Cooperation Services for Widely Distributed Applications

Manuel Munier, Claude Godart
UMR n^{o}7503 LORIA - Université Henri Poincaré
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex - FRANCE
E-mail: {munier,godart}@loria.fr

Abstract

Due to the popularization of Internet, widely distributed cooperative applications are expected to become commonplace on the Web. Virtual enterprises and mobile computing are good examples of such distributed applications. They both involve several actors connected together and needing to co-operate by sharing some (possibly unstable) data. To ensure consistency of shared data, traditional distributed systems are based on a client/server architecture. However, this implies that client activities have to be connected to the server continuously.

This paper describes our approach to build distributed cooperative applications using a peer-to-peer architecture. An activity is viewed as a self contained component which cooperates with other components by exchanging, during its execution, some results. To ensure consistency of shared data, two activities have to negotiate a cooperation pattern first, which purpose is to control subsequent exchanges of these data between these two activities.

1. Introduction

Due to the popularization of Internet, widely distributed cooperative applications are expected to become commonplace on the Web. Indeed, organisations whose staff work at widely distributed sites invest much time and incur large travel expenses to work as a team, exchange information and ensure coordination among their various sites. This effort can be considerably reduced by using a fairly sophisticated mechanism of communication and coordination. As the various activities of such a distributed system can share some (possibly unstable) data, we need to make sure that whenever there are changes, we ensure that people get notified whenever they should have been notified, and that the efforts of various people on the project are coordinated well enough to absorb the impacts of the changes.

However, ensuring the safety of data exchanges between cooperative applications should not restrict their autonomy, neither for data access (by using a centralized server to store all the data of the system) nor for coordination of the interactions (by assigning the control of all the exchanges to a single server). This means that we don't want to force cooperative applications to be connected to some server at all times. Obviously, this is the main requirement in the case of applications running on mobile terminals or desktop computers connected to the network by lazy connections (slow connections or even connected occasionally via modem).

Virtual enterprises are another kind of widely distributed cooperative applications. The concept of virtual enterprise depicts the idea that many applications are the result of cooperation between several actors, playing different roles and building a temporary relational system structured by a common objective and for the duration of a project. Building trade is a good example of such short-life enterprises: it involves several partners (architect, research consultant, control office, building firm, electrician, carpenter, ...) who constitute an enterprise for the duration of the building construction. Virtual enterprises require new technologies to organise their short duration networks. As partners can be small enterprises and have a small computer infrastructure, technologies must be easy to understand and to use, and must not disrupt the habits of users. Particularly, this means that it is not acceptable to force them to move all the data on which they work on a centralized server, to which they have to be connected expensively at all times. Moreover, many of the partners will not agree their (possibly private) data will be managed by somebody else !

With these requirements in mind, we have developed an approach which allows to develop cooperative applications by assembling basic cooperation patterns. To assume acceptance of assembling, cooperation patterns can be negotiated between partners. To assume safety of assembling and execution, each concurrent engineering activity executes as

a transaction, i.e. in a frame which allows to assert safety properties on executions.

This paper describes our approach and presents some preliminary results. In a first time, section 2 explains why we chose to develop a peer-to-peer architecture. Section 3 presents our cooperation model to define and to manage cooperation patterns. Then, section 4 gives the principle of our implementation of this cooperation model as CORBA services. Finally, we conclude by generalizing the principles developed previously to better cover the characteristics of a lot of concurrent engineering applications.

2. Peer-to-Peer vs Client/Server Architecture

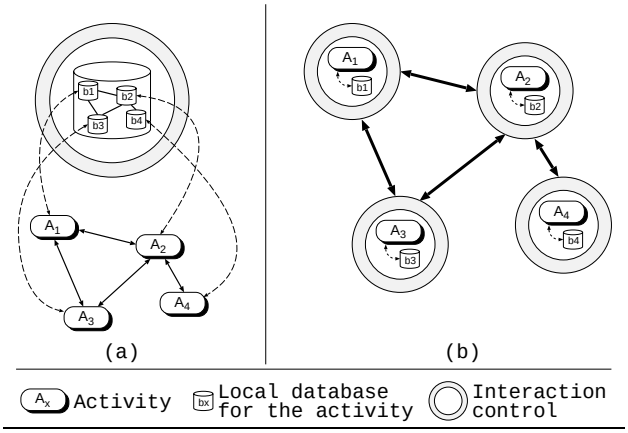
Most distributed systems are based on a client/server architecture in which, though single activities may execute at geographically distributed nodes, the knowledge about the processes which execute is kept in a centralized database at the server level. This centralization makes it easier to synchronize and monitor the overall execution as all decisions are taken on this server which has a global view of the whole system. With this architecture, configuration management tools like Continuus [9], ClearCase [1, 2], Adèle [4], or database systems using a transactional approach [6, 12, 3, 8] can be used to coordinate updates performed by activities.

Using a client/server architecture, a distributed system is viewed as a set of clients (activities) connected to a server (figure 2.1-a). Even if we use caching or replication features to store some data on activities, data exchanges between these activities always have to go through the server. This does not correspond to the nature of the applications that we consider in which:

- Each actor already has his own environment and doesn't want to change his habits. Moreover, he has to be free to work as he wants in his environment.
- Due to the expensiveness of network connections, actors are generally disconnected when they work, but this sould not prevent them to work.
- In large projects, nobody possesses the entire knowledge of the system. Therefore, it is extremely difficult and often impossible to determine, in a centralized way, all the possible impacts of a given change.

To support these requirements, our approach is based on a peer-to-peer architecture (figure 2.1-b). That is, each activity is both a *client* of and a *server* for other activities. The terms "client" and "server" are merely roles that are filled on a per-request basis. Very often, a client for one request

Figure 2.1 Centralized Control vs Distributed Control



is the server for another. Thus, an activity is viewed as a self contained component that cooperates with other components by exchanging, during its execution, some (possibly preliminary) results. By self contained we mean that an activity should manage itself its own data, both for data storage and interaction control. Doing this, each activity is responsible for its data exchanges with others activities. So, unlike configuration management tools or transactional systems, we avoid activity denpendency towards any kind of server, neither for data access nor for interaction control.

In order to coordinate data exchanges between activities, we need to define cooperation protocols. Such a protocol is a set of rules that two activities have to check if they want to share data consistently. Here are some rule examples (section 3.2 will go further into this notion of cooperation protocol):

- **no lost-update**: A result produced by an activity can not be overwritten by another activity before it is read by any other activity.
- **no dirty-read**: An activity can not publish a result produced from outdated parameters.
- **exchanges orientation**: If exchanges are oriented from the activity A to the activity B, B can read values produced by A, but A is not allowed to read values produced by B.
- **re-read of the final value**: If an activity A reads a preliminary result produced by an activity B, then A has to read the final value produced by B before commit time.
- **notification on update**: When an activity produces a new result for some data, it notifies all the activities with which it shares this data. This rule is used for group awareness rather than access control.

The main idea is that when an activity want to communicate with another one, these activities begin by negotiating a cooperation protocol. This negotiation will ensure, at least, that the protocol one activity wants to use is known by the other. Thus each activity will have a cooperation table which purpose will be to show which protocol to use to control the exchanges of a given data with a given activity.

Then, the system will ensure that all exchanges between these two activities will respect the negotiated protocol, while keeping them independent of one another: in case of protocol violation, only the faulty exchange is refused. From the user point of view, the main advantage of our approach is that we define cooperation protocols to coordinate data exchanges between activities and not to control the activities themselves. Therefore, an activity is largely independent of other activities for the task it performs. This means that each actor of the system is free to work like he wants, as long as his data exchanges with other actors are correct.

3. Cooperation Model

3.1. Cooperation by Intermediate Result Exchange

Our objective is to define a framework to build distributed systems by assembling several autonomous activities. These activities will cooperate by exchanging some data during their execution. These data are called intermediate results or inter values. An intermediate value is a value of a data which is subject to further modifications by the same activity and is generally inconsistent. For instance, in a software development team, somebody can start to develop an application using a preliminary version of a library. By preliminary version we mean that some functions of the library can be missing or untested. So this version of the library is now inconsistent with regard to its specification. However it is considered as being potentially interesting for other activities. In our example, it is not necessary to wait for the validated and final version of the library before to start to develop the application. The concept of intermediate value allow to break the isolation between activities during their execution.

If it is important to preserve consistency of data, this must not prevent users of doing work. For that, intermediate results allow users to work as they like, the only obligation being that final results of their activities must be consistent. In other words, an activity lives with inconsistencies but has to reach a consistent state before it completes. To ensure consistency of data shared by two activities, we define a set of rules that all their exchanges have to respect. Such a set of rules is called a cooperation protocol.

3.2. Coordination/Cooperation Control

As depicted on figure 2.1-b, a distributed system is a network in which a node is an activity and an edge is a basic cooperation protocol (called cooperation pattern) that controls data exchanges betwen the two related activities. But it is not sufficient to define patterns; we have to be able to manage interactions between several patterns too. Indeed an activity could share the same data with several activities using various patterns.

3.2.1. Basic cooperation patterns

The coordination of the exchanges between two activities is performed by a cooperation pattern. We already defined three basic patterns of cooperation [10, 11]:

- the **client/server** pattern: one actor (the client) can work on preliminary versions of a data produced by another actor (the server). The only compulsory rules are: the server must produce the final version of each data it has produced in a preliminary version, an d the client must take into account the last version produced by the server for each data it has read in a preliminary version.

  This pattern permits some cooperative work which enhances productivity by allowing to start the client activity before the end of the server activity while ensuring the client will read the final version of the data produced by the server.

- the **cooperative write** pattern: two actors can modify at the “same time” the same data. Actually, each of them modifies the copy of this document he owns in its local database. They have to follow some rules: to be aware of each other work (by exchanging preliminary versions of this data) and to converge towards a same view of this data (i.e. they have to agree on the same final version of this data).

- the **writer/reviewer** pattern: this third form of cooperation corresponds to the case in which an actor produces a data under the control of another. This pattern ensures that the final data produced by the server will be reviewed by the reviewer, and that the server will read the last version of the data produced by the reviewer. Thus both the writer and the reviewer have to agree to the same final versions of data they shared.

We can note that these cooperation patterns only control data exchanges between activities, and not the individual work of activities. As a matter of fact, when two activities want to exchange a data, each of them checks the pattern they negotiated according to its current state (i.e. according

to the state of data it shares with others activities). If at least one of them detects a protocol violation, the exchange does not occur. Therefore, each activity of the system is largely independant of the others, as long as its data exchanges with other activities are correct with respect to the negotiated cooperation protocols.

3.2.2. Interactions between patterns

When two activities need to cooperate, they negotiate a cooperation pattern to control their data exchanges between their respective local databases. So, an activity can share various data with many other activities using a different pattern with each of them. Thus each activity build its own cooperation table which purpose is to show which cooperation pattern to use to exchange a given data with a given activity. Such a cooperation table is depicted in figure 3.1.

Figure 3.1 Cooperation Table

activity	*remote activity*	*shared data*	*pattern*	*role*
myself	lib. provider	library	client/server	client
myself	co-developer	source code	cooperative write	$\sim$
myself	doc. writer	source code	client/server	server
myself	doc. writer	documentation	writer/reviewer	reviewer
myself	doc. writer	doc. advice	writer/reviewer	writer

Consequently, a same data can be shared in many ways (i.e. using various patterns). So, it is at the data level that patterns will interact and possibly conflict.

By conflict we mean that a given transfer operation between two activities could be allowed by one cooperation pattern whereas it could be prohibited by another one. In concrete terms, we use a transactional approach [1] to implement protocols, and each cooperation pattern is based on a "correctness criterion" which characterizes the set of correct interactions between two activities. To specify it, the principle is to store the sequence of some specific events (transfer operations between the two activities), i.e. to build an history of the execution, and to verify that this history respects some well defined properties. Thus, a conflict between two patterns occurs when an execution is correct for one correctness criterion but not for the other. To detect such conflicts between patterns, an activity can proceed in two ways:

[1] In [7, 11] we already used such an approach to define a new correctness criterion : the *COO*-serializability. Then we described a protocol to maintain this criterion in a client/server architecture (i.e. the coordination of the activities is ensured by a centralized persistent store). As the *COO*-serializability ensures the basic properties common to the three cooperations patterns (client/server, cooperative write, writer/reviewer), this protocol allows the assembling of these three patterns within a single framework.

- **statically** (i.e. before the execution): as described above, if two activities want to share data, they have to negotiate a cooperation pattern first. If an activity chooses to detect conflicts statically, it has to be able to test if the specifications (i.e. the correctness criteria) of two patterns are "compatible". This kind of problem can not only be complex, but it can also restrict the cooperation between activities. For instance, two activities can fail to negotiate a pattern because one of them already uses another pattern that could, in some cases, conflict with all patterns these activities agree for the negotiation.

  For instance, as defined in figure 3.1, the activity `myself` has to review the documentation and to produce a documentation advice. Now, let suppose it want to do this work with somebody else. For that, they could want to share the documentation using a cooperative write pattern to be able to annotate it. However, this creates a conflict because, as a reviewer, the activity `myself` is not allowed to modify it (with regard to the correctness criterion defining the writer/reviewer pattern).

- **dynamically** (i.e. at run time): during the negotiation, activities do not care for compatibility between patterns. Conflicts will be detected when they occur effectively, i.e. when an activity try to exchange some data with another activity. Let an activity A share the same data with various activities $B_1..B_n$ using the cooperation patterns $P_1..P_n$. It can proceed in two way:

  - each exchange between A and B_i is controlled by all the patterns $P_1..P_n$. A conflict will be immediately detected and the faulty exchange will be refused. In such a way we always ensure data integrity. But if such an exchange initiated by B_i is refused, this means the activity B_i could be disturbed by a cooperation pattern different from the one negociated with A (i.e. the pattern P_i).
  - an exchange between A and B_i is controlled only by the pattern P_i. So exchanges between A and B_i strictly respect the cooperation pattern negotiated between A and B_i . If such an exchange creates a conflict for a pattern P_j ($j \neq i$) between activities A and B_j, this conflict will be detected when the activity A will try a new exchange with the activity B_j. As we can see, the activity A is the only one to be aware of this conflict.

Obviously, the third solution is the best in the case of virtual enterprises or mobile computing, as a conflict has to

be resolved by the activity on which it occurs.

Neverthless, this solution can lead to some deadlocks. For instance, an activity cooperating with several other activities, possibly using various patterns, could end in a situation in which all the exchanges it will try will break at least one pattern and so will be refused. Thus we should provide not only tools to define cooperation patterns and to detect conflicts between these patterns, but mecanisms for conflict management too:

- **conflict notification**: first, the user (i.e. the actor driving the activity) should be noticed when a conflict occurs. However, the system should not only notify that an exchange operation was refused, but should also detail the reasons why it was refused.

- **pattern renegotiation**: such a deadlock situation could be due to the pattern negotiated between two activities and defined too strictly. So activities should be able to renegotiate this pattern.

- **version management**: another complementary solution is to be able to "undo" some work. As for configuration management tools, this means that an activity can develop, in its local database, its own version branche for data it uses. At the end of this activity, it should be able to produce its results according to any versions of these data. For instance, the activity could come back to the data version it imported. This means that the modifications this activity did on these data will not be visible to other activities.

3.2.3. Formal Description

As above-mentioned, we use a transactional approach. This means that our cooperation model can be considered to be a flexible transaction model, i.e. cooperating activities can be seen as being concurrent transactions. But unlike classical ACID[2] transactions, these activities (eventually distributed over a Wide Area Network) can exchange intermediate results. So we break down the isolation property.

To synthesize our model we use the ACTA transaction framework[8]. ACTA was designed to specify and reason about the nature of interactions between extended transactions in a particular model. ACTA is a first-order logic based formalism which allows to specify the effects of a transaction on other transactions (inter-transactions dependencies) and also to specify their effect on objects (visibility and conflicts between operations on objects) by means of constraints on histories.

[2] ACID stands for Atomic, Consistent, Isolated, Durable

The axiomatic definition of a transaction model aims to determine if a new event[3] can be invoked. Especially, the preconditions of the event derived from the axiomatic definition of its invoking transaction are evaluated with respect to the current history H_{ct}. If its preconditions are satisfied, the new event is invoked and appended to the H_{ct} reflecting its occurrence.

Using this formalism we define each cooperation protocol (i.e. a set of rules) by preconditions on events invoked by activities (begin/commit/abort of an activity, transfer operations, ...). These preconditions are first-order logic predicates which are evaluated on the (local) current histories of involved activities.

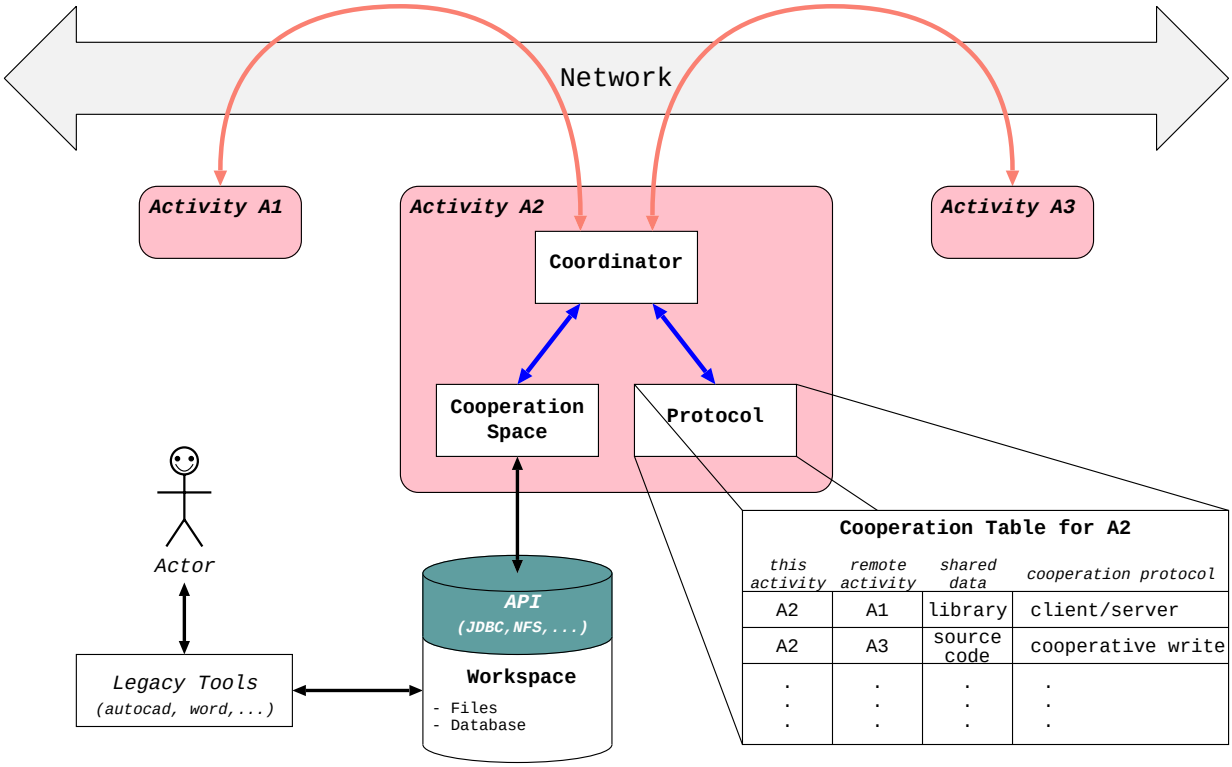
Obviously, one of our main objectives is to ensure that if a property is satisfied with respect to local histories of activities, then the same property is satisfied with respect to the global history of the whole system. From a transactional point of view this means that we want to distribute correctness criteria, i.e. to ensure classical global properties (like the serializability) using local controls on data exchanges between activities in a peer-to-peer architecture.

3.3. Cooperation Architecture

Now, we present the architecture we designed to *coordinate*, using various *cooperation patterns*, *data* exchanges between widely distributed activities. We must define how an activity stores its data locally, what is a cooperation pattern, and how activities control data exchanges using these patterns.

Remember that we don't want to disrupt the habits of actors. From the user point of view, this means that we should be able to work on shared data using legacy tools, which generally can access to files and directories only. So we split the local repository of an activity in two parts: a public one, called *cooperation space*, in which the activity will store versions and configurations of data it uses, and a private which is the *workspace* itself (i.e. where legacy tools will store their files). So, when a user imports a document from another activity, this document is stored in his cooperation space. To be able to access it through his legacy tools, he must check out this document to his workspace to obtain a file his applications can read and write. When he want to publish his work (possibly as intermediate result), he checks in the file to the cooperation space. This operation updates the document in his cooperation space, i.e. creates a new version of this document which is now available to other activities.

[3] An event is either an invocation of an operation on an object (object event) or a transaction management primitive (significant event) like `Begin`, `Commit` or `Abort`.



As depicted in figure 3.2, each activity is made of four main parts:

- a **Cooperation Space** which is the local repository of the activity. It is in charge of creating/deleting resources, checking in/out data between the cooperation space and the workspace, and managing version and configuration for local resources.
- a **Workspace** which shows resources as files and repositories (for instance), so actors can use their legacy applications.
- a **Protocol** which is based on the cooperation table of the activity. It ensures that a sequence of exchanges between two activities is correct (according to the cooperation pattern they negotiated).
- a **Coordinator** which is the activity interface for other activities. Thus all exchanges between activities have to be done through their coordinators. This prevents activities to perform, directly between their cooperation spaces, data exchanges which are not controled by any cooperation pattern. Moreover, it allows the cooperation space and the protocol to be defined independently.

As explained in section 3.2.3, a cooperation pattern is defined as a set of properties that have to be repected by data exchanges. These properties are used by the coordinator (cf figure 3.2) as pre-conditions for data exchange operations. So, any transfer operation which fails to respect one of its pre-conditions is refused by the coordinator. This means that a data exchange between two activities can occur only if the two coordinators allow it. Thus, each activity is completely responsible for the consistency of data it owns in its cooperation space.

The autonomy of activities is not the only advantage of such an architecture: another key feature is the scalablity. Therefore, an activity A only knows the activities with which it is directly connected and objects it shares with these activities. It needs only a local view of the system. Thus we can add as many activities and objects as we want, if they don't concern the activity A, this activity will not be disturbed (from a performance point of view) by these new activities and objects.

4. Implementation

An important characteristic of widely distributed systems, like virtual enterprises or mobile computing, is that they are heterogeneous. For example, in the case of a virtual enterprise, some partners could use mainframes or UNIX workstations while others use desktop computers or even

laptops. In recognition of these problems, the Object Management Group (OMG) defined the *Object Management Architecture* (OMA), whose key component is the *Common Object Request Broker Architecture* (CORBA) specification [13]. In the OMA Object Model, an object is an encapsulated entity with a distinct immutable identity whose services can be accessed only through well-defined interfaces. The *Object Request Broker* (ORB) component is responsible for communications between clients and objects. Thus, the implementation and location of each object are hidden from the requesting client. Unlike typical distributed software systems, which are tied closely to underlying networking protocols ans mechanisms, CORBA-based applications are abstracted away from the networking details and thus can be used in a variety of environments.

Another key feature is that CORBA itself and application built on top of it are best designed using object-oriented software development principles. For example, the fact that object interfaces must be defined in OMG IDL helps developers think about their applications in term of interacting reusable components.

Thus, to implement our cooperation model, we chose to conform to CORBA, which has became a standard in the distributed object community. Indeed, our objective is to provide a toolkit to build distributed cooperative applications in such distributed heterogeneous environment. We don't want to develop yet another proprietary system, but to define a set of cooperation services instead. In other words, we intend to describe what are the basic components needed to build distributed cooperative applications on top of CORBA. That way, our services can use and can be used by other CORBA services. So they easily can be integrated into existing CORBA distributed systems, which is a key feature of our framework for cooperation support.

As detailed in section 3.3, we identified four main services: a cooperation space service (to manage the local database of an activity), a workspace service (to allow legacy applications to access shared data as files or directories), a protocol service (to set the cooperation pattern to be used to cooperate with a given activity), and a coordination service (to ensure that data exchanges between activities are controled by the right pattern). Obviously, the core of our framework is the service related to protocol management. It provides the following functionalities:

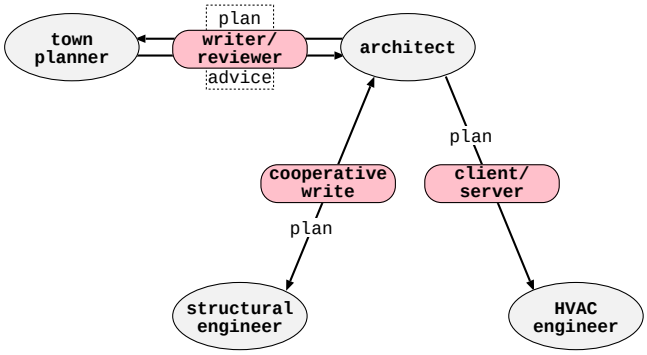
- definition of basic cooperation patterns
- pattern negotiation between activities
- management of the cooperation table
- control of data exchanges
- detection and notification of conflicts; this includes the ability to find and explain what caused a conflict

As explained in section 3.2.2, in order to ensure that its data exchanges with other activities respect the negotiated cooperation patterns, each activity logs all these transfer operations into its local history. Moreover, a cooperation pattern is define by a set of properties on this history. To check the correctness of a sequence of operations (the history) according to a given property, we chose to implement such a property with a Prolog predicate defined on a list of operations. Thus, a cooperation pattern is mapped to a set of Prolog clauses. Even if this solution is not the best efficient way, this allows us to have an implementation of the cooperation patterns close to the correctness criteria.

5. Example

To illustrate our approach, we reuse the example developped in [5]. It consists in designing a one-storey apartment containing a living room with a glass wall. Four kinds of designers cooperate to achieve this work:

Figure 5.1 Document Exchanges and Cooperation Patterns



- the *architect*: his activity is to design and represent the apartment spatial organization with its walls, windows,... To construct his plan, the architect only takes care of volumes, spaces and luminosity of the apartment.
- the *structural engineer*: his activity consists in specifying the structural elements of the apartment. Such elements (cross walls, beams,...) will be chosen to respect, as far as possible, the choices made by the architect and the overall harmony of the building. Such an activity leads to modify the plan provided by the architect.
- the *HVAC engineer*: he will intervene to change the glass wall according to the climate and the apartment exposure.
- the *town planner*: he controls the architect's plan and gives advice in return. The architect has to consider

this advice and possibly to modify his plan according to it. The town planner has to validate the final plan. The town planner only takes care of town planning aspects of the apartment.

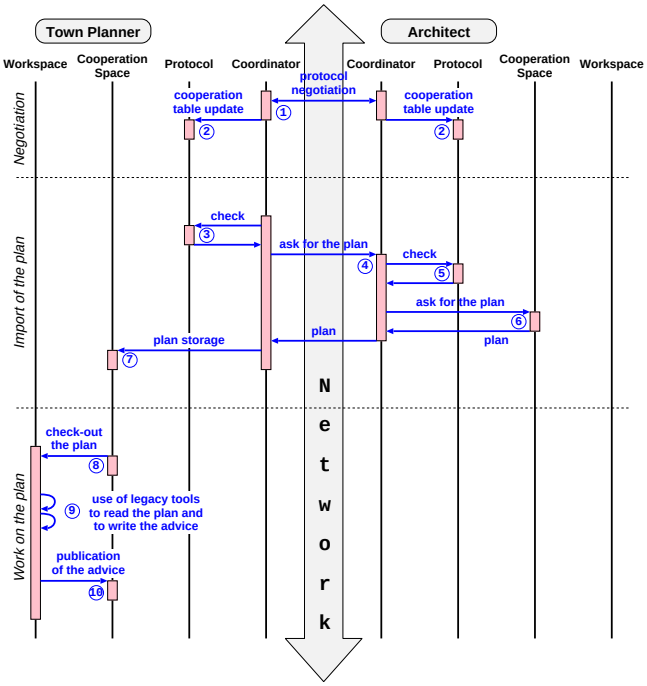
These activities operate on various documents they will share: the plan and the advice of the town planner. All these exchanges between activities will not be driven by the same set of rules. If in some cases we will encourage activities to cooperate, in some others we will put some constraints. For example, the architect and the town planner share both the plan and the advice, but in a read-only mode. It is obvious that the architect will not be allowed to modify the advice delivered by the town planner, and vice-versa. Cooperation patterns used to control exchanges between the various activities are presented below and illustrated in figure 5.1:

- *client/server*: the architect (the server) provides several versions of the plan to the HVAC engineer (the client). Thus exchanges occur from the architect to the HVAC engineer.
- *writer/reviewer*: the town planner reads (but does not modify) the plan provided by the architect. Then he transmits his advice to the architect that reads (but does not modify) this document and modifies his plan according to it.
- *cooperative write*: both the architect and the structural engineer work on the same plan. During the design activity, they will possibly modify it at the same time, so they will have to merge changes made by the other in order to produce only one version of the plan on which they will agree at the end of the acitivity.

Here is an execution example where the town planner imports the plan drawn by the architect and reviews it to produce its advice (cf figure 5.2).

1. When two activities want to share some data, they first negotiate a cooperation pattern to coordinate their exchanges.
2. At the end of the negotiation, each activity updates its own cooperation table with the negotiated cooperation pattern.
3. Now, let us suppose that the town planner wants to import the current version of the plan drawn by the architect. Before to send the request to the architect, the coordinator of the town planner checks that its own protocol allows this operation.
4. Then it asks the coordinator of the architect to send to it the current version of the plan.

Figure 5.2 Protocol Negotiation & Import of the Plan



5. When the coordinator of the architect receives the request, it checks it can give the plan to the town planner.
6. Then it retrieves the plan from its cooperation space and sends it to the town planner.
7. When the coordinator of the town planner receives the plan, it stores it in its cooperation space.
8. Now the town planner can check the plan out of his cooperation space (i.e. the object $plan$ in his local database) to get a regular file (in the DXF format for instance).
9. Then the town planner works in his workspace using his legacy tools, i.e. he can use his favorite CAD software to read the plan and his favorite text editor to write his advice.
10. Later, the town planner publishes[4] his advice, i.e. he converts the file into an object in his cooperation space. Now, the architect can import this advice using the import process described above (steps 3 to 7).

We can note that neither the architect nor the town planner trusts the other. Each of them systematically asks its own protocol to know if the requested exchange is allowed with regard to its negotiated cooperation patterns.

[4]The town planner can perform this operation during the review process, and thus can make visible preliminary versions of his advice.

To conclude this example, we show on figure 5.3 cooperation tables built for each activity of this example.

Figure 5.3 Cooperation Tables for the Example

activity	*remote activity*	*data*	*pattern*	*role*
architect	struct. engineer	plan	cooperative write	$\sim$
architect	HVAC engineer	plan	client/server	server
architect	town planner	plan	writer/reviewer	writer
architect	town planner	advice	writer/reviewer	reviewer

activity	*remote activity*	*data*	*pattern*	*role*
town planner	architect	plan	writer/reviewer	reviewer
town planner	architect	advice	writer/reviewer	writer

activity	*remote activity*	*data*	*pattern*	*role*
struct. engineer	architect	plan	cooperative write	$\sim$

activity	*remote activity*	*data*	*pattern*	*role*
HVAC engineer	architect	plan	client/server	client

6. Conclusion

This paper has introduced our approach to build distributed cooperative applications. Whereas most distributed systems on a client/server architecture, our approach uses a peer-to-peer architecture in which an activity is viewed as a self contained component that cooperates with other components by exchanging, during its execution, some (possibly preliminary) results. Before being allowed to exchange some data, two activities have to negotiate a cooperation pattern first. As an activity could share the same data with several activities using various patterns, we have to deal with interactions between cooperation patterns.

If this approach is ambitious and difficult to achieve in its globality, our experience [7, 10, 11] demonstrates that it is feasible with a limited set of cooperation pattern (client/server, cooperative write, writer/reviewer). Our objective now is to extend this approach by incorporating new cooperation behaviors, i.e. by defining new cooperation patterns in order to better cover the characteristics of a lot of concurrent engineering applications. The more difficult problem we tackle is to find out new correctness properties to validate the integration of the new patterns corresponding to the new cooperation behaviors.

References

[1] Larry Allen, Gary Fernandez, Kenneth Kane, David Leblang, Debra Minard, and John Posner. ClearCase MultiSite: Supporting geographically-distributed software development. In Jacky Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science, pages 194–214. Springer-Verlag, October 1995.

[2] Atria Software Inc. ClearCase product summary. Technical report, Atria Software Inc., 24 Prime park Way, Natick, Massachusetts 01760, 1994.

[3] Goodman N. Beeri C., Bernstein P.A. A model of concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, 1989.

[4] N. Belkhatir and J. Estublier. ADELE–TEMPO : An Environment to Support Process Modelling and Enaction. In J. Kramer A. Finkelstein and B. Nuseibeh, editors, *Software Process Modelling and Technology*. Research Study Press, 1994.

[5] K. Benali, M. Munier, and C. Godart. Cooperative models in co-design. In *International Conference on Agile Manufacturing (ICAM'98)*, Minneapolis, USA, June 1998.

[6] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing surveys*, 13(2):186–221, 6 1981.

[7] G. Canals, P. Molli, and C. Godart. Concurrency control for cooperating software processes. In *Proceedings of the 1996 Workshop on Advanced Transaction Models and Architecture (ATMA'96)*, Goa, India, 1996.

[8] P.K. Chrysanthis and K.Ramamritham. Synthesis of Extended Transaction Models. 19(3):451–491, 1994.

[9] Continuus/CM. Change management for software development. Technical report, http://www.continuus.com/developers/developersACED.html.

[10] C. Godart, G. Canals, F. Charoy, P. Molli, and H. Skaf. Designing and Implementing *COO*: Design Process, Architectural Style, Lessons Learned. In *International Conference on Software Engineering (ICSE18)*, 1996. IEEE Press.

[11] P. Molli. COO-Transaction: Enhancing Long Transaction Model with Cooperation. In *7th Software Configuration Management Workshop (SCM7), LNCS*, Boston, USA, May 1997.

[12] J. Elliot Moss. *Nested Transactions: An Approach to Reliable Distributed Computing,*. PhD thesis, MIT, 1981.

[13] OMG. The Common Object Request Broker: Architecture and Specification. Technical Report 2.0, Object Management Group, 1995.