# CORODS: A Coordination Programming System for Open Distributed Systems

**Juan Carlos Cruz**

**Institut für Informatik (IAM)**
**Universität Bern**

*Neubrückstrasse 10*
*3012 Berne, Switzerland*
*cruz@iam.unibe.ch*

ABSTRACT. *Open Distributed Systems are the dominating intellectual issue of the research in distributed systems. Figuring out how to build and maintain those systems becomes a central issue in distributed systems research today. Although CORBA seems to provide all the necessary support to the construction of those systems, CORBA provides a very limited support to the evolution of their requirements. The main problem is that the description of the elements from which the systems are built, and the way in which they are composed are mixed into the application code, making systems difficult to understand, modify and customize. A possible solution to this problem goes through the introduction of the so-called coordination models and languages into the CORBA model. We propose in this paper a coordination programming system called CORODS which introduces the CoLaSD coordination model and language into the CORBA model. CoLaSD is a coordination model based on the notion of Coordination Groups, entities that specify, control and enforces the coordination of groups of collaborating active objects.*

RÉSUMÉ. *La construction des systèmes reparties ouverts est aujourd'hui un sujet majeur de recherche. Bienque CORBA semble offrir tout le support necessaire à la construction des tels systèmes, CORBA offre très peu de support pour l'évolution des ces systèmes. Le principal problème est que la description des éléments qui composent ces applications et la manière dont ils sont composés se trouvent mélangés, rendant difficile sa compréhension, modification et particularisation. Une possible solution passe par l'introduction de langages et modèles de coordination dans le modèle CORBA. Nous proposons dans ce papier un système de coordination appelé CORODS qui introduit un modèle et langage de coordination CoLaSD dans le model CORBA.*

KEY WORDS: *Open Distributed Systems, Coordination, CORBA, CoLaS*

MOTS-CLÉS :*Systèmes reparties ouverts, Coordination, CORBA, CoLaS*

## 1. Introduction

Software development of distributed systems has changed significantly over the last two decades. This change has been motivated by the goal of producing Open Distributed Systems (ODS in the following) [CRO 96]. ODSs are systems made of components that may be obtained from a number of different sources which together work as a single distributed system. ODSs are basically ”open” in terms of their topology, platform and evolution: they run on networks which are continuously changing and expanding, they are built on top of a heterogeneous platform of hardware and software pieces, and their requirements are continuously evolving. Evolution is the most difficult requirement to meet, since no all the application requirements can be known in advance. ODSs are the dominating intellectual issue of the research in distributed systems. Figuring out how to build and to maintain those systems becomes a central issue in distributed systems research today.

In 1988 the International Standards Organization (ISO) began a work on preparing standards for Open Distributed Processing (ODP). These standards have now been completed, and define the interfaces and protocols to be used in the various components of an ODS. The ODP standards provide the framework within which ODSs may be built and executed. One of the most popular (if not the most) specifications for some parts of the ODP is the Common Object Request Broker Architecture (CORBA) [OMG 95]. The CORBA middleware proposed by the Object Management Group (OMG) provides a standard for interoperability between independently developed components across networks of computers. Details such as the language in which components are written or the operating system in which they run is transparent to their clients. The OMG focused on distributed objects as a vehicle for system integration. The key benefit of building distributed systems with objects is encapsulation: data and state are only available through invocation of a set of defined operations. Object encapsulation makes system integration and evolution easier: differences in data representation are hidden inside objects, and new objects can be introduced or replaced in a system without affecting other objects.

Although the CORBA middleware seems to provide all the necessary support for building and executing ODSs, it only provides a very limited support to the evolution of their requirements. The main problem is that the description of the elements from which systems are built, and the way in which they are composed remains still mixed into the application code. This problem makes those systems difficult to understand, modify and customize. To our point of view a possible solution to this problem goes through the introduction of the so-called coordination models and languages into the CORBA model. The main goal of a coordination model and language is to separate computational and coordinational aspects in a distributed system. Separation of concerns facilitates abstraction, understanding and evolution of concerns. We propose in this paper a coordination programming system for ODS called CORODS. This programming system introduces the CoLaSD coordination model into the CORBA framework under the form of a coordination service. The CoLaSD model is an extension of the CoLaS coordination model introduced in [CRU 99a] to the coordination of distributed active objects.

## 2. The CORODS System

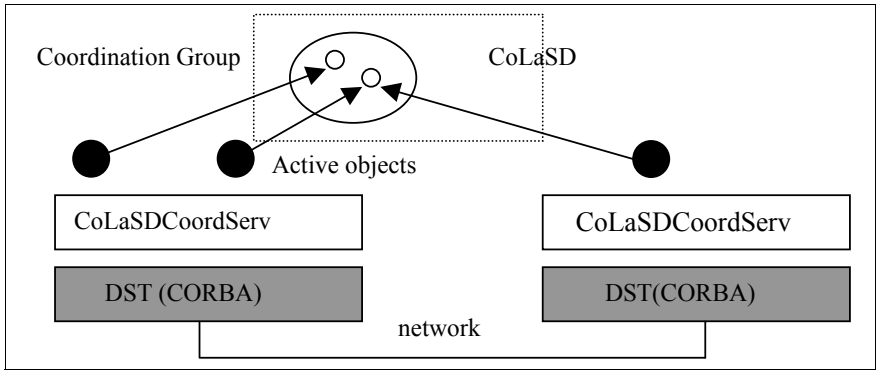


**Figure 1.** *The CORODS System*

CORODS is a coordination programming system for distributed active objects based on the CoLaSD coordination model. The CoLaSD coordination model is based on the notion of coordination Groups [CRU 99a]. A Coordination Group is an entity that specifies and enforces cooperation protocols, multi-action synchronizations, and proactive behavior within groups of collaborating distributed active objects. A prototype of the CORODS system is being built on top of a middleware framework called DST (Distributed Smalltalk)[CIN 00]. DST is a CORBA 2.0[OMG 95] compliant framework for Smalltalk. From a coordination point of view, the DST framework provides the basic facilities necessary to implement the CoLaSD coordination model (i.e. remote object interaction, lifecycle service, distributed naming service, distributed transactions). The CoLaSD model is introduced in the DST framework as a basic coordination service. The CORODS coordination service CoLaSDCoordService supports the creation, the reference, the modification and the destruction of Coordination Groups across the network. In fig 1, we show the structure of the CORODS coordination programming system and the elements that compose it. They are: a CORBA compliant middleware framework (DST in this case), and a coordination service CoLaSDCoordServ based on the CoLaSD model.

### 2.1. *DST-A Middleware Framework*

DST (Distributed Smalltalk) is a middleware framework that provides an advanced object oriented environment for prototyping, development and deployment of CORBA 2.0[OMG 95] compliant distributed applications. CORBA is the standard interface of the central component of the OMA (Object Management Architecture) architecture the Object Request Broker (ORB). The CORBA standard defines common methods of communication between distributed objects on heterogeneous platforms. The most important function of an ORB is to enable a

client to invoke operations on a potentially remote object. To communicate with remote objects, a client must identify the target object by means of an object reference. The ORB is responsible for locating the object preparing it to receive the request, and passing the data needed for the request to the object. If the operation identified by the request implies some reply from the remote object the ORB is responsible for communicating the reply back to the client. One of the most important features of CORBA is its IDL (Interface Description Language) language. The IDL language is used by the other components of the OMA to specify the services they offer through the ORB. A set of common services have been defined in the OMA architecture. These services represent generally useful services independent of the application domain. They are called Common Object Services (COS) and currently they are 15. DST provides six of them: naming, lifecycle, event notification, transactions, persistence, and concurrency control.

The DST middleware provides additionally a facility called the Implicit Invocation Interface (I3), and extension to the CORBA facilities that provides a paradigm for developing distributed Smalltalk applications. Instead of explicitly specify distributed classes interfaces using IDLs, Smalltalk developers using DST can turn on the I3 message transmission mechanism and allows I3 to handle object marshalling and unmarshalling between distributed Smalltalk objects. The I3 mechanism avoids developers to specify distributed classes interfaces in IDL. From a coordination point of view DST provides the basic facilities required to implement the CoLaSD model: remote object interaction facilities, a distributed naming service (to locate distribute objects by names independently of the place where they find), a lifecycle service (to control creation and destruction of distributed objects) a concurrency control service (to mediate concurrent access to distributed objects), and a transactions service (to control atomicity of distributed transactions).

### 2.2. *The CoLaSD Coordination Model and Language*

Coordination technology addresses the construction of open and flexible systems from active and independent software entities in concurrent and distributed systems. A coordination model and language specifies the glue that binds the independent software entities in those systems [GEL 92]. CoLaSD is a coordination model and language based on the notion of Coordination Groups. A Coordination Group is an entity that specifies controls and enforces the coordination of a group of collaborating distributed active objects. The CoLaSD model is based on the CoLaS model introduced in [CRU 99a]. The CoLaSD model is built out of two kinds of entities: the Coordination Groups and the Group Participants (Participants in the following). The original CoLaS model is extended in CoLaSD to specify coordination of distributed active objects. It takes into account the possibility of failures in participants during the coordination. The CoLaSD model replaces the CoLaS object communication model by the ACS (Apply, Call, and Send) communication protocol [RAC 92]; a protocol designed to support reliable object distributed applications. The ACS communication protocol merges the nested actions model with the model of nested asynchronous invocations messages. Services are enclosed inside atomic actions and the logical nesting of services corresponds to the nesting of actions. Action trees representing logical nesting of

invocations can be built, giving full control on the granularity of the atomicity. The CoLaSD model includes some other important modifications like a new internal object model for the participants, and new coordination elements within the Coordination Groups. We will explain in details these modifications in the next subsections.

### 2.2.1. *The Participants*

In CoLaSD the participants are atomic distributed concurrent active objects (distributed objects in the following). They are concurrent because they can process multiple methods invocations at the same time. They are active [PAP 95] because they have control over the concurrent methods invocations. They are distributed because physically they may run on different processors or machines. And, they are atomic because they process invocations atomically.

In CoLaSD distributed objects control concurrent method invocations according to a Readers and Writers synchronization policy. The Readers and Writers synchronization policy specifies that invocation methods defined as writers are processed mutually exclusively, and that invocations methods defined as readers are processed concurrently between readers and mutually exclusively with writers methods invocations. By default every method invocation is consider as writer, however, it is possible to define which methods invocations must be consider as readers. A method invocation that can not be processed because of the Readers and Writers policy, is reified and stored into the object's mailbox until the object is ready to process it. There is not limit in the number of invocations that can be stored in the mailbox, and no guarantee on the order on which stored invocations are processed by the distributed object. The internal object model used in CoLaSD to build the active objects is different to the internal model used in the CoLaS model. In the CoLaS model the active objects were internally sequentials: only one invocation method was processed at a time by an object. The CoLaSD active object model allows multiple methods invocations to be processed at the same time introducing potentially more parallelism to the applications.

Distributed objects in CoLaSD communicate by exchanging messages. Messages represent requests for method invocations. In CoLaSD the messages are exchanged asynchronously: distributed objects do not block while their invocations are processed by other distributed objects. Each request to a method invocation on a distributed object implicitly generates a *Future*. The Futures are used to reply results to methods invocations. When an object needs to use the result of a method invocation, it sends the message result to its future. If the result is not ready the object blocks in the future, if the result is ready it receives the result and continues working. Futures are also used to synchronize senders and receivers of invocations (synchronous communication). When a distributed object wants to communicate in a synchronous way with another object, it sends a method invocation to the other object, and waits in its future until it receives a notification indicating that the invocation was completely processed by the other object. The object that sent the method invocation sends the message wait to the future to receive such notification. If the invocation has not been completely processed the object blocks in its future.

Failures in the execution of the distributed objects and/or in the communication system may modify and thus affect the overall consistency of a system. The

consistency of a distributed object system depends on assertions done by the distributed objects about the state of other distributed objects. Several solutions have been proposed to tackle this problem [LIS 83][WAL 90](just to mention some of them). Basically what they propose is to give to the object message passing metaphor a powerful semantic for reliable distributed computing. The most interesting work to our viewpoint is [RAC 92]. In this work a communication protocol named ACS is introduced. This protocol merges the nested actions model [MOS 81] with the model of nested asynchronous methods invocations. In the ACS protocol, objects communicate through three different types of asynchronous message passing: Apply, Call and Send. All three types of asynchronous messages imply the execution of some subaction on another object. An Apply message sent by an object to another object implies the abort of the sender action if a failure occurs during the execution of the Apply subaction. A Call message behaves similarly to an Apply message; it differs in that Call messages do not imply the abort of the sender actions if a failure occurs. Different subactions are thus allowed to fail independently of each other. The sender may know that the request has failed, and may consequently choose to abort or to continue the execution of its action. A sender may know that a request has failed by sending to its future the message failed. In both Apply and Call messages, the sender may decide to abort its actions. All the subactions are also aborted if this happens.

To provide a simple way to trigger a subaction not requiring atomicity, the ACS protocol introduces a third kind of asynchronous message called Send. Senders of Send messages, does not rely, neither expect any reply from the execution of the subaction on the other object. CoLaSD replaces the basic asynchronous communication model of CoLaS by the ACS communication protocol. Any message send to another object must be preceded by a special message (protocol message) indicating the type of the asynchronous message it precedes. There are three types of protocol messages: apply, call and send in CoLaSD. By default a message received without a precedent protocol message is consider as a send message. The ACS protocol is built on top of the DST nested transactions service. Several modifications were done to adapt this service to a model of active object. They include for example, the introduction of multiple transaction contexts. Although this is an interesting work, we do not present these modifications here; they will be out of the main scope of the paper.

### 2.2.2. *Coordination Groups*

A Coordination Group (CG in the following) is an entity that specifies, control and enforces the coordination of a group of distributed objects in the realization of a common task. Each CG has associated a name. This name is used to identify uniquely CGs across the network. The primary tasks of a coordination group are: (1) to enforce cooperation actions between participants, (2) to synchronize the occurrence of participant actions, and (3) to enforce proactive actions (in the following proactions) on participants based on the state of the coordination.

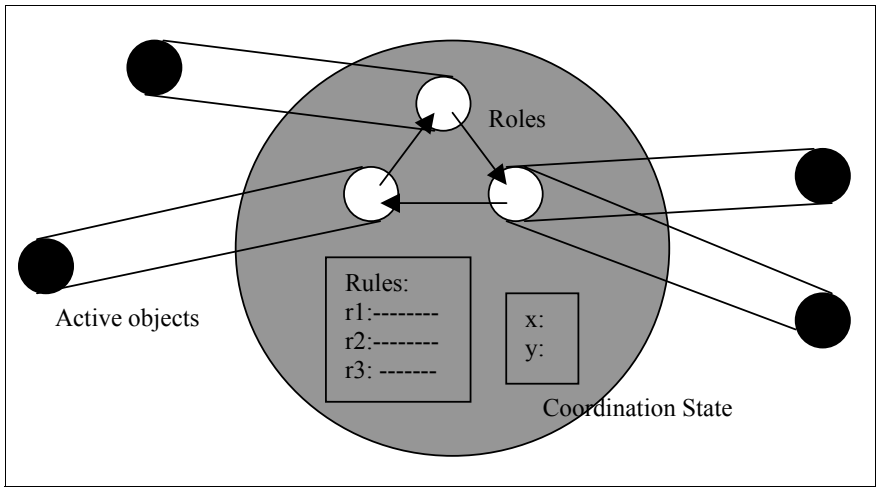


**Figure 2.** *A Coordination Group*

2.2.2.1. Coordination Specification.

A CG is composed of six elements: a Role Specification, a Coordination State, a Cooperation Protocol, Coordination Interceptors, Multi-Action Synchronizations and Proactions (fig. 2).

-*The Role Specification*: defines the roles that the participants may play in the group. Distributed objects join groups in CoLaSD by enrolling to group roles. A role identifies abstractly a set of entities sharing the same coordination behavior within the group. Each role has an associated role interface. To play a role, an object should posses at least the functionalities required by the role interface (interface compatibility). Roles can be played by more than one distributed object, and distributed objects can join more than one role. Nevertheless, to avoid semantic problems during coordination, we avoid that distributed objects join multiple roles having common coordination behavior.

-*The Coordination State*: defines general information needed for the coordination of the group. It concerns information like whether a given action has occurred or not in the system (i.e. historical information), or information about the state of a participant (i.e. busy or free). The Coordination State is specified by declaring variables within the CG. The variables are associated to the group or to each participant playing a given role.

-*The Cooperation Protocol*: define implications between participants actions. These implications have the form <Role> defineBehavior:<Message> as:<Coordination Actions>. They represent actions that have to be done whenever a participant playing a role <Role> receives an invocation request <Message>. The <Coordination Actions> include actions that: manipulate the coordination state of

the group, that select specific participants or collections of participants playing some role, and invocation requests sent to roles or to specific participants. The cooperation protocol corresponds to behavior specifically related to the coordination. This behavior must not be known in advance by the participants, they ”learn” it when they enroll to the roles, and remains part of the participant knowledge as long as the participant stays playing the role. Behavior implications are new to the CoLaS model. They are introduced in the CoLaSD model to provide a clean separation of the coordination and computation behavior between participants and coordination groups.

-*Coordination Interceptors*: the interceptors define actions that modify the coordination state of a CG at different moments during the handling of an invocation request by a participant. The interceptors have the form <Role> <Message> <Entry Point> <State Actions>. There are three different types of interceptors according to the moment (or <Entry Point>) at which the invocation request is intercepted: InterceptAtArrival, InterceptBeforeExecution, and InterceptAfterExecution. InterceptAtArrival specifies that the <State Actions> have to be performed when the message <Message> arrives to some participant playing the role <Role>. InterceptBeforeExecution and InterceptAfterExecution specify that the state actions have to be executed before and after respectively of the execution of the message <Message> by the participant. <State Actions> are actions that affect exclusively the coordination state of the group.

-*The Multi-Action Synchronizations*: specifies synchronizations constraints over message exchanged between participants. Multi-action synchronizations have the form <Role> <Message> <Operator> <Synchronization Conditions>. They specify conditions that constraint the execution of a message <Message> received by a participant playing the role <Role>. Two types of operators can be specified: Ignore and Disable. Depending on the operator, the message <Message> should be ignored or delayed by the object. Because of the non-determinism in which participant actions may occur in a system, multi-action synchronizations constraints are necessary to ensure properties such as: (1) mutual exclusion, and (2) temporal ordering of invocations. The synchronization conditions refer to information like: the state of the group, the identity of the receiver or the sender of a message, historical information on the coordination, etc. For a detailed description of the kinds of synchronization conditions refer to [CRU 99a].

-*The Proactions*: Until now the coordination of the system has been purely reactive. Coordination Actions are done in response to the reception invocation requests; they cannot be initiated by their own by the CG. Proactive behavior [AND 96] is introduced in CoLaS to specify actions that must be enforced by the CG independently of messages exchanged by participants (assuming that a certain condition holds). Proactions have the form <Conditions> <ProOperator> <Coordination Actions>. The <Conditions> are conditions referring the coordination state of the CG. Only one kind of ProOperator can be specified: validateAlways. The ValidateAlways proaction ensures that the proactive actions are executed always that the conditions specified on the proactions are satisfied. The evaluation of the conditions is done periodically by the CG in a non-deterministic way.

The last four elements of this model are specified using rules [AND 96][MIN 97]. The advantage of using rules is that they make the coordination explicit. They avoid programmers to deal with low-level details on how the coordination occurs.

The coordination is specified in a CG independently of the internal representation of their participants. Distributed objects can be used independently of how they are coordinated, and coordination protocols may be used independently on different groups of distributed objects. This independence allows for a clear separation of computation and coordination concerns, as promoted by coordination models and languages. It promotes abstraction, understanding and evolution of both concerns. The CoLaSD model combines the advantages of using groups as an organization abstraction and rules as explicit entities regulating the coordination aspect. We consider nevertheless, that the main advantage of CoLaSD is that CGs are highly dynamic entities that support evolution of the coordination. This is done in three distinct axes: (1) CGs are created dynamically at any time, (2) objects join and leave groups at any time, and (3) the behavior of a CG can be modified dynamically to adapt to new coordination requirements. As we mentioned before evolution is the most difficult requirement to meet, since not all the application requirements can be known in advance.

### 2.3. *The Coordination Service - CoLaSDCoordService*

The CoLaSDCoordService coordination service supports the creation, the reference, the modification and the destruction of CGs across the network. This service was created using the basic lifecycle service of DST. The basic lifecycle service of DST provides support for creating, deleting, copying and moving objects both locally and remotely, facilities required for object population control and object migration. All the CGs created across the network within CORODS are created using the CoLaSDCoordService. The service uses a coordination group factory object to create the CGs. In the Common Object Services (COS) terminology of CORBA, factories are objects that create objects in response to clients requests. In the DST framework a factory is any class that can be instantiated and has interfaces registered for creating objects in the interface repository. Factory objects are registered during the initialization factories phase of the ORB initialization. For a class to be registered as a factory it must have an instance method call abstractClassID (which returns the appropriate UUID-Universal Unique Identifier of the class).

To locate the correct class which then create the new objects, the COS specification of the lifecycle service introduces the notion of factories finders. A factory finder is an object at a specific location that helps clients to locate factories of a particular class. In order to create a remote object, clients need an object reference to the factory where the remote object will be created. The CoLaSDCoordService manages transparently all the interaction with the coordination group factory object and the DST lifecycle service to control the creation of CGs. A client of the CoLaSDCoordService must only know the interface of the coordination service.

To create a CG, a client sends the message createCGNamed:<Group Name> to the coordination service. Each CG created by the coordination service has a unique

name associated with it. The name of a CG is used to identify and to obtain references to CGs from the CoLaSDCoordService coordination service. If a request for a CG reference is done to the coordination service, and the CG does not finds at the same place (same machine) where the request was done, the coordination services replies a remote reference (a proxy) to the remote CG. In other case the service replies the reference of the requested CG. Requests for CGs references have the form getReferenceToCGNamed: <Group Name>. For a user manipulating a CG is transparent whether CG finds locally or remotely.

By using the CoLaSDCoordService all the CGs created are potentially accessible and manipulable from whatever place across the network. The CoLaSDCoordService itself is accessible to clients by means of the DST naming service (the naming service is a COS service). Clients can receive references to the CoLaSDCoordService by sending the message contextResolve: 'CoLaSDCoordService' asDSTName to the naming service. References to the naming service are obtained via the initial references mechanism of the ORB. This mechanism allows an application to know which objects have references available in the ORB. A listInitialServices operation is provided which returns a sequence of names. Each name represents an object which is available through this mechanism. Currently there are four objects available in DST at initialization: the Interface Repository, the Naming Service, and the Factory Finder. To obtain the object reference to each one of these objects the operation resolveInitialsReferences: <Object Name> must be used.

## 3. An Example- The Administrator Pattern [PAP 95]

We present in this section an example of the use of the CORODS programming system. We define a CG around a classical pattern used in distributed systems ”The Administrator”. The administrator is an object that uses a collection of ”workers” objects to service requests. The administrator pattern defines three kinds of entities: (1) the clients that issue requests to the administrator, (2) the administrator that accept requests from multiple concurrent clients, distributes the requests to the workers and forwards back the results to the clients, and (3) the workers that handle the administrator requests and send back results to the administrator. The administrator may seek to maximize parallelism by load balancing or it may allocate jobs to workers based on their individual capabilities. This example illustrates the following coordination problems:

-*Transfer of information between entities*: client requests received by the administrator must be forwarded to the workers. The administrator must decide which request goes to which worker. Additionally, the administrator must control the transfer of replies from workers to clients.

-*Assignment of share resources*: the administrator controls the assignment of request to workers. The shared resource in this case is the workers processing time. The administrator may apply different assignment policies. To simplify the problem we decided that all the workers have the same capabilities and that we implement a simple assignment police the ”first free” policy. In this policy the administrator chooses in a non deterministic way one free worker between its workers.

Additionally the administrator must prevent the assignment of multiple requests at the same time to the same worker.

*Dynamic Evolution of the coordination*: The systems must be able to scale. New workers may be added or removed from the system at any time. Additionally, the system must accept the arrival of new clients. Assignment policies may also change at runtime.

**Site-Ziyal -**

```
1.adminCG defineRoles:#(administrator workers).
2.adminCG defineInterface:#(do:) forRole:workers.
3.
4.adminCG defineParticipantVariable:free forRole:workers initialValue:true.
5.
6.
7.[1] administrator defineBehavior: 'do:args' as:
8.          [|worker |
9.          worker :=
10.                    workers selectAParticipantThat:          "Select a free worker "
11.                    [:aParticipant|
12.                    group valueVariable:free ofParticipant:aParticipant].
13.         group setVariable:free ofParticipant:worker value:false.
14.         ^worker apply do:args ]
15.
16.[2] administrator 'do:args' disable:
17.         [ workers notExistsAParticipant: "Validates if there is a free"
18.                    [:aParticipant |          "worker."
19.                    group valueVariable:free ofParticipant:aParticipant ]].
20.
21.[3] workers 'do:args' InterceptAfterExecution:
22.         [ group setVariable:free ofParticipant:receiver value:true ]
```

**Figure 3.** *The Administrator Pattern*

### 3.1. *Role Specification*

In the administrator example fig. 3, participants play one of the two roles: the administrator or the workers. These roles are defined in line 1. The minimal interface that objects should support to play the role workers for example is specified in line 2. The method `do:` specified in the interface corresponds to the service that the administrator offers to its clients and that workers execute. Suppose `do:` is an increment service that increases the value of received argument by ten.

### 3.2. *Coordination State*

The coordination state of the group is specified by declaring variables. In the administrator example we have only one, the variable free defined in line 4. This variable is a participant variable. Each participant playing the role workers has one variable free associated with it. This variable is used by the administrator to control the assignment of request to workers. When the variable validates to true it indicates that the worker is free to do some job, and when it validates to false it indicates that the worker is busy.

### 3.3. *Behavioral Specification*

In the administrator example three rules are defined:

*Rule 1* (line 7). A do: request received by the administrator triggers a do: request on a free worker. The state of the worker is set to busy by changing to false the value of its coordination variable free. In line 14, we can see how the administrator forwards the request it receives from the client to the selected worker. Furthermore, the administrator replies "^" to its client with an implicit future that contains the result sent back by the worker that executes the job. Each request to a method invocation on a distributed object implicitly generates in CoLaSD a future. In this particular example the administrator reply contains the future it receives from the invocation of the method do: on the worker. When the client wants to verify if the reply to its invocation request is ready, it sends the message result to the future. The client blocks if the worker has not finish to process its request.

*Rule 2* (line 16): Requests are delayed by the administrator when no worker is free to handle the received request. To determine whether a worker is free or busy, the rule uses the coordination variable free associated with each worker.

*Rule 3* (line 21): Once a worker has finish to perform a request the state of the worker is updated to free. We use an interceptAfterExecution coordination interceptor to do this. This interceptor is applied after the execution of the behavior do: on the worker. It modifies the coordination variable free of the worker that processed the request to true.

### 3.4. *Pseudo-Variables*

There are three pseudo-variables that can be used within the CGs. They are: group, receiver, and sender. The group variable refers to the CG on which the variable appears (lines 12, 13, 19 and 22 in fig. 3). The sender variable refers to the distributed object that sent the invocation request and the receiver variable the distributed object handling the invocation request (line 22).

### 3.5. *Failures*

In line 14 fig. 3, we see how the ACS is used in the example. The apply special protocol message precedes the invocation of the do: request on the selected free worker. This implies that in case that the invocation do: fails on the worker the system will abort the do: action of the administrator. The client may ask to its future is the request has failed by sending the message failed to its future, and modify the rule to define a completely different strategy like retrying the execution of the operation on another worker. In Fig. 4 line 9, the apply message has been replaced by a call message and a validation to the ocurrence of a failure. If a failure is detected during the execution of the do: operation on the worker, the operation is retried on another worker

```
1.[1] administrator defineBehavior:'do:args' as:
2.      [|worker result|
3.
4.      worker :=
5.              workers selectAParticipantThat:
6.              [:aParticipant|
7.              group valueVariable:free ofParticipant:aParticipant].
8.      group setVariable:free ofParticipant:worker value:false.
9.      (result := worker call do: args) failed    "verifies failure"
10.     ifTrue:
11.             [worker :=
12.             workers selectAParticipantThat:
13.                     [:aParticipant|
14.                     group valueVariable:free ofParticipant:aParticipant].
15.             group setVariable:free ofParticipant:worker value:false.
16.             result := worker call do: args]
17.     ifFalse: ...
18.
19.     ^result ]
```

**Figure 4.** *Managing Failures*

### 3.6. *Group Creation and Enrolment of Participants*

To illustrate how active objects are created and then how they join the CGSs in a distributed session we define a specific scenario in which we have an administrator, and two workers. The administrator, the workers and the CG run on three different machines (Ziyal, Albert, and Globi). The administrator participant and the ADMINISTRATOR CG run on Ziyal. One worker Worker1 runs on Globi, and the other worker Worker2 runs on Albert. The CORODS system is installed on each of the three machines

In fig. 5 (Site 1 line 5) we can see how the 'ADMINISTRATOR' CG is created using the CoLaSDCoordService coordination service. The CG becomes in this way

potentially accessible to participants running on different machines. A distributed object that wants to participate in a CG must contact the coordination service to obtain a reference to the CG. In Site2 lines 4 and 5, we can see how a worker gets a reference to the CG. In line 8 site 2, we can see how a worker active object uses this reference to enrol into the CG in the role workers

The Administrator and Worker classes are subclasses of the class COLASDDistributedObject. In CoLaSD every distributed object class must be a subclass of COLASDDistributedObject . This special class introduced in CORODS manages transparently all aspects related with the internal activity of objects and their interaction with the CGs. Instances of distributed objects are create using a creation method active (line 7 sites 1 and 2).

**Site 1- Ziyal**

```
1. namingService coorService adminCG administrator |
2.
3. namingService := ORBObject resolveInitialReferences:#NameService.
4. coordService := namingService contextResolve:'CoLaSDCoordService' asDSTName.
5. adminCG := coordService createCGWithName: 'ADMINISTRATOR'.
6.
7. administrator := Administrator active.
8. namingService contextBind: 'SERVER' asDSTName to: administrator.
9. adminCG addParticipant:administrator withRole:administrator.
```

**Site 2- Globi**

```
1. namingService coordService adminCG administrator worker1|
2.
3. namingService := ORBObject resolveInitialReferences: #NameService.
4. coordService := namingService contextResolve: 'CoLaSDCoordService' asDSTName.
5. adminCG := coordService getReferenceToCGNamed: 'ADMINISTRATOR'.
6.
7. worker1 := Worker active.
8. adminCG addParticipant: worker1 withRole: workers.
```

**Figure 5.** *Group creation and enrolment of participants*

### 3.7. *Dynamic Properties*

CoLaSD supports three types of dynamic coordination changes: (1) new participants can join or leave the group at any time, (2) new groups can be created and destroyed dynamically, and (3) the coordination behavior can be changed by adding or removing rules to the CG. To introduce new participants a message addParticipant: <newParticipant> forRole:<Role> must be sent to the CG (line 8 site 2 fig 5). To leave the CG a participant must send the message removeParticipant: <Participant> fromRole:<Role> (or removeParticipant: to remove it from all the roles it plays). To add new coordination rules a message

addRule:<aRule> must be sent to the CG. The rule definition must correspond to one of the four different types of rules specified in CoLaSD: cooperation rules, multi-actions synchronization rules, coordination interceptors rules, or proactions rules. To remove a rule a message removeRule: <aRule> must be sent to the CG. In the administrator example it is possible to adapt dynamically the number of workers in the system. The administrator may decide for example to add new workers to the system whenever the number of pending requests is too high and to remove those workers once that the number of pending requests becomes normal .

## 4. Related Work and Conclusions

Traditionally the coordination layer of Open Distributed Systems have been managed using concurrent and distributed object oriented languages. These languages provide only limited support for the specification an abstraction of the coordination. As a result, the coordination founds mixed into the application code, making it difficult to identify, to scale and to customize. The idea of separating coordinational aspects of systems using coordination languages was introduced by [GEL 92] in their language called Linda. The Linda model is a data driven model. The main goal of the CoLaSD model is to support coordination of ODS. As concurrent and distributed object oriented languages promote data encapsulation and behavior over the data, we think that naturally the coordination in object oriented systems must be control driven. CoLaSD is a control driven model because CGs enforces and control actions occurring in the system not on data. Due to space limitation we limit this related work to coordination languages specifically designed for object oriented systems and to coordination languages which use the same approach of message interception to realize the coordination (for a complete related work refer to [CRU 99a]). The most important related works are Synchronizers [FRO 93] and Moses [MIN 97]. Both specify coordination using rules as in CoLaSD. The main differences with respect to CoLaSD are: (1) In synchronizers the coordination is restricted to synchronization of messages, in CoLaSD and Moses coordination actions can be enforced on participants too. Nevertheless, in Moses those actions only affect the receiver of the message. In CoLaSD coordination actions may affect any participant of a CG. (2) In Moses and Synchronizers the coordination state refers only to local information of a participant. In CoLaSD the coordination state may refer to the coordination state of any participant of a CG. (3) Synchronizers are pure reactive entities; they react to the arrival of message invocations. In both Moses and CoLaSD proactions may be initiated independently of the arrival of some message. (4) Synchronizers, neither Moses support dynamic evolution of the coordination as in CoLaSD. In CoLaSD the coordination can be modified on the fly: new coordination rules can be added, new distributed objects can join CGs, and new CGs can be create dynamically. Finally, (5) Synchronizers neither Moses includes the possibility of failures into their models.57

Concerning related work on introducing the so-called coordination models and languages into the CORBA model. Our work is very new; this idea was presented in [CRU 99b]. To our knowledge the only work that could be consider as related in this domain is [DRI 99]. This work proposes a cooperation service for CORBA based on graph grammar techniques. The main differences with respect to our approach are: (1) they coordinate sequential objects (2) coordination is specified as

graphs transformations, and (3) they do not manage the evolution of the coordination rules that applied over the coordination graphs,

We consider the CoLaSD coordination model a good candidate to the integration with the CORBA model. This integration will provide the necessary support to build and evolve ODS. It will provide separation of concerns between computation and coordination in ODS simplifying the understanding, modification and customization.

## 5. Bibliographie

[AND 96] J-M.ANDREOLI, S. FREEMAN and R.PRESCHI, *The Coordination Language Facility: Coordination of Distributed Objects*, (TAPOS), vol.2, n. 2, 1996, pp. 635-667.

[CIN 00] CINCOM Inc., *Distributed Smalltalk*, 1998,http://www.cincom.com/

[CRO 96] J. CROWCROFT, *Open Distributed Systems*, UCL Press, 1996.

[CRU 99a] J.C.CRUZ and S. DUCASSE, *A Group Based Approach for Coordinating Active Objects,* COORDINATION '99, LNCS 1594, Springer Verlag, pp. 355-370.

[CRU 99b] J.C.CRUZ and S. DUCASSE, *Coordinating Open Distributed Systems,* Future Trends of Distributed Computing Systems, IEEE, pp. 125-130.

[DRI 99] K.DRIDRA, F, et all., *A Cooperation service for CORBA objects,* EuroPar'99 LNCS 1685.

[FRO 93] S.FROLUND et all., A Language Framework for Multi-Object Coordination, ECOOP'93, LNCS 707, Springer Verlag, pp. 346-360.

[GEL 92] D. GELERNTER and N. CARRIERO, *Coordination Languages and their Significance*, CACM vol. 35, n. 2, February 1992.

[LISK 83] B.LISKOV and R.SHEIFLER, *Guardians and Actions: Linguistic Support for Robust Distributed Programs*, ACM TOPLAS, July 1983.

[MIN 97] N.MINSKY et all, *Regulated Coordination in Open Distributed Systems*, COORDINATION'97, LNCS 1282, Springer-Verlag, 1997, pp. 81-97.

[MOS 81] J.E.B MOSS, *Nested Transactions and Reliable Distributed Computing*. Ph.D Thesis, MIT, 1981

[PAP 95] M.PAPATHOMAS, *Concurrency in O.O. Programming Languages*, Object Oriented Software Composition, P. Hall.

[RAC 92] R.GUERRAOUI, et all, *Nesting Actions thorough Asynchronous Message Passing: the ACS protocol*, ECOOP 92.

[OMG 95] OMG, *The Common Object Request Broker: Architecture and Specification*,1995.

[WAL 90] E. WALKER, et all, *Asynchronous Remote Operation Execution in Distributed Systems*, Proc. IEEE Conf. on Distributed Computing Systems, May 1990