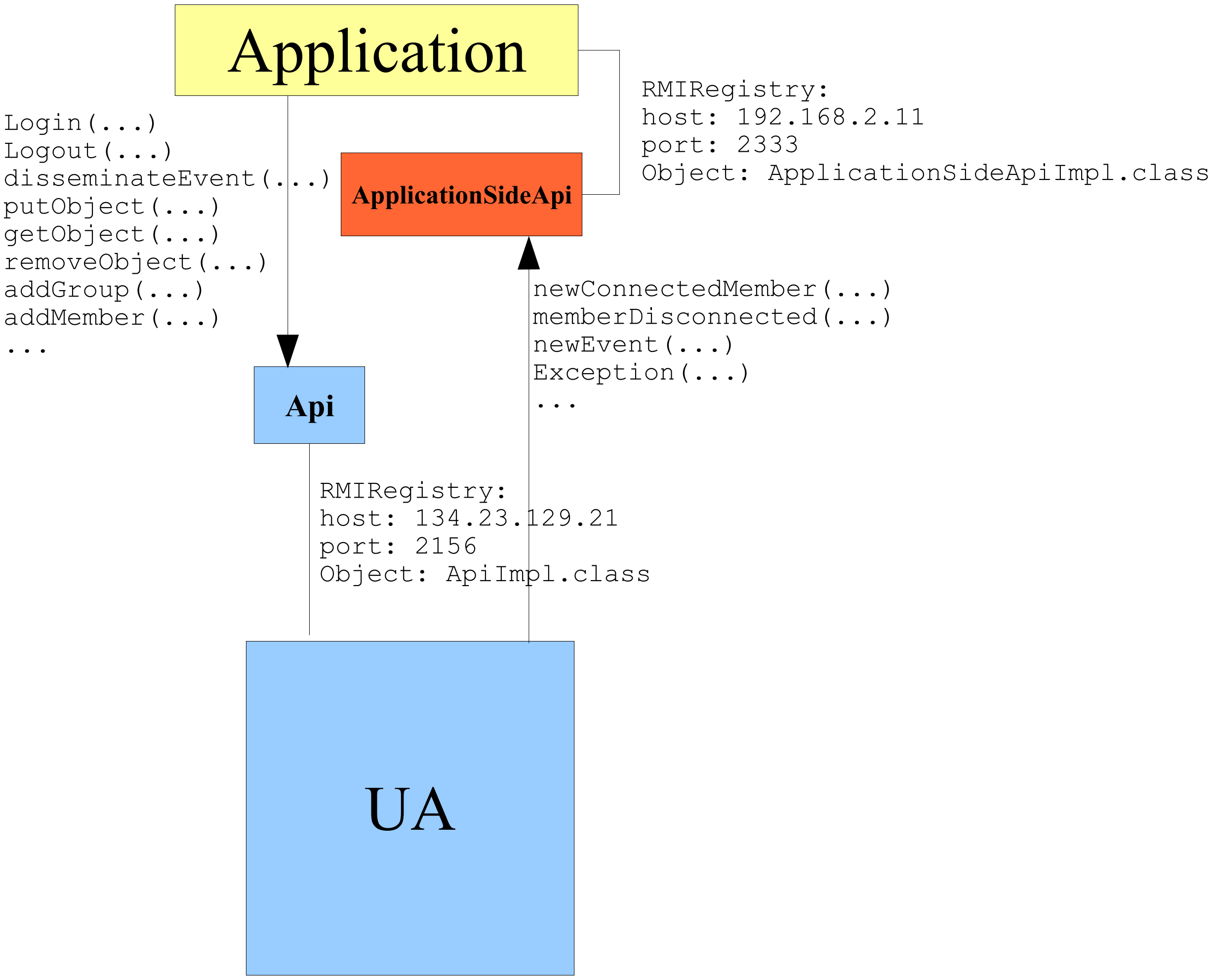

Figure 3. API design overview.

API

The functionalities provided by LaCOLLA API are described below:

Table A. API functions that User Agents offer to applications. Functions marked with** are not yet implemented.

Category	**Function**	**Description**
Presence	login	Connects user to group.
	logout	Disconnects user from group.
	whoIsConnected **	Which members are connected to the group?
Events	disseminateEvent	Sends an event to all applications belonging to group.
	eventsRelatedTo	Which events have occurred to a specific object?
Objects	putObject	Stores an object in LaCOLLA.
	getObject	Obtains an object stored into LaCOLLA.
	removeObject	Removes an object stored in LaCOLLA.
Tasks	submitTask	Submit a task to be executed by computational resources belonging to group.
	stopTask	Stops a task.
	getTaskState	In which state is the tasks?
Instant Messaging	sendInstantMessage	Sends a message to specified members of the group.

Category	Function	Description
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 asynchronously. (See the groupInfo function)
	getGroupInfoSync**	Gets information about the properties of a group in a synchronous manner. The 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 B. API functions that UA invokes on applications. Functions marked with ** are not yet implemented.

Category	Function	Description
Presence	newConnectedMember	Notifies that a new member has been connected.
	memberDisconnected	Notifies that a member has been disconnected.
Events	newEvent	Reception of an event occurred in the group.
Tasks	taskStopped	Notifies that the task has been stopped nicely.
	taskEnded	Notifies the ending of a task.
Instant Messaging	newInstantMessage	Reception of a new instant message.
Groups	groupInfo	Reception of the group information.
Other functions	exception**	Notifies that an internal exception or anomalous situation has occurred.
	appIsAlive	UA queries the state of the application. A boolean result must be returned.

The next two sections presents the use cases and operation contracts of LaCOLLA API respectively. As mentioned before, LaCOLLA API is divided in two parts, the API used by applications and provided by UserAgents an the API used by UserAgents and defined by applications. Each of the following sections is divided in that two parts.

3.Use cases of the API operations

Describes the events sequence and the actors of that sequence. Also describes the interaction within the actors and the API.

API used by Applications:

That part contains the methods provided by LaCOLLA API an susceptible to be used by applications.

1. Presence

1.1.Use case login

USE CASE	*Authentication process*
ACTORS	Application
PURPOSE	Authenticate and login a member
SUMMARY	The user tries to be authenticated in LaCOLLA. If the operation success,the user logins LaCOLLA. Receives its memberId. Henceforth, the user is allowed to use all the mechanisms provided by LaCOLLA API for this session.

Events sequence

Application	*LaCOLLA*
1. *login (String groupId,String userId, String pswd, String GapaId,String GapaAdress, int GapaPort,String aplicHost_,int aplicPort_)*	
	2. The user is authenticated. If the authentication process succeeds *newConnectedMember* notification is executed.
3.a response is received	

1.2.Use case logout

USE CASE	*Logout of an Application*
ACTORS	Application
PURPOSE	Logout of a connected user of LaCOLLA
SUMMARY	A user asks for the disconnection of the group which is connected.

Events sequence

Application	*LaCOLLA*
1. *logout (String groupId,String userId,String aplicName)*	
	2. Disconnects the user from the group. Notifies *memberDisconnected* to the rest of members of the group.

1.2.Use case whoIsConnected

USE CASE	*Who is connected to a Group*
ACTORS	Application
PURPOSE	Get the list of connected members
SUMMARY	The user asks for the list of connected members

Events sequence

Application	*LaCOLLA*
1. *whoIsConnected (String groupId)*	
	2. Gets a list with the memberId of connected all members to the specified group.
3. Receives a list with the identificators of the connected members.	

2. Events:

2.1.Use case disseminateEvent

USE CASE	*Disseminate an Event*
ACTORS	Application
PURPOSE	Notify to rest of the members of the group a new event.
SUMMARY	The user wants to send an event to the rest of the members of the group.

Events sequence

Application	*LaCOLLA*
1. disseminateEvent(String groupId,Event evt)	
	2. Store the event at least into one RA. Executes *newEvent* operation to deliver the new Event to the rest of the members of the group. The event will be delivered to disconnected members in the login process.

2.2.Use case eventsRelatedTo

USE CASE	*Events related to an Object*
ACTORS	Application
PURPOSE	Get all the events related to an object.
SUMMARY	The user wants to get all the events generated over an object.

Events sequence

Application	*LaCOLLA*
1. eventsRelatedTo(String groupId,String objectId)	
	2.Selects the set of events related with the specified object.
3.A list with all events is received.	

3. Objects:

3.1.Use case putObject

USE CASE	*Put Object*
ACTORS	Application
PURPOSE	Store an object permanently in LaCOLLA. The new object will be accessible by the rest of the members of the group.
SUMMARY	The user wants to store an object. The object is accessible by the rest of the members of the group.

Events sequence

Application	*LaCOLLA*
1. *putObject (ObjectLaCOLLA obj)*	
	2. Stores the object in a RA Delivers an event informing about the new Object. It uses *newEvent* operation.
3. a response is received.	

3.2.Use case getObject

USE CASE	*Get Object*
ACTORS	Application
PURPOSE	Get an object stored in LaCOLLA
SUMMARY	The user asks for a stored object in LaCOLLA

Events sequence

Application	*LaCOLLA*
1. *getObject (ObjectLaCOLLA obj)*	
	2. Gets the object referenced by the *objectId*
3. the object is received	

3.3.Use case getInfoObject

USE CASE	*Get Object Information*
ACTORS	Application
PURPOSE	Gets the information of the specified object.
SUMMARY	The user asks for the information of the stored object.

Events sequence

Application	*LaCOLLA*
1. *getInfoObject (String groupId,String objectId)*	
	2.Gets the information relative to the object
3.the object is received	

3.4.Use case removeObject

USE CASE	*Remove Object*
ACTORS	Application
PURPOSE	Deletes the object stored in LaCOLLA

USE CASE	*Remove Object*
SUMMARY	An user ask for the deletion of an object.

Events sequence

Application	*LaCOLLA*
1. *removeObject (String groupId, String objectId)*	
	2. An RA storing the object deletes the object an the references to it. An event notifying the deletion of the object is delivered. The rest of RA of the system delete the object at the event reception. The UA deletes all entries of the object in its summary when the event is received.
3. a response is received.	

4. Group Administration:

4.1.Use case addGroup

USE CASE	*Add Group*
ACTORS	Application
PURPOSE	Creates a new group and the user is connected to it.
SUMMARY	The user is connected to the group. The resources of the user become resources of the new group.

Events sequence

Application	*LaCOLLA*
1. *addGroup (String userId,GroupInfo groupInfo)*	
	2. Created the new group with the information contained in the GroupInfo.
3.The identifier of the new group is received	

4.2.Use case getInfoGroup

USE CASE	*Ask for the information of the group*
ACTORS	Application

USE CASE	*Ask for the information of the group*
PURPOSE	Ask for the information of the group
SUMMARY	The user asks for the information of the group.

Events sequence

Application	*LaCOLLA*
1. *getInfoGroup (String userId,String groupId,String aplicId)*	
	2. Ask for the information of the group. The information is delivered to the application by newInfoGroup.
3. The information about the group is received	

5. Member Administration:

5.1.Use case addMember

USE CASE	*Add Member*
ACTORS	Application
PURPOSE	Adds a member to the group
SUMMARY	The user asks for the new member addition.

Events sequence

Application	*LaCOLLA*
1. addMember (String memberId, String groupId, Object memberInfo, String role,String username, String password, String emailAddress)	
	Adds a new member to the specified group.
3. The notification of the new member is delivered.	

5.2.Use case getInfoMember

USE CASE	*Get Member Information*
ACTORS	Application
PURPOSE	Get the information related to a member
SUMMARY	The user asks for the information of a member

Events sequence

Application	*LaCOLLA*
1. getInfoMember(String memberId,String groupId)	
	2.gets the information of a member
3.the information is received	

6. Tasks

6.1.Use case submitTask

USE CASE	*Submit Task*
ACTORS	Application
PURPOSE	Sends task to execute
SUMMARY	The user sends a task to execute

Events sequence

Application	*LaCOLLA*
1. submitTask (byte[] xml,,String groupId)	
	2.Sends the task described in **byte[] xml** to be executed.
3.The task identifier is received	

6.2.Use case stopTask

Use case	*Stop Task*
ACTORS	Application
PURPOSE	Stops an executing task.
SUMMARY	The user stops a task being executed.

Events sequence

Application	*LaCOLLA*
1. stopTask (String idTask,,String groupId)	
	2.The task wit identifier *idTask* is stopped
3.The notification of stopped task is received.	

6.3.Use case getTaskState

USE CASE	*Get Task State*
ACTORS	Application
PURPOSE	Get the task state

USE CASE	*Get Task State*
SUMMARY	The user asks for the current task state

Events sequence

Application	*LaCOLLA*
1. getTaskState(String idTask,String groupId)	
	2.ask for the task *idTask* state.
3.The notification with the state of the task is received.	

7. Instant Message Service:

7.1.Use case sendInstantMessage

USE CASE	*Send Instant Message*
ACTORS	Application
PURPOSE	Sends an instant message to a list of members
SUMMARY	The user sends a message to a list of members of the group

Events sequence

Application	*LaCOLLA*
1. sendInstantMessage (String memberId,String groupId,Object message,ArrayList targetList)	
	2.The system gets all the sites where the members of the list are connected and sends an instant message to each of them.
3.The notification of message send is received	

API used by UserAgent:

That part contains the methods provided by the application API an susceptible to be used by UserAgent.

1. Presence

1.1.Use case newConnectedMember

USE CASE	*New Connected Member Notification*
ACTORS	UserAgent and applications
PURPOSE	Notify to all connected members the connection of a member.
SUMMARY	The UA communicates to the connected applications the connection of the member.

Events sequence

Application	*LaCOLLA*
	1. *newConnectedMember(String groupId,String userId,String memberId)*
2. The application receives the notification.	

1.2.Use case memberDisconnected

USE CASE	*Member disconnected notification*
ACTORS	UserAgent and applications
PURPOSE	Notify to all connected applications the logout of a member.
SUMMARY	The user agent notifies to the application the disconnection of a member.

Events sequence

Application	*LaCOLLA*
	1. *memberDisconnected(String groupId,String userId)*
2. The application receives the notification.	

2. Events:

2.1.Use case newEvent

USE CASE	*New Event Notification*
ACTORS	UserAgent and applications.
PURPOSE	Notify to the connected applications the event received.
SUMMARY	The UserAgent notifies to connected applications the event.

Events sequence

Application	*LaCOLLA*
	1. *newEvent(String groupId,Event evt)*
2. The application receives the event	

3. Group Administration:

3.1.Use case newInfoGroup

USE CASE	*Notificació d'Informació de Grup*
ACTORS	UserAgent and applications
PURPOSE	Notifies to application the information of the group.
SUMMARY	The applications receives the requested group information.

Events sequence

Application	*LaCOLLA*
	1. *newInfoGroup(String userId, String groupId, String aplicId, GroupInfo info)*
2. The application receives the requested group information.	

4.Tasks:

4.1.Use case Exception

USE CASE	*Exception*
ACTORS	UserAgent and applications
PURPOSE	Notify the exception of a task being executed.
SUMMARY	The application receives the task exception

Events sequence

Application	*LaCOLLA*
	1. *exception(String groupId,String Message)*
2.The application receives the exception	

4.2.Use case notifyStopTask

USE CASE	*Task Stop Notification*
ACTORS	UserAgent and applications
PURPOSE	Notify the task stop.
SUMMARY	The application receives the stop task notification.

Events sequence

Application	*LaCOLLA*
	1. notifyStopTask*(String groupId,String idTask, Object result)*
2. The application receives the notification.	

4.3.Use case notifyTaskState

USE CASE	*Task State Notification*
ACTORS	UserAgent and applications
PURPOSE	Notify the task state
SUMMARY	The application receives the task state

Events sequence

Application	*LaCOLLA*
	1. notifyTaskState*(String groupId,String idTask,String state,Object result)*
2. The application receives the notification.	

5. Applications state control:

5.1.Use case AppIsAlive

USE CASE	*Is application alive*
ACTORS	UserAgent and applications
PURPOSE	Ask to the application its current state.
SUMMARY	The UserAgent asks to the application its current state. The application responds with true is connected.

Events sequence

Application	*LaCOLLA*
	1. *appIsAlive(String appID)*
2.The application answers true.	

6. Instant Message Service:

6.1.Use case newInstantMessage

USE CASE	*Instant MessageNotification*
ACTORS	UserAgent and applications

USE CASE	***Instant MessageNotification***
PURPOSE	Notify to applications the reception of an instant message.
SUMMARY	The application receives an instant message.

Events sequence

Application	***LaCOLLA***
	1. *newInstantMessage(String groupId, String userId, String destMemberId, Object message)*
2.The application receives the message.	

4.API operation contracts:

Operation contracts describes the operations effects. The operation outputs and the states of the information during the execution are also described. Insures the software reliability with preconditions and postconditions.

API used by Applications:

That part contains the methods provided by LaCOLLA API an susceptible to be used by applications.

1. Presence

- **Authentication (login)**

1.1.Contract login

OPERATION	**Login**
PARAMETERS	String groupId String userId String pswd String gapaId String gapaAdress int gapaPort String aplicHost int aplicPort
SEMANTICS	Identification process of an user in order to join the group.
PRECONDITIONS	All parameters are not null.
POSTCONDITIONS	The user is authenticated in the system. The resources of the user are added to the resources of the group. The application receives the application identifier. Henceforth the user could use LaCOLLA services.
RESULT	Returns the application identifier.

- **Disconnection(logout)**

1.2.Contract logout

OPERACIÓ	**Logout**
PARÀMETRES	String groupId String userId String aplicName
SEMÀNTICA	Disconnects a member from the group.
PRECONDICIONS	AplicName is the application identifier. All parameters are not null.

POSTCONDICIONS	The member is disconnected. Only can execute the login operation.
SORTIDA	No result.

- **List of connected members to the group(whoIsConnected)**

1.2.Contract logout

OPERACIÓ	**whoIsConnected**
PARÀMETRES	String groupId
SEMÀNTICA	Gets the list of connected members
PRECONDICIONS	GroupId is not null
POSTCONDICIONS	The member receives a list with the identifiers of all connected members
SORTIDA	List

2. Events.

- **Disseminate Event. (disseminateEvent)**

2.1.Contract disseminateEvent

OPERATION	**DisseminateEvent**
PARAMETERS	String groupId Event evt
SEMANTICS	An event is sent to all members of the group. In the case that a member is not connected during the message dissemination, the event will be delivered during the login process of that member.
PRECONDITIONS	All the parameters are not null. Event is the data structure defined by LaCOLLA API and contains information about the event to be disseminated.
POSTCONDITIONS	If the event is not stored in at least one RA it is not considered as being disseminated. In that case LaCOLLA retries to send the event a configurable number of times. If it not success an error message is returned. The non-connected member will receive the event during the connection process. If any component connected to the group don't receives the event. Eventually the event will be received during a future synchronization or consistency session with one RA. The events are stored permanently in any RA.

RESULT	No return

- **Events Related to an object (eventsRelatedTo)**

2.2.Contract eventsRelatedTo

OPERATION	**eventsRelatedTo**
PARAMETERS	String groupId String objectId
SEMANTICS	Lists all events related to an object.
PRECONDITIONS	The object and the group must exists.
POSTCONDITIONS	A list with all events related to an object is created.
RESULT	Return the list with all events related to an object.

3. **Objects:**

- **Put Object (putObject)**

3.1.Contract putObject

OPERATION	**putObject**
PARAMETERS	ObjectLaCOLLA obj
SEMANTICS	Store the object permanently in a repository of LaCOLLA. The object will be available for the rest of the members of the group. The object will be replicated among the rest of repositories.
PRECONDITIONS	The parameters are not null.
POSTCONDITIONS	If the event is not stored in at least one RA, the object is not stored. In t hat case the object must be retransmitted. Once the object is stored the repository disseminates an event notifying the new object. The object is unique identified in the system. The identifier is returned to the application.
RESULT	The object descriptor is returned to the application. It contains the objectId.

- **Get Object (getObject)**

3.2.Contract getObject

OPERATION	**getObject**
PARAMETERS	ObjectLaCOLLA obj
SEMANTICS	Get the stored object.

PRECONDITIONS	The parameters are not null.
POSTCONDITIONS	The application receives the object.
RESULT	An input stream is received. The application must read the binary object from that stream.

- **Get the information of the object (getInfoObject)**

3.3.Contract getInfoObject

OPERATION	**getInfoObject**
PARAMETERS	String groupId String objectId
SEMANTICS	Get the information relative to an object.
PRECONDITIONS	The parameters are not null.
POSTCONDITIONS	The application receives the required object.
RESULT	An input stream is received. The application must read the binary object from that stream.

3.4.Contract removeObject

OPERATION	**removeObject**
PARAMETERS	String groupId String objectId
SEMANTICS	Remove a permanently stored object in LaCOLLA
PRECONDITIONS	The parameters are not null.
POSTCONDITIONS	If the operation is not carried out in at least one RA and an event generating the deletion of the object is not generated, the operation is not considered. The repository sends an event informing about the objects deletion. The repository itself, removes any reference of the object. The rest of component delete the object when receive the event.
RESULT	No result.

4. Group Administration:

- **Add Group (addGroup)**

4.1.Contract addGroup

OPERATION	**AddGroup**
PARAMETERS	String userId GroupInfo groupInfo

SEMANTICS	Creation of a new group. Authentication of the member to the new group.
PRECONDITIONS	The parameters are not null. groupInfo contains information of the group.
POSTCONDITIONS	A new group is created with an unique identifier and the resources of the group creator. 1. The creator is authenticated into the group.
RESULT	Returns the groupId.

- **Get Group Information (getInfoGroup)**

4.2.Contract getInfoGroup

OPERATION	**getInfoGroup**
PARAMETERS	String userId String groupId String aplicId
SEMANTICS	Request the information relative to a group.
PRECONDITIONS	The parameters are not null. groupInfo contains information of the group.
POSTCONDITIONS	The information of the group is notified asynchronously by **newInfoGroup** operation.
RESULT	If the UserAgent has the information cached, it is returned immediately.

5. Members Administration:

- **Add Member (addMember)**

5.1.Contract addMember

OPERATION	**addMember**
PARAMETERS	String memberId String groupId Object memberInfo String role String username String password String emailAddress
SEMANTICS	A member is invited to a group.
PRECONDITIONS	UserId must belong to the group.
POSTCONDITIONS	The group has one more member.

RESULT	Returns the new memberId

- **Get Member Information (getInfoMember)**

5.4.Contract getInfoMember

OPERATION	**getInfoMember**
PARAMETERS	String memberId String groupId
SEMANTICS	Request information about a member.
PRECONDITIONS	The parameters are not null.
POSTCONDITIONS	A message to a GAPA is sent requesting the desired information. The GAPA sends the identifier of the object containing the information. Then, the object is requested to a RA and sent to the application.
RESULT	No result.

6. Tasks:

- **SubmitTask:**

5.1.Contract submitTask

OPERATION	**submitTask**
PARAMETERS	**byte**[] xml, String groupId
SEMANTICS	A task is sent to execute.
PRECONDITIONS	The group must exist
POSTCONDITIONS	A new task is being execute in LaCOLLA
RESULT	Returns the task identifier.

- **StopTask:**

5.2.Contract stopTask

OPERATION	**submitTask**
PARAMETERS	String idTask String groupId
SEMANTICS	Stop the task.
PRECONDITIONS	The group must exist.
POSTCONDITIONS	The task is stoped
RESULT	No result.

- **getTaskState:**

5.3.Contract getTaskState

OPERATION	getTaskState
PARAMETERS	String idTask String groupId
SEMANTICS	Request the state of a task.
PRECONDITIONS	The group must exist.
POSTCONDITIONS	A TDA is requested about the state of a task
RESULT	No result.

7. **Instant Message Service:**

- **Send an Instant Message (sendInstantMessage)**

7.1.Contract sendInstantMessage

OPERATION	sendInstantMessage
PARAMETERS	String memberId String groupId Object message ArrayList targetList
SEMANTICS	Send an instant message to a list of members of the group.
PRECONDITIONS	The parameters are not null. Message is the message to be sent. TargetList is the destination members list.
POSTCONDITIONS	The members of the targetList receive the message.
RESULT	No result.

API used by UserAgent:

That part contains the methods provided by the application API and UserAgent will use them to notify/deliver information to the application.

1. **Presence:**

- **New Connected Member Notification. (newConnectedMember)**

1.1.Contract newConnectedMember

OPERATION	newConnectedMember

PARAMETERS	String groupId String userId String memberId
SEMANTICS	New connected member notification.
PRECONDITIONS	Both parameters are not null.
POSTCONDITIONS	A new member is connected to the group.
RESULT	No result.

- **Member Disconnected Notificationt (memberDisconnected)**

1.2.Contract memberDisconnected

OPERATION	**memberDisconnected**
PARAMETERS	String groupId String memberId
SEMANTICS	Notify to the application the disconnection of a member.
PRECONDITIONS	Both parameters are not null.
POSTCONDITIONS	The member is not connected to the group.
RESULT	No result.

2. Events:

- **New Event Notification (newEvent)**

2.1.Contract newEvent

OPERATION	**newEvent**
PARAMETERS	String groupId Event event
SEMANTICS	Notify to the application a new event.
PRECONDITIONS	Both parameters are not null. Event is a data structure provided by LaCOLLA.
POSTCONDITIONS	The application receives the event.
RESULT	No result.

3. Group Administration:

- **Group Information Notification(newInfoGroup)**

3.1.Contract newInfoGroup

OPERATION	**newInfoGroup**

PARAMETERS	String userId String groupId String aplicId GroupInfo info
SEMANTICS	Notifies to the application the group informationBoth parameters are not null.
PRECONDITIONS	The parameters are not null. A getInfoGroup operation has been invoked.
POSTCONDITIONS	The application receives the group information.
RESULT	No result.

4. Tasks:

- **Exception**

4.1.Contract exception

OPERATION	**exception**
PARAMETERS	String groupId String message
SEMANTICS	Notification of a task exception.
PRECONDITIONS	Both parameters are not null.
POSTCONDITIONS	The application receives the exception.
RESULT	No result.

- **Task Stop Notification**

4.2.Contract notifyStopTask

OPERATION	**notifyStopTask**
PARAMETERS	String groupId String idTask String result
SEMANTICS	Notifies the task stop.
PRECONDITIONS	The parameters are not null.
POSTCONDITIONS	The application receives the task result.
RESULT	No result.

- **Task State Notification**

4.3.Contract notifyTaskState

OPERATION	**notifyTaskState**

PARAMETERS	String groupId String idTask String result String State
SEMANTICS	Notifies the task state.
PRECONDITIONS	The parameters are not null.
POSTCONDITIONS	The application receives the task state.
RESULT	No result.

5. Applications state control:

- **State request service (*appIsAlive*):**

5.1.Contract appIsAlive

OPERATION	**appIsAlive**
PARAMETERS	String appId
SEMANTICS	LaCOLLA request the application state.
PRECONDITIONS	The application exists.
POSTCONDITIONS	The application remains connected if the result is true.
RESULT	Boolean.

6. Instant Message Service:

- **Instant message reception (newInstantMessage):**

6.1.Contract newInstantMessage

OPERATION	**newInstantMessage**
PARAMETERS	String groupId String userId String destMemberId Object message
SEMANTICS	Reception of an instant message.
PRECONDITIONS	The parameters are not null. Message contains the message text.
POSTCONDITIONS	The application receives the instant message.
RESULT	No result.

5.LaCOLLA data structures description:

That section describes the most important data structures that applications may use when use LaCOLLA. For each of the data structures, a description of the main attributes, a constructor description and a detailed methods description is provided.

Description of the objects used by LaCOLLA:

Each object is described in LaCOLLA by its corresponding descriptor. The descriptor is the ObjectLaCOLLA object. The ObjectLaCOLLA contains information about the object to be stored, recovered, removed,..... Each application must describe the objects to store in LaCOLLA using the provided descriptor.

Each descriptor contains a reference to a local file containing the "real" information to be stored. For instance, if we want to store a mp3 file, it is necessary to indicate in the descriptor the local file containing such information. Moreover another file must be set in the descriptor. The infoObject file contains relevant information of the real object. The infoObject is stored together with the object.

Attributes	
`String objId`	Unique object identifier of the object. It is generated by LaCOLLA when the object is stored in the system. Is returned inside the ObjectLaCOLLA returned by the operation *putObject*
`Date date`	The creation date of the object.
`String description_`	Any user defined description.
`long sz`	The size in bytes of the object to be stored/recovered.
`String grpId`	The group identifier where the object must be stored/deleted.
`File file`	The local file containing the object to be stored.
`File fileInfoObject`	The local file containing the information of the object to be stored.
`String versionId`	The version of the object

Constructor
`ObjectLaCOLLA(String objId, Date date, String description_, String grpId, long sz)`
`ObjectLaCOLLA(String objId, Date date, String description_, String grpId, long sz, String path_)`
`ObjectLaCOLLA(String objId, String obj, long sz)`

Methods		
Date	getCreationDate()	Returns the creation date of the object.
String	getDescription()	Sets the description of the object
File	getFile()	Return the file containing the object to be stored.
String	getFileName()	Returns the filename
String	getGroupId()	Returns the groupId of the object
Object	getInfoObject()	Returns the file containing information about the object
String	getObjectId()	Returns the objectId.
long	getSize()	Returns the object size
String	getVersionId()	Returns the versionId of the object.
void	setCreationDate(Date creationDate_)	Sets the creation date.
void	setDescription(String desc)	Sets the object description
void	setFile(File ffile_)	Sets the file.
void	setGroupId(String groupId)	Sets the groupId.
void	setInfoObject(Object infoObject)	Sets the file containing the object's information.
void	setObjectId(String objId)	Sets the objectId
void	setSize(long size_)	Sets the object's size
void	setVersionId(String versionId)	Sets the versionId of the object.

Example:

The next example presents the usage of the ObjectLaCOLLA structure. In the example we consider that the information to be stored permanently in LaCOLLA is in a file in the local disc named "*foo.dat"*. Also we consider the existence of a file named *"foo_infoObject.dat"* containing some information of the object to be stored.

Eventually the ObjectLaCOLLA is created. The parameters are the file name (in this example *foo.dat*), the current time, the textual description of the object to be stored, the group identifier where the object must be stored and finally the file length.

At the end, the file and the *infoObject* file are set and the *putObject* operation is invoked. The result of that operation is the same *ObjectLaCOLLA* containing the *objectId* generated by LaCOLLA.

The *putObject* operation is invoked to store the object. The object descriptor is returned.

The next part of the example presents the way to get an object stored in LaCOLLA. The descriptor of the object to be got could be obtained either as a result of a *putObject* operation or by the specific *objectLaCOLLA* creation. (in that case we must note that the *objectId* is required and must be set in the *OjectLaCOLLA*).

The *getObject* operation returns an *InetSocketAddres*. Using that *InetSocketAddress* LaCOLLA sends the bytes of the object to the application. The purpose of the class *Receiver* (provided in LaCOLLA distribution) is to read the object and to store it in the local disk. The parameters are the object descriptor, the *InetSocketAddres*, the folder where the object must be placed and the file name where the data must be set.

```
File file=new File("foo.dat");
File fileInfoObject= new File("foo_infoObject.dat");
ObjectLaCOLLA obj= new ObjectLaCOLLA(file.getName(),
                 Calendar.getInstance().getTime(),
                 "foo description", groupId, file.length());
obj.setFile(file);
obj.setInfoObject(fileInfoObject);
ObjectLaCOLLA o = api.putObject(obj);


//After a few seconds the object is stored.
//it is possible to get the object.
InetSocketAddress isa =(InetSocketAddress)api.getObject(o);

System.out.println("The received ISA: " + isa);

if (isa!=null){
      Receiver rec=new Receiver(o,isa,"home_directory","foo-received.dat");
      rec.start();
      System.out.println("Object received.”);
}
else{
      System.out.println("isa is null- the object cannot be obtained – Please
                          retry later");
}//else

...
```

Example 1. ObjectLaCOLLA

Description of the events used by LaCOLLA:

An event is something that takes place; an occurrence, something that an user wants to communicate to the rest of community.

Whenever something occurs in LaCOLLA an event is disseminated in order to communicate to the listeners what happened. The event itself is handled by the application and consequently treated.

Usually a member wants to notify something to the rest of the users in the group. For that purpose an event can be disseminated.

The events disseminated by LaCOLLA are described as follows:

Attributes	
`String applicationId`	The application identifier. It is returned in the login operation.Only it is set if the event is generated by an application.
`String userId,`	The user identifier. It is returned when the new connected member notification is received.
`String groupId,`	The groupId where the event must be disseminated.
`GroupInfo infoGroup`	The group information object. Only accessible when the event received notifies the new group information.
`String memberInfo`	The objectId of the object containing the member information.
`String objectId`	Only used in the case that the event is related to an object.
`String event,`	The event itself. The text disseminated.
`String componentId`	The component who has generated the event. Only in the case that the event has been generated by a component.
`Int eventType`	The type of the event. See description below.

Constructor	
`Event()`	

`Event(String userId,`
`String applicationId, String groupId,`
`String eventId, Timestamp timestamp,`
`String objectId, int eventType,`
`String event, String componentId)`	

Methods		
String	**getApplicationId**()	Returns the application identifier.
String	**getComponentId**()	Returns the source component identifier.
String	**getEvent**()	Returns the text of the event
int	**getEventType**()	Returns the type of the event
String	**getEventId**()	Returns the event Id.
String	**getGroupId**()	Returns the GroupId.
GroupInfo	**getGroupInfo**()	Returns the GroupInfo. Only if the event notifies the new group information.
String	**getMemberInfo**()	Returns the identifier of the object containing the member information.
String	**getObjectId**()	Returns the objectId related to the event.
String	**getUserId**()	Return the user who has generated the event.
void	**setApplicationId**(String applicationId)	Sets the application Id who has generated the event.
void	**setComponentId**(String componentId)	Sets the component Id who has generated the event
void	**setEvent**(String event)	Sets the text to disseminate
void	**setEventType**(int eventType)	Sets the type of the event
void	**setEventId**(String eventId)	Sets the event identifier
void	**setGroupId**(String groupId)	Sets the group where the event has to be disseminated.
void	**setGroupInfo**(GroupInfo groupInfo)	Sets the group information.
void	**setMemberInfo**(String memberInfo)	Sets the member information

void	**setObjectId**(String objectId)	Sets the object related to the event.
void	**setUserId**(String userId)	Sets the memberID of the sender of the event.

Example:

In the following example we invoke the login operation of LaCOLLA. As a result of this operation we obtain the application identifier of our application. Furthermore we set our userId as “*foo*”. Eventually the Event structure is created. The parameters are, the user identifier, the application identifier, the group identifier, the event identifier, and the text of the event. The rest of parameters must be *null*.

Note that the event identifier is generated by invoking the *generateID* operation of the *Identificator* class provided by LaCOLLA distribution.

```
Sting  aplicIdentifier=api.login(.....)
String userId=new String(”foo”);
Event evt = new Event(userId,aplicIdentifier,groupId,
      (String)Identificator.generateID("EVENT",""),
      null,null,null,"The event text to be set",null);
```

Example 2. Event

Event classification:

Events are classified as follow:

	Type of Event	***Description***
Modify state events	eventNewObject	This event is received when a new object is stored in LaCOLLA.
	eventNewReplica	This event is received when a new replica of an stored object is created.
	eventDeleteObject	This event is received when an object is deleted from LaCOLLA.
	eventDeleteReplica	This event is received when replica of an object is deleted from LaCOLLA.
	eventNewMember	This event is received when a new Member is invited to the group.

	Type of Event	*Description*
Informative events	eventRead	This event informs that someone has read an object in LaCOLLA.
	eventNewConnectedMember	This event informs about a new connected member.
	eventMemberDisconnected	This event informs about a member has disconnected.
	eventApplication	This type of event is used by applications. An application would need to disseminate an event to the group informing about whatever has happened. This type of event is never used by LaCOLLA, so if an application receives that kind of event, it has been generated by another application connected to LaCOLLA.

Events types can be accessed by the Api function events.getEventType(). The types of events are defined i the class LaColla.core.util.constant.

The type of the event is always set by LaCOLLA. Whether the event is an application event or not, the task of classifying the event is done by the UA, hence the application will never need to classify the event but may need to read the type of the event.

Description of the GroupInfo structure used by LaCOLLA:

This structure is used to describe information about a group. The creator of the group must specify that information at creation time. When the addGroup operation is called, the group information must be provided.

Attributes	
`String groupName`	The group name.
`String member`	The creator of the group.
`Date fundationDate`	The foundation date.

Constructor
`GroupInfo``()`
`GroupInfo``(String groupName, String member, Date fundationDate)`

Methods

Date	**getFundationDate**()	Returns the creation date of the group
String	**getGroupName**()	Returns the name of the group
ArrayList	**getMembers**()	Returns a list of the members of the group
void	**setFundationDate**(Date fundationDate)	Sets the fundation date of the group.
void	**setGroupName**(String groupName)	Sets the name of the group.
void	**setMembers**(ArrayList members)	Sets the list of members of the group.

Example:

In the next example we create a GroupInfo structure. We set the current date as a creation date and we set the name of the group, in that case, "*foo-group*". Finally we invoke the *addGroup* operation of LaCOLLA API with our memberId and the information of the group to be created. The new groupId is returned.

```
GroupInfo gi=new GroupInfo();
gi.setFundationDate(Calendar.getInstance().getTime());
gi.setGroupName("foo-group");
String groupId=api.addGroup("member#dd7e5490bc0810048ef186aa17efe6e6#",gi);
```

Example 3. GroupInfo

6.API usage example.

This example introduces to the developer the best way of starting the construction of an application using the API of LaCOLLA. In that example, it is explained how to connect our application with LaCOLLA API, also it shows how to get the API of LaCOLLA in our application and how to invoke the methods provided by them. Eventually the example presents a way to implement the ApplicationsSideApi an to make it available for the local LaCOLLA agent.

1.Setup in the *classpath* of the project LaCOLLA *.jar files*.

2.Create a *package* for the application.
For example: **package** Apps.pasApas;

3.Create a subdirectory *API* in the *package* of our application.
For example: **package** Apps.pasApas.API;

4.Create the class *ApplicationsSideApiImpl* inside the package API

```
package Apps.pasApas.Api;

import java.rmi.RemoteException;

import LaColla.Api.ApplicationsSideApi;


public class ApplicationsSideApiImpl
            extends java.rmi.server.UnicastRemoteObject
                  implements ApplicationsSideApi{

      /**
       * @throws RemoteException
       */
      protected ApplicationsSideApiImpl() throws RemoteException {
            super();

      }
}
```

Example 4. ApplicationsSideApi

5.Implement the methods of the *interface ApplicationSideApi*.

```
      public void newConnectedMember(String groupId, String userId)
                                    throws RemoteException
      {
            //definir el comportament desitjat del mètode
            System.out.println("El nou membre connectat al grup
                                    "+groupId+ " és: " + userId);
      }

      public void memberDisconnected(String groupId, String userId)
                                    throws RemoteException
      {
            //definir el comportament desitjat del mètode
      }

      public void newEvent(String groupId, Event evt)
                                    throws RemoteException
      {
            //definir el comportament desitjat del mètode
      }
```

Example 5. Methods redefinition

6.Create a new class for the application.

```
package Apps.pasApas;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

import LaColla.Api.Api;

public class pasApas {

      //constructor
      public pasApas(){

      }
}
```

Example 6. Applications Class

7.Create a method to resolve LaCOLLA API.

```
        //constructor
        public pasApas(){
                //...
        }
        //API LaCOLLA resolve method
        public API resolveApiLaCOLLA(String host, long port){
                API api=null;
                try {

                        api = (API)Naming.lookup("//"+host+":"+port+"/API");

                }catch (MalformedURLException murle) {
                        System.out.println("MalformedURLException: " + murle);
                }
                catch (RemoteException re) {
                        System.out.println("RemoteException: " + re);
                }
                catch (NotBoundException nbe) {
                        System.out.println("NotBoundException: " + nbe);
                }

        return api;
        }
}
```

Example 7. API resolve methodology

8. Create a method to publish the *ApplictionsSideApi* redefined on steps 6 and 7.

```java
package Apps.pasApas;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import Apps.pasApas.API.ApplicationsSideApiImpl;
import LaColla.Api.Api;

public class pasApas {

      //constructor
      public pasApas(){
            //...
      }

      //API LaCOLLA resolve method
      public API resolveApiLaCOLLA(String host, long port){
            //...
      }


      public ApplicationsSideApiImpl
            bindApplicationsSideApi(String host, int port){

            ApplicationsSideApiImpl aplicapi=null;

            try {
                  java.rmi.registry.LocateRegistry.createRegistry(port);
                  aplicapi = new ApplicationsSideApiImpl();
                  Registry registry =
                              LocateRegistry.getRegistry(host,port);
                  registry.rebind("/AplicationsSideApi", aplicapi);

            } catch (Exception e) {
                  e.printStackTrace();
            }

      return aplicapi;
      }
```

Example 8. ApplicationsSideApi binding

9.API method invocation example.

Taula . Init method

```
package Apps.pasApas;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import Apps.pasApas.API.ApplicationsSideApiImpl;
import LaColla.Api.Api;

public class pasApas {

      //constructor
      public pasApas(){
            //...
      }

      //API LaCOLLA resolve method
      public Api resolveApiLaCOLLA(String host, long port){
            //...
      }


      public ApplicationsSideApiImpl
            bindApplicationsSideApi(String host, int port){
            //...
      }

      public void init(String host,String localHost,
                                   int port, int localPort){

            ApplicationsSideApiImpl localApi;
            Api LaCOLLAApi;
            String appId;
            //fem bind de l'ApplicationsSideApi
            localApi=this.bindApplicationsSideApi(localHost,localPort);
            //resolem l'API de LaCOLLA
            LaCOLLAApi=this.resolveApiLaCOLLA(host,port);

            //podem invocar algun mètode de l'API
            //LOGIN
            appId=LaCOLLAApi.login(
                  groupId,userId,password,gapaId,gapaHost,
                                   gapaPort,localHost,localPort);
            //...

      }
}
```

Example 9. Init method

10.Compile the code

11.Execute:
rmic.exe Apps.pasApas.Api.ApplicationsSideApiImpl.

12.Execute the new application and the configuration of LaCOLLA.

- Create an execution file.

For example: pasApas.bat

- Edit pasApas.bat

  make copy&paste from uoc1.bat created on the step 2.
  Modify the last line of the file.
  Write--> java Apps.pasApas.pasApas #params

  Where #params are the application parameters.

Also you would find a set of templates in "C:\LaCOLLA Middleware\LaCOLLA\doc\templates"