A Protocol for Building Consensual and Consistent Repositories

Jérôme Euzenat

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A protocol for building consensual and consistent repositories

Jérôme Euzenat

N° 3260

Septembre 1997

THÈME 3

Rapport de recherche

ISSN 0249-6399

A protocol for building consensual and consistent repositories

Jérôme Euzenat*

Thème 3
Interaction homme-machine, images, données, connaissances

Rapport de recherche n° 3260 Septembre 1997 — 46 pages

Abstract: Distributed collaborative construction of a repository (e.g. knowledge base, document, design description) requires tools enforcing the consistency of the repository and the agreement of all the collaborators on the content of the repository. The CO$_4$ protocol presented herein manages the communication between collaborators in order to maintain these properties on a hierarchy of repositories. It mimics the submission of articles to peer-reviewed journals (except that each change must be accepted by all the participants). The protocol is independent from the nature of the repository and is based on a restricted set of message types. The communication between collaborators is described through a set of rules. The protocol is live, fair and maintains a consistent repository consensual among the collaborators.

Key-words: Computer supported collaborative work — groupwork — knowledge sharing — negotiation — interaction protocol — knowledge communication — consensus.

Version du mardi 23 septembre 1997

* Jerome.Euzenat@inrialpes.fr

Unité de recherche INRIA Rhône-Alpes
655 avenue de l'Europe, 38330 Montbonnot Saint-Martin (France)
Téléphone: +33 (0)476 61 53 66 — Télécopie: +33 (0)476 61 52 07

Établissement public national à caractère scientifique et technologique - Décret N°85.831 du 2 août 1985

Un protocole pour la description consensuelle et consistante d'artefacts

Résumé : Le travail en réseau sur un même artefact (base de connaissance, document, etc.) demande des outils pour gérer la consistance de l'artefact et s'assurer que les modifications qui y sont apportées soient acceptées par tous les participants. Le protocole de CO4 présenté ici permet de gérer la communication entre les participants de manière à maintenir ces propriétés sur une hiérarchie de représentations de l'artefact. Il est fondé sur la soumission d'un article à une revue scientifique (mais ici, la contribution doit être acceptée par l'ensemble des participants). Le protocole est indépendant du type d'artefact et repose sur un ensemble restreint de messages. La communication entre les différents participants est décrite en détail par un ensemble de règles. Le protocole est vivace, équitable et maintient une représentation de l'artefact consistante et consensuelle parmi les participants.

Mots-clé : Collecticiel — travail collaboratif — partage de connaissance — négociation — protocole d'interaction — transmission de connaissance — consensus.

A protocol for building consensual and consistent repositories

Groupwork (for individuals, firms or groups themselves) can be favoured through telecommunication and computer advances. Telecommunication allows for communication of and distributed access to the same artefact (a text, a contract, a product design, a knowledge base, etc.); computer science allows the manipulation of such an artefact through more and more sophisticated tools and is able to deal with many aspects of the cooperation process (coherence, broadcast and security). The development of software agents let us foresee a kind of groupwork addressed to both humans and programs.

Our aim is to enable people to find an agreement on the construction of a particular artefact such as a knowledge base, a good design or an article. One can imagine several laboratories and firms grouped together with the aim of building and maintaining an encyclopaedic knowledge server about a particular domain (whose contents can be text and images for instance). For that purpose this group can be implemented through one or several software agents aimed at collecting data and distributing it to those who are allowed to consult it.

This "computer as medium" idea could be extended towards the knowledge itself (i.e. the declarative representation of executable knowledge). For that purpose, a computer environment called CO4 (for collaborative construction of consensual knowledge) is presented in [Rechenmann 1993, Euzenat 1995]. CO4 is dedicated to the incremental and concurrent building of a knowledge base organising a set of various annotations around formalised knowledge. The annotation can be text, bibliography, image, experimental data which the knowledge originates from.

Yet, such a knowledge base must evolve with research results. So, CO4 provides users with support for, on one hand, expressing, annotating and manipulating their knowledge, and on the other hand, making it available to other people. However, if the server can be modified not by only one authority but by any member of the group, new problems arise: under what policy can one modify the knowledge base? Which software programs can implement these policies?

The present paper provides answers to these questions by describing the CO4 protocol.

The CO4 protocol implements a consensual decision policy (i.e. a policy in which a modification, in order to be accepted, must have been agreed by all the other members). The principles underlying CO4 are derived from those of peer-reviewed journals: before introducing it into a consensual repository, the knowledge must be submitted and accepted by the whole community. For that purpose, knowledge is submitted to the repository, reviewed by the other participants and accepted or amended according to their reactions. The informal knowledge (text, pictures, etc.) is also subject to submissions, reviewing, etc. Thus finally, the knowledge stored in a consensual knowledge base is safe enough so that anybody can use it confidently and easily. This protocol works at several levels: the group repositories can be recursively grouped together into a more important group base and so on, so forth. However, the behaviour of such a group base is still subject to the consensual approbation of its subscribers. For instance, a consensual representation could be achieved inside a particular firm before being submitted to the inter-institution repository.

The presentation below answers to the second question (implementing the protocol). The formalism used for describing the protocol has been defined with regard to the needs for a reactive multi-agent protocol. It has been kept simple: rules are triggered by a single event (identified by the class of the sender and the type of the message) and simple side conditions. The reaction consists in sending other messages and manipulating ordinal counters and sets. Its detailed description allows a non ambiguous understanding of the protocol and eases its modification by adding and/or modifying a single rule. Moreover, it provides several advantages:

- the protocol has been described in the LOTOS language and went through extensive model checking [Pecheur, 1997];
- the LOTOS description allows to generate simulators (used in §5);
- several properties are analytically proven below (§6);
- its implementation has been easily achieved as a Unix library.

The protocol has also been kept flexible, extensible and general but non trivial. It is not a toy protocol: it is made of 55 rules which cover the aspects of administration — registering, errors, notification —, voting — including denying and challenging proposals —, broadcasting, etc.. This can be compared with the “contact net” protocol [Smith 1980] modelled with only five rules in a language of very similar power [Gaspari 1997].

Since the work presented here has been designed in the context of knowledge base construction by a team of researchers in some scientific field, examples will be given in that perspective. However, CO4 has been designed independently from the kind of artefact under construction. Once accepted, a modification is applied to the concerned artefact through the usual manipulation software for that artefact, the only constraint being that the artefact must be described in a digital format in some kind of repository accessible with few primitives. The advantage brought by this generality is the ability to apply and experiment the protocol on various applications such as computer aided-design, collaborative hypertext documentation or the redaction of a corporate memory.

The consensual aspect of the protocol does not restrict its applicability: it should be quite easy to change the policy from “consensual” to “majority” or “intersection”. However, the properties would not be the same.

The paper first introduces the simple social organisation of the repositories (§1). Then the general communication policies and notations are described (§2) before presenting the complete protocol from the viewpoint of individual repositories (§3) and consensual repositories (§4). An extended trace is then given for demonstrating the potentiality of the protocol (§5) and the main properties enjoyed by the CO4 protocol are presented and proven (§6). Finally, several features of the protocol are discussed (§7) and the protocol is compared with related work (§8)

1. Overall architecture and presentation

In order to specify CO4, requirements for the software manipulating repositories and communicating between them must be designed. The specification of the software is presented below together with the way it is used either manually or automatically.

The organisation of repositories in order to contribute to a consensual repository is first introduced (§1.1) before providing the principles of the CO4 protocol form the viewpoint of the individual users (§1.2) and from that of consensual repositories (§1.3).

1.1 The network of repositories

In CO4, any cooperator is viewed by the system as a repository. In order to build a consensual repository, the individual repositories must be linked together. Repositories are organised into a tree whose leaves are user repositories and whose intermediate nodes are called group repositories (see Figure 1). Each group repository represents the knowledge consensual among its sons (called subscriber repositories). This structure imposed to the collaboration can be stuck on the structure of a particular firm or that of a particular group in the firm, but it can also be independent from that structure. A repository can subscribe to only one group. A human user can create several repositories (possibly subscribing to different group repositories) representing different trends, and knowledge can be transferred from one repository to another. Also, nothing prevents several human users from sharing the same repository.

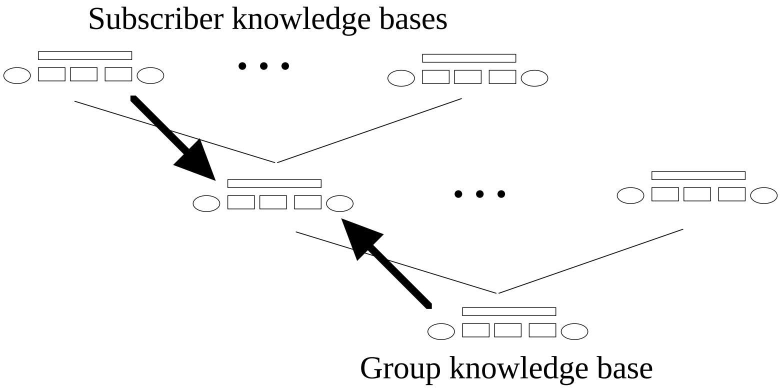

Figure 1. The hierarchical architecture and message flow (dark arrows). The repositories are organised in a tree whose leaves are individual repositories and nodes represent the consensus between of connected individuals. The downward types of messages include the submission of a proposal and the reports of approval, rejection or counter-proposal about a submitted proposal. The upward messages include the broadcast of accepted proposals and the call for comments (ask-all) about a submitted proposal.

Individual users can subscribe to a consensual repository by sending a register request which has to be accepted by the group repository (thus by the current subscribers). Upon acceptance the respective repository definitions of the group repository and the individual repository are aware of each other. As soon as the repository is part of a group repository, it receives the complete contents of that repository (that it is supposed to accept). It is also entitled to give its opinion on all submissions currently under examination and is allowed to submit knowledge. The interesting point is the submission of knowledge which is described right below (§1.2 and 1.3).

Some independent repositories can subscribe to group repositories as observers: these repositories are sent by the group repository whatever is introduced in the base but cannot modify it. Observers are not further considered in this presentation.

A group repository sends to its subscribers messages for broadcasting a change accepted by everyone and calls for comments in order to establish whether a change must be committed or not. A (group or individual) repository sends to its group repository changes which it wants the group repository to integrate. Of course, any group repository, as an individual repository also receives calls for comments and change broadcast.

Group repositories have the same structure as individual ones: they are made of the same pieces of software. The main difference between group and individual repositories is that the former are completely automated and only respond to stimuli from other repositories: they do not require human assistance. The architecture of repositories is depicted in Figure 2. It consists of two layers. The communication layer connects the repository to its group repository and its subscribers (or a user interface in order to be manipulated by the user). The repository layer contains four components:

- The storage of the repository content (called K below) which is manipulated by usual software for that content (called "Update and revision controller"),
- The cooperation controller with its repository definition (corresponding to all the other variables in §2.3) which implements the CO4 protocol, and
- The "Negotiation controller" which manages the interaction between the artefact management software and the protocol by calling the usual software (§2.3) and interacting with the other bases or with the user (for answering call for comments or committing the changes into the knowledge base).

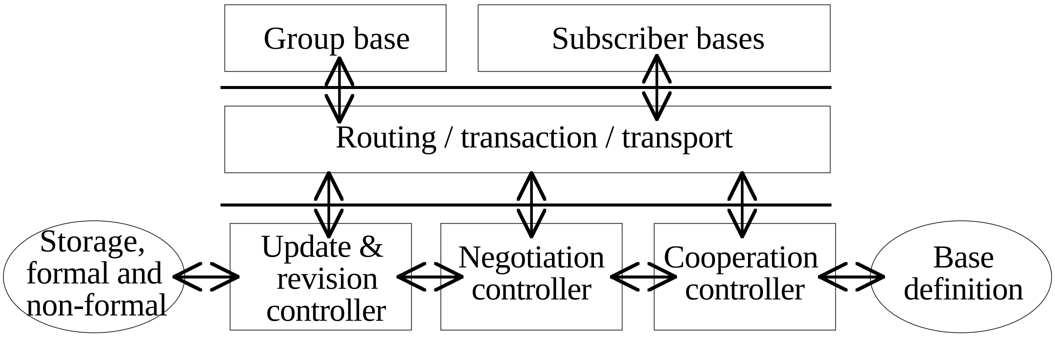
Group base
Subscriber bases
Routing / transaction / transport
Storage, formal and non-formal
Update & revision controller
Negotiation controller
Cooperation controller
Base definition

Figure 2. The repository definition contains the situation of a repository inside the tree of repositories. It thus stores the addresses of group (g hereafter) and subscriber repositories (S). When a problem occurs, the negotiation controller has the choice between displaying it on a HTML browser (for individual repositories) or passing it to the cooperation controller which sends it to the subscriber repositories (for group repositories).

The present paper only deals with the cooperation controller and the repository definition. Thus the presentation is abstracted from the inter-repository communication problems and the knowledge representation issues. The protocol is, in fact, independent of the knowledge representation aspects and the kind of artefact considered. These aspects are seen through an abstract interface (see §2.3) which enables the use of the manipulation software capacities (in terms of inconsistency checking, or querying, for instance).

1.2 A glimpse at submission from the user viewpoint

When subscribers are confident enough with some pieces of knowledge, they can submit them to the group repository to which they subscribe. This is achieved by circumscribing the submitted part and calling the submission procedure of the negotiation controller. In order to complete the submission message, the negotiation controller collects the sets of differences between the consensual group repository connected and the selected changes (logged by the revision controller) and sends them to the group repository. Usually, the group repository, through its own revision and negotiation controllers, issues a report describing how the submitted knowledge can be added to the group repository. Thus, the user can eventually choose a better (and consistent) way to achieve the submission. This proposal is then submitted to the other subscribers and finally committed if it reaches consensus.

As subscriber of a group repository, the user also receives the calls for comments issued by the group repository in response to the submission of some material. Users can read the submission or apply it in their own repository by submitting it to the revision controller. This can result in a favourable agreement report or an inconsistency detection that can be used by the user for issuing a counter-proposal. In response to the call for comments, users must answer by one of the following: accept when they consider that the knowledge must be integrated in the consensual repository, reject when they do not, and challenge when they propose another change.

When the group repository has gathered enough comments, it integrates, or not, the change in the repository. If the change is consensual, it is broadcast to all subscribers. It may happen, however, that the research they are currently involved in contradicts what is in the group repository. So users can refuse the new piece of knowledge (just as they can modify parts of the group repository knowledge in their local repository) which is then stored in a change logbook for further change submission.

The fact that anyone can maintain a repository different from the consensus obviously allows the exploration of concurrent paths. On a more basic ground, this enables communication, negotiation and acceptation to be asynchronous. It reproduces the way papers are submitted, discussed and accepted or rejected in a scientific journal: reviewers can take time for carefully examining a proposal since it will not stop the work of the repository which issued it.

1.3 Paper submission metaphor as implemented by consensual repositories

Any system allowing to build some artefact must have a particular change policy. The CO$_4$ protocol mimics that of editorial boards: before being introduced in a consensual repository, knowledge must be submitted and accepted by the community. To our knowledge, the peer-reviewing protocol [Peters 1995] has never been used for building knowledge bases. The choice of such a protocol is not innocent: it proved to be practicable within the scientific community and, in the consensual version, it enforces the dialogue between people (rather than a simple majority or intersection protocol).

Consistency and formality require more strictness in the protocol than pure peer-reviewing; this leads to the consensus requirement (i.e. in which, a modification, for being accepted, must have been agreed by all other members — for instance, in the context of genome sequencing, a consensus map is a map that all the people involved in the research field think correct). Integrating knowledge requires its submission to the repository.

When a group repository receive a submission it issues a call for comments towards all of its subscribers. Among the answers provided by the subscribers, three cases may happen:

- All of them agree on the modification acceptance, then the modification is committed into the group repository and broadcast to every subscriber repository;
- One of them rejects the proposal, then the changes are not committed and the comments provided by the rejecter are sent to the submitter (the call for comments is cancelled);
- One submitter sends a counter-proposal, then the call for comments is replaced by a call for comments about all the available proposals (those who already accepted the change, are asked to consider the new proposal and to answer again).

It can also happen that the submitter retracts the proposal thus leading to the retraction of the call for comments from all the repositories.

The CO4 protocol applies to several levels: the group repositories can be grouped together into a more important group repository and so on. However, the behaviour of such a group repository is still subject to the consensual approbation of its subscribers. Thus, for instance, a consensual representation could be achieved inside a particular firm before being submitted to the inter-institution repository.

In order to implement these proposals, the communication between repositories must be specified so that they understand each other. This is the purpose of the CO4 protocol. In order to describe it precisely and formally in §3 and 4, some definitions and overall principles are given in the next section.

2. Communication

The protocol is presented first through the main conversation policies (the protocol skeletons) implemented in the CO4 protocol (§2.1). Then, the notations and languages required for expressing the exchanged messages are described, the details of the protocol are introduced (§2.2) and the data structure manipulated by the protocol are presented (§2.3).

2.1 Conversation policies

Conversation policies are presented as diagrams intended to express how a query from a repository can be processed by the others. This is a very general and synthetic description of what happens. They are implemented in section 3 and 4 through a set of rules which specifies the initiation of a policy and the chaining of messages. These policies, in CO4, are reduced to only two schemes depending on which repository the initiative comes from: from the group repositories to the subscriber repositories or the other way around.

A policy can be schematised by a picture and a table. In the picture (see Figure 3), arrows represent messages; numbers labelling them are a stratification of their occurrence order. All the messages carrying the same number must have been sent before the arrows carrying the successor can be instanciated. Table 1 provides the type of each message in each instance of the policy (e.g. for subscribing or submitting a piece of knowledge the type of message initiating the conversation is not the same). The arrows may or may not be instanciated. Moreover, additional communication may happen between two stages (for instance a group repository which receives a call for comments, initiates its own call, and replies to the initial call for comments only when the latter reaches completion).

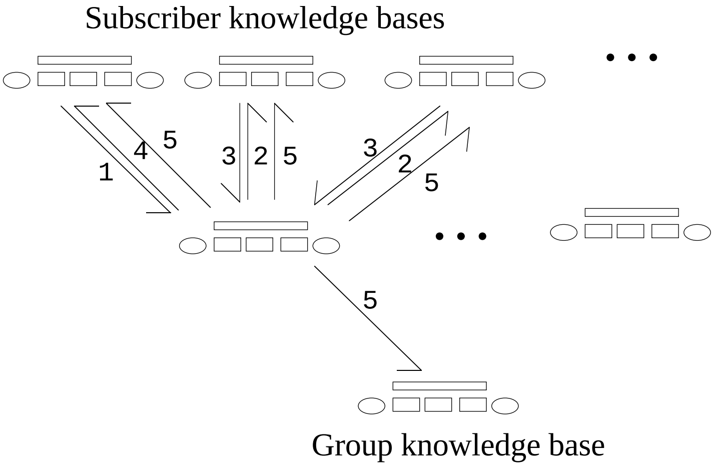

Figure 3. Downward policy: how a group repository processes an initiative from a subscriber (see Table 1).

phase	subscribe	submit	forward
1: query	`register`	`achieve`	`forward(P)`
2: call	`ask-all`	`ask-all`	`ask-all`
3: vote	`reply`	`reply`	`reply`
4: report	`notify/pnotify`	`notify/pnotify`	`pnotify`
5: commit	`tell`	`tell`	`P`

Table 1. Downward policy instanciations.

For downward policy, there are 5 stages which can be instanciated in three distinct processes. For instance, the submission is achieved through (1) a repository sending an `achieve` message to its group repository, (2) a call for comments emitted with the `ask-all` message from the group repository to its subscribers, (3) a `reply` from the subscribers to the group repository accepting or rejecting the proposal (which corresponds to their vote for or against the proposal), (4) a notification of the issue to the voters and (5) the introduction in the group repository of the proposal and a broadcast of this to all the subscribers with the `tell` performative.

The same stages are found in the upward policy (see Figure 4 and Table 2) though they do not correspond to the same set of arrows. For instance, a broadcast is achieved through (1) the `tell` message considered above, (2) a call for comments to all the subscribers (for group repository), (3) the same `reply` as above from the subscribers, (4) nothing in that case and (5) the introduction or not in the repository of the content of the message and its broadcast to the subscribers (through a new `tell`).

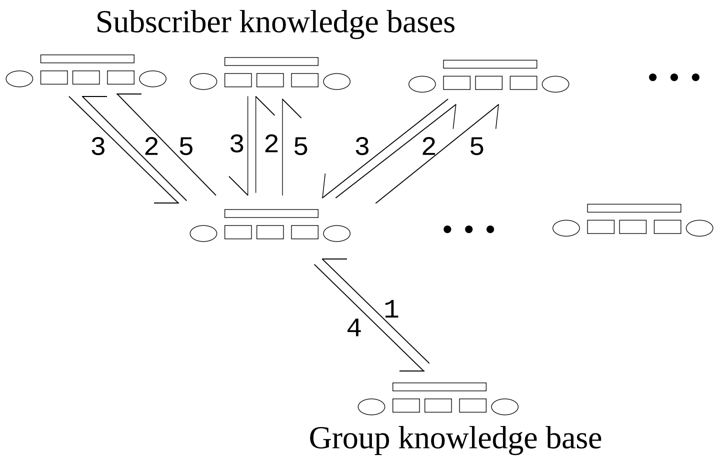

Figure 4. Upward policy: how a group repository processes an initiative from its own group repository (see Table 2).

upward policy	call	broadcast	notify/pnotify
1: query	ask-all	tell	notify/pnotify
2: call	ask-all	ask-all	notify/pnotify
3: vote	reply	reply	
4: report	pnotify	pnotify	
5: commit	reply	tell	

Table 2. Upward policy instanciations.

2.2 Messages and rules

Repositories communicate only through message-passing. The messages are sent and received asynchronously (within a finite delay, so the communication layer is supposed to be reliable). Their reception order is independent from the emission order. The system is thus an asynchronous message passing system [Fagin& 1995].

The messages sent from one repository to another are expressed through a speech act (loosely inspired from "speech act theory"). This notion has several advantages for the particular architecture presented here:

- It allows the separation of knowledge from its use (its addition or retraction from a particular repository, for instance);
- It is independent from the representation language and the protocol can thus be expressed abstractly and the library implementing it can be generic;
- A speech act can refer to another speech act (retracting a submission, for instance).

The inter-repository communication uses an extension of KQML — Knowledge Query and Manipulation Language [Finin& 1993] — a language defined as part of the American DARPA knowledge sharing effort. KQML enjoys three interesting aspects: (1) Coming from speech act theory it carefully distinguishes between the contents (and its meaning) from why this contents is communicated — this is invaluable in our context —, (2) it covers our needs, and (3), there is some software for it already implemented. Initially, KQML (and its spirit) has been chosen for these reasons and in order to be interoperable with other kinds of agents. However, the ongoing revisions of the language [Cohen& 1995; Labrou& 1997] are made with a so precise kind of application in mind that they prevent it to become a general purpose agent communication language. Hence, it can be considered that the performatives used here are not anymore KQML. This does not weaken the presentation since the rules given below provide a formal semantics for the messages used. Other languages could have been used (e.g. the MAIL language [Haugeneder 1994] providing the message types for `propose`, `refine`, `modify`, `accept`, `reject` and `tell`).

Messages are expressed as a collection of performatives (the type of a speech act): `achieve` (submit a proposal for inclusion into the consensual repository), `ask-all` (ask for the of subscriber repositories, call for comments), `accept` (accept the insertion of the piece of knowledge), `reject` (reject it), `challenge` (submit a concurrent proposal), `retract` (the submitter retracts the submission), `tell` (send an accepted proposal to each subscriber repository).

The actual protocol uses the whole set of KQML message keywords (but `force`). The messages thus have the following shape:

```
(performative
  :sender        % the repository sending the messages
  :receiver      % the repository to which it is sent
  :reply-with    % the surrogate of this message
  :in-reply-to   % the surrogates of the message to which this one answers
  :language      % the language in which the contents is expressed
  :ontology      % the ontology to which the contents refers
  :content       % the actual content of the message, may be another message
)
```

The usual values for language are the language used for representing knowledge (call it KRL), KQML (the contents is another KQML message), Co4 (it is a message expressing a CO4 performative). The ontology is assumed to be the same throughout the paper. The message types used by CO4 are the following:

performative	language	content	informal meaning
`register`	–	`g,B`	S intend to subscribe to the group repository R
`achieve`	`krl`	`k`	S submit a proposal C to R
`evaluate`	`krl`	`k`	S want R to simplify C
`ask-all`	`kqml`	`p`	S ask the comment of R on C
`tell`	`krl`	`k`	the group repository S tells R that it contains C
`reply`	`co4`	`r`	S reply to the R message identified by N
`forward`	`kqml`	`p`	S wants R to send the message C
`deny`	`kqml`	`m/p`	S retract the message to R identified by N

Table 3. Message types used in the protocol (S is the sender, R the receiver, C the content and N a message identifier; see the text for more information).

The content of the performatives can be classified in several categories which are denoted below by different letters: (k) corresponds to knowledge, i.e. syntactic expressions which can be inserted in the repository content; (p) corresponds to proposals which are subject to vote before being accepted at the level of a group repository (this concerns, achieve(k), forward(p) and register(g,B)); (r) corresponds to reply to a call for comments, i.e. accept, reject (with explanations) and challenge (with challenging proposals); (m) correspond to the other KQML messages (ask-all(p), tell(p), evaluate(k), reply(r)). Only the knowledge (k) can be challenged in this specification.

Other performatives have been introduced which are not standard KQML performatives. Thus, in order not to burden KQML with performatives and to stress the fact that the implementation of the performatives is dependent of CO4, they are labelled as CO4 performatives:

performative	language	content	informal meaning
`accept`			S accepts the proposal of R identified by N
`reject`	`krl`		S rejects the proposal identified by N
`challenge`	`krl`	`k,k`	S sends to R a counter-proposal to that identified by N
`notify`	`co4`	`r`	S notifies R of the final result achieved by one of its proposal[1]
`pnotify`	`co4`	`r`	S notifies R of the result achieved by a proposal it had to vote for (poll-notify)

Table 4. Non-KQML performative used in the protocol (S is the sender, R the receiver, and N a message identifier).

The cooperation protocol is based on the architecture of the repositories (some of which being group repositories, the other ones being only subscribers) and a complete set of behaviour rules (see below). The formalism used for describing it has been kept simple: rules are triggered by a single event which is identified by the class of the sender and the name of the message, the possible reactions are the sending of other messages and the manipulation of data structures in the repository (basically, ordinal counters and sets). The rules are expressed in the following formalism:

(rule name) $$\frac{s \text{ — } act \rightarrow r}{a_1,\ldots a_n,\ r \text{ — } act' \rightarrow s'}\quad c$$

when r receives message act from s and condition c is satisfied, it performs actions $a_1,\ldots a_n$ on its own state and sends message act' to receiver s' (any part of these rules may be void; empty triggering messages denote user initiative). The execution of the rules in a repository are supposed to be atomic (non interuptible for executing something else) and non concurrent (the repository cannot execute two rules at once). When inter-repository communication is required, the shape of the KQML primitive to be used is described in the formal rules which represent the automatic behaviour of the message receiver. In the former rule the message: s — perf(arg) → r, signifies that the message:

[1] Because of the forward, this (and the following) could be implemented in two ways: notify when the vote ends or wait until the complete forward achieves a status. The latter is implemented for notify and the former for pnotify.

```
(perf :sender s :receiver r :content arg)
```

has been sent.

Expressing the protocol under such rules allows to check that no performative has been forgotten. This has the advantage of being modular: each rule can be considered in itself and the protocol can be modified by only changing a small set of rules, furthermore anyone can check that all the phases are taken into account and the emitted messages can be dealt with.

The protocol has several particularities:

- there is no need for human intervention in the group repositories;
- there is no message but from subscriber repository to group repositories and back;
- every decision has been approved by all the subscribers (and recursively for a group of group repositories).

The actual protocol is routed automatically (once a repository has subscribed to another), the performative and content levels are interpreted automatically in the group repositories and the performative level is automatically interpreted by the individual repositories (however, the system asks the user before committing these performatives).

2.3 Data structures

The data structures used by the repositories are represented in the following table (the last part represents structures found only in group repositories):

Name	Contents	Purpose
A	<surr,msg,cfc-surr,initiator-id>*	set of submitted proposals
P	<surr,msg,state>*	set of proposals to consider
L	Non specified	set of ignored modifications
K	Non specified	repository content
S	id*	set of subscribers name
O	id*	set of observers name
C	<sndr-id,surr,cfc-surr,msg,rply#>*	set of issued call for comments

Table 5. Data structures used in the repositories. Identifiers for other repository (id) allow to communicate with the group repository and subscribers, surrogates (surr) allow to identify message threads and to answer with the correct reference.

The set of current proposals (P) contains a triple with the proposal submitted, the surrogate identifying it, and the status of the answer if it has been provided. A proposal is created when the call for comments is received by the repository; the state is then set to @. When the repository replies to the call for comments, the state is set to the reply (accept, reject, challenge). The proposal is discarded either when the group repository denies the call for comments (indicating that the initial submitter decided to retract it) or when the issue of the vote if given (through the pnotify message).

The set of proposals (A) contains the proposals submitted by the current repository. Such a proposal is created at submission time and contains then the initial submission message and its surrogate. It is discarded whenever an error is sent back, the proposal is automatically accepted (e.g. when no other repository subscribes to the group repository) or the status of the proposal is notified (through the notify message). When such a proposal is submitted by a group repository upon initial submission of one of its subscribers, then this initiator and the initial surrogate of the submission are also recorded in that structure.

The set of call for comments issued by a group (C) contains a quintuple of the initial submitter, the proposal submitted and its surrogate, the surrogate of the call for comments and a structure storing the answers from the subscribers; here, for simplicity's sake, this is restricted to a count of the subscribers which have not answered yet. The call for comments is created when the call for comments is emitted towards the subscribers. It is discarded either when the proposal is retracted by the initial submitter or when the proposal achieves a final status (accepted, rejected or challenged).

These sets are manipulated through usual set operators: union ($\cup$), intersection ($\cap$), complementation (—), membership test ($\in$) and cardinal ($|\cdot|$). A substitution operator (+/) has been added which replaces in a particular member one element by another.

The surrogates for messages are generated by an operator (!) which provides a brand new surrogate. They are always handled by a particular repository which ensures the uniqueness of surrogates. So, each proposal is stamped by a surrogate unique in the context of the issuer. This surrogate is stored in the A directory and in the C directory of the group repository. The group repository answers to the issuer (through error and notify messages) with that surrogate. On the other hand, the voting process is handled by the group repository which generates its own unique surrogates for each call for comments. This surrogate is used by the subscribers for replying the call for comments (through the reply message) and by the group repository for discarding the call for comments (deny) or notifying the result of the vote (pnotify). There are no surrogate collisions in the C and A directories since they are indexed by concerned bases nor in the P directory since all the call for comments there are indexed by a unique group repository.

The repository content (K) has a particular status: it is seen as a repository and the actions are just side-effects on that repository. The structure of the repository is handled by the application software and does not matter here. However, the CO4 protocol must interact minimally with it and for that purpose it recognises three operations:

K: returns the whole contents of the repository;

K+k: adds k (or rather applies k) to the repository;

K?k: reports about applying k to K (in the present context, tells if it is possible to apply k to K).

For the purpose of challenging proposals, it is also assumed that proposals (identified by k) can be split into several smaller proposals and that it is closed under intersection (which can be tested for non-emptyness). Although, the protocol still works without this assumption, the challenge answer must then be disabled.

3. The user and the group

All the actions which can be undertaken by the user are formally described hereafter. They are presented in three groups: the actions that users can perform at any moment (§3.1), the actions that they can only perform when they have been solicited by the group repository (§3.2) and the actions that the software automatically performs when some solicitation comes from the group repository (§3.3). Hereafter, s represents the repository name and g is the name of its group repository when it has one.

3.1 Initiatives

Here are the performatives that an agent can use in order to influence the system. They have in common the absence of triggering message: they only have to fulfil triggering conditions but users can use them whenever they want. This corresponds to phase 1(emission) of the downward policy.

The first initiative consists in registering an individual repository to a particular group [register]. For that purpose, the repository must not been registered neither it must have asked its registration elsewhere. Here the message is sent to a repository b which is not yet in g.

$$\text{[register]}\quad \frac{}{A{:=}A\cup\{<!n,\mathrm{register}(s,b),\_,\_>\},\ s \text{ --- } \mathrm{register}(s,b) \rightarrow b}\quad g=\emptyset,\ A=\emptyset$$

When subscribers are confident enough with some content of their repository, they can submit them to the group repository which they subscribe to. This is achieved by circumscribing the submitted part through the graphic interface and sending the selected changes to the group repository. Usually, the submitter begins by asking a report from the group repository [evaluate]. The repository should compare the contents of the message with that of its repository and return a report on that comparison. Then the user can choose a better (and consistent) way to achieve the submission.

$$\text{[evaluate]}\quad \frac{}{A{:=}A\cup\{<!n,\mathrm{evaluate}(k),\_,\_>\},\ s \text{ --- } \mathrm{evaluate}(k) \rightarrow g}$$

The repository can submit some proposal to be included into the group repository. To that extent, the proposal is sent with the `achieve` performative [achieve].

$$\text{[achieve]}\quad \frac{}{A{:=}A\cup\{<!n,\mathrm{achieve}(k),\_,\_>\},\ s \text{ --- } \mathrm{achieve}(k) \rightarrow g}$$

The forwarding of some work to the group repository (g) asks the group repository to submit the work (usually carried out in the group repository) to its own group repository [forward]. This work includes: submitting knowledge in the group repository to its own group repository, making the group repository subscribing to another group repository, making the group repository forwarding something to its own group repository or asking the group repository to deny something attempted (through a forward). As usual, these performatives are subject to a call for comments and consensual acceptance from all the subscribers of the group repository.

$$[\text{forward}]\qquad \frac{}{A{:=}A\cup\{<!n,\text{forward}(p),\_,\_>\},\ s — \text{forward}(p) \rightarrow g}$$

At any moment — as long as the proposal is under examination — the repository can retract some message it issued [deny]. The group repository should then erase the effects of the previous assertion. This is especially important when, during the review process, a repository realises that the proposal is not worthwhile.

$$[\text{deny}]\qquad \frac{}{A{:=}A—\{<n,\_,\_,\_>\},\ s — \text{deny}(n) \rightarrow g}\ <n,\_,\_,\_>\in A$$

Note that any subscriber can forward a deny to its group repository, since if the previous forward has been accepted it is not sent on behalf of the original sender but on that of the group repository itself. It is not possible to deny a vote message because they are not in A. However, the user can submit a new proposal aiming at retracting the concerned proposal.

3.2 Replies

As a subscriber of the group repository, the user also receives the call for comments issued by the group repository in response to the submissions (by other users) of some material. Users can read the submission or play it in their own repositories by submitting it to the revision controller. This can result in a favourable report or an inconsistency detection that can be used for issuing a counter-proposal. In response to the call for comments, users must answer by one of the following: accepted when they consider that the knowledge must go in the consensual repository, rejected when they do not and challenged when they propose another change.

Here are the performatives that an agent can issue as an answer to a call for comments. The answer is provided by the user initiative (no triggering messages) from the proposal which is stored in the local proposal directory (P). This corresponds to the phase 3(emission) of the downward and upward policies. All answers are sent wrapped into the `reply` performative referring to the initial call for comments surrogate. The acceptance [accept] does not require additional justification.

$$[\text{accept}]\qquad \frac{}{P{:=}P+\{<n,p,@/A>\},\ \ s — \text{reply}(n,\text{accept}) \rightarrow g}\ <n,p,@>\in P$$

The rejection [reject] must provide a comment explaining why the submission has to be rejected.

$$[\text{reject}]\qquad \frac{}{P{:=}P+\{<n,p,@/R>\},\ \ s — \text{reply}(n,\text{reject}(r)) \rightarrow g}\ <n,p,@>\in P$$

The counter-proposal emission only applies to submission (achieve or forward*(achieve)). It divides the initial proposal into two parts: one (k+) which is accepted and the other (k-) which is refused [challenge]. To these two parts, the user can add another one (k*) to be submitted to the target base. So, challenging combines voting and submitting.

$$\text{[challenge]}\ \frac{}{\begin{array}{c}P{:=}P+\{\langle n,\ \mathrm{forward}^*(\mathrm{achieve}(k)),@/C\rangle\},\\ A{:=}A\cup\{\langle !m,\mathrm{challenge}(k\text{-},k\text{+},k^*),\_,\_\rangle\},\\ s \text{ — } \mathrm{reply}(n,\mathrm{challenge}(m,k\text{-},k\text{+},k^*)) \rightarrow g\end{array}}\ \begin{array}{c}\langle n,\mathrm{forward}^*(\mathrm{achieve}(k)),@\rangle\in P,\\ k\text{-}\neq\emptyset,\ k\text{+}\neq\emptyset\end{array}$$

3.3 Automatic parts of individual repositories

Here are the performatives that an agent can receive and which are automatically processed (for later examination by the user) into the individual repository. They are necessary in order to let the users decide when they consider the messages which are sent to the repository. Thus all these transactions are buffered by the individual repository in order to be processed later on (see the “replies” section). This corresponds to phase 2(reception) and 5(reception) of the downward and upward policies.

Whenever a proposal is submitted for approval, the proposal is stored into the local proposal directory (store-cfc). The user can reply later to the call for comments (accept, reject, challenge).

$$\text{(store-cfc)}\qquad \frac{g \text{ — } \mathrm{ask\text{-}all}(n,p) \rightarrow s}{P{:=}P\cup\{\langle n,p,@\rangle\}}$$

The first answer concerns the notification of registration. Then, the group repository is stored in the g variable.

$$\text{(notify-register)}\ \frac{b \text{ — } \mathrm{notify}(n,\mathrm{accept}) \rightarrow s}{A{:=}A\text{—}\{\langle n,\mathrm{register}(s,b),\_,\_\rangle\},\ g{:=}b}\ \begin{array}{c}\langle n,\mathrm{register}(s,b),\_,\_\rangle\in A\\ g=\emptyset\end{array}$$

Other notifications (errors or issues of a call for vote) are handled by the following — technical — rule which only flushes the A table (however, in case of individual repositories, this table could be flushed by the user).

$$\text{(handle-notify)}\qquad \frac{g \text{ — } \mathrm{notify}(n,r) \rightarrow s}{A{:=}A\text{—}\{\langle n,\_,\_,\_\rangle\}}\ \langle n,\_,\_,\_\rangle\in A$$

At the end of the vote (or during it when the submission is denied by its initiator), the outcome is broadcast to the voters so they can flush their P table (again this can be left to the users in individual repositories).

$$\text{(handle-pnotify)}\qquad \frac{g \text{ — } \mathrm{pnotify}(n,r) \rightarrow s}{P{:=}P\text{—}\{\langle n,\_,\_\rangle\}}\ \begin{array}{c}\langle n,\_,\_\rangle\in P,\\ r\neq\mathrm{challenge}(\_,\_,\_,\_)\end{array}$$

Whenever a proposal is accepted, its contents is stored in the local proposal directory (store-proposal). When the group repository has gathered enough comments, it integrates or not the change into the repository. The change being now consensual, it is broadcast to all the subscribers. It may happen, however, that the current view that the individual user has of its repository is not totally coherent with what is in the group repository. So the users can refuse that new knowledge (just as they can also modify parts of the group repository contents in their local repository) which is then stored in a change logbook (L) for further change submission (of course the users can also select a p in L in order to put it into their repository).

$$\text{(store-proposal)} \qquad \frac{g \text{ — } \mathrm{tell}(p) \rightarrow s}{L \cup \{p\}}$$

4. The submission protocol

The group repository manages the negotiation for applying a change (issuing the call for comments, receiving the answers, managing the counter-proposals and committing or retracting the changes). This requires the identification of the change by a unique number, the count of positive answers, the management of counter-proposals and retractions of a proposal. This also requires the recording of the process in order to recall the reasons why some change is made or not. The group repositories blindly apply the negotiation rules given below and the final decision comes from the users who hold the leaves of the architecture. When a message is issued by a group repository whose subscribers are also group repositories, these last repositories only dispatch the messages to their subscribers. Here, g denotes the current group repository, s denotes one of its subscribers (as message sender) and G is its own group repository (if any).

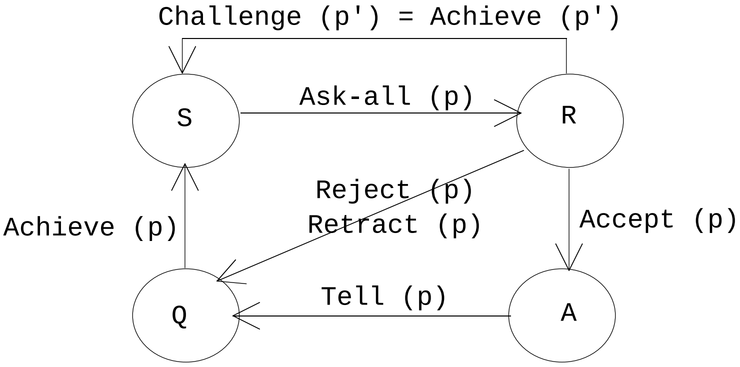

Figure 5. The automaton corresponding to the submission of proposal p at the scale of the whole system of group repository plus subscribers. It reproduces four states of publication submission: initial state (Q), submitted (S), under review (R), accepted (A).

The protocol is summarised in Figure 5 as a finite state automaton. It is noteworthy that the protocol is (1) asynchronous, so that several proposals can be in different states concurrently, (2) parameterised by proposal p, so that the real situation in the CO_4 is the Cartesian product of automata corresponding to all the proposals, and (3) abstracted from the status of each individual repository with regard to the protocol.

The protocol roughly always follows the same policy but the actions performed differ. From now, a performative initiated by an individual repository is taken into account from the viewpoint of the group repository which receives it. It is expressed through two rules: the first one deals with the issuing of a call for the proposal and the second one deals with what to do when the proposal is accepted. This corresponds to phases 1(reception) and 5(emission) of the downward policy for the subscribe, achieve, and forward protocols.

4.1 Subscription

The user of a workstation can subscribe to a consensual repository. This is achieved very simply through the repository definition controller which manages the description of the group to which the repository subscribes (and for group repositories, their set of subscribers). This repository definition enables the communication layer to route queries from one repository to its group repository and its subscribers. It is intended that it also describes the topics which the repository is interested in, etc.

When a group repository receives a subscription query, it issues a call for comments towards the already registered subscribers (cfc-register). This call is handled by the usual rules (reject-reply, accept-reply, challenge-reply).

(cfc-register)

$$\frac{s \text{ — } \mathrm{register}(n,s,g) \rightarrow g}{\begin{array}{c} C{:=}C\cup\{<s,n,!n',\mathrm{register}(s),|S|>\},\\ g \text{ — } \mathrm{ask\text{-}all}(\mathrm{register}(s),\ n') \rightarrow S \end{array}}\quad S\neq\emptyset$$

However, if the group repository has no subscriber, it will always accept a new subscriber (reply-register).

(reply-register)

$$\frac{s \text{ — } \mathrm{register}(n,s,g) \rightarrow g}{\begin{array}{c} S{:=}S\cup\{s\},\ g \text{ — } \mathrm{notify}(n,\mathrm{accept}()) \rightarrow s,\\ g \text{ — } \mathrm{tell}(K) \rightarrow s \end{array}}\quad S=\emptyset$$

When the subscription is accepted, the group repository erases the call for comments from C. Then it adds the new repository to the subscriber set, sends it a notification of registration, the whole repository contents and all the current calls. As soon as the repository is part of a group repository, it receives the complete contents of that repository and is entitled to give its opinion on all the submissions currently under examination.

(accept-register)

$$\frac{s' \text{ — } \mathrm{reply}(n',\mathrm{accept}) \rightarrow g}{\begin{array}{c} g \text{ — } \mathrm{pnotify}(n',\mathrm{accept}()) \rightarrow S,\\ C{:=}C\text{—}\{<s,n,n',\mathrm{register}(s),1>\},\ S{:=}S\cup\{s\},\\ g \text{ — } \mathrm{notify}(n,\mathrm{accept}()) \rightarrow s,\\ g \text{ — } \mathrm{tell}(K) \rightarrow s,\\ \forall <z,m,m',p,x>\in C,\ \left\{\begin{array}{l} C{:=}C+\{<z,m,m',p,x/x+1>\}\\ g \text{ — } \mathrm{ask\text{-}all}(p,m') \rightarrow s \end{array}\right. \end{array}}\quad <s,n,n',\mathrm{register}(s),1>\in C$$

The interesting point is the submission of knowledge, so let see what happens.

4.2 Submission

When a group repository receives a new proposal to integrate in the group repository, it issues a call for comments, identified by a unique number (surrogate), towards all the subscribers (cfc-achieve) which is handled by the usual rules (reject-reply, accept-reply, challenge-reply).

(cfc-achieve)

$$\frac{s \text{ — } achieve(m,p) \rightarrow g}{\begin{array}{c} C{:=}C\cup\{ <s,m,!n,achieve(p),|S|>\}, \\ g \text{ — } ask\text{-}all(achieve(p),\ n) \rightarrow S \end{array}} \quad K?p$$

However, the group repository only accepts the submissions which are consistent with its current content. Otherwise, it replies by an error.

(error-achieve)

$$\frac{s \text{ — } achieve(n,p) \rightarrow g}{g \text{ — } notify(n,error()) \rightarrow s} \quad \neg K?p$$

Once the proposal has been accepted by all the subscribers, it is inserted into the group repository and sent to all the subscribers and readers through the `tell` performative (accept-achieve).

(accept-achieve)

$$\frac{s \text{ — } reply(n,accept) \rightarrow g}{\begin{array}{c} C{:=}C\text{—}\{<s',n',n,achieve(p),1>\},\ K{:=}K+p, \\ g \text{ — } pnotify(n,accept()) \rightarrow S, \\ g \text{ — } notify(n',accept()) \rightarrow s',\ g \text{ — } tell(p) \rightarrow S \end{array}} \quad <s',n',n,achieve(p),1>\in C$$

4.3 Calls

The call for comments on a particular proposal is dealt with in the same way through the same set of rules. Here is the heart of the consensual protocol. The acceptances are recorded as long as they do not concern all the subscribers, the rejection answers terminate the consultation and the counter-proposals modify the initial proposal. This corresponds to phase 2(emission), 3(reception) and 5(emission) of the downward and upward policies. In each case, the initial proposal is issued through the `ask-all` performative:

The answers provided by the subscribers can be found in software engineering [Narayanaswamy& 92] and other fields. Among them, three cases may happen:

- They all agree that the modification must be accepted, then the modification is committed in the group repository and broadcast to all the subscriber repositories.
- One of them rejects the proposal, then the changes are not committed and the comments provided by the rejecter are sent to the submitter (the call for comments is discarded in all the subscriber repositories).

- One submitter sends a counter-proposal, then the call for comments is replaced by a call for comments about all the proposals available (those who already accepted the change, are asked to consider the new proposal and to answer again).

It also can happen that the submitter retracts the proposal thus leading to the retraction of the call for comments from all the repositories.

Upon a rejection message (issued by reject or accept-forward) the proposal is closed, it is cancelled for all the subscribers which did not answer yet and the answer to the initial proposal (including the justification) is sent to the sender (reject-reply).

$$\text{(reject-reply)}\qquad \frac{s \text{ — } \mathrm{reply}(n,\mathrm{reject}(r)) \rightarrow g}{\begin{array}{c} C{:=}C\text{—}\{<s',n',n,p,x>\}, \\ g \text{ — } \mathrm{pnotify}(n,\mathrm{reject}(r)) \rightarrow S, \\ g \text{ — } \mathrm{notify}(n',\mathrm{reject}(r)) \rightarrow s' \end{array}} \quad <s',n',n,p,x>\in C$$

An acceptance answer which is not that of the last subscriber to answer is simply recorded in the current call for comments directory (accept-reply).

$$\text{(accept-reply)}\qquad \frac{s \text{ — } \mathrm{reply}(n,\mathrm{accept}()) \rightarrow g}{C{:=}C+\{<s',n',n,p,x/x\text{-}1>\}} \quad <s',n',n,p,x>\in C,\ x>1$$

When a call for proposal is under processing and the initial issuer wants to cancel the proposal, the group repository removes the proposal from the current cfc directory (C) and sends a cancelling message towards all the subscribers (deny-reply). This corresponds to the phase 1(reception) and 5(emission) of the downward policy for the deny protocol.

$$\text{(deny-reply)}\qquad \frac{s \text{ — } \mathrm{deny}(n) \rightarrow g}{\begin{array}{c} C{:=}C\text{—}\{<s,n,n',\_,\_>\}, \\ g \text{ — } \mathrm{pnotify}(n',\mathrm{deny}()) \rightarrow S \end{array}} \quad <s,n,n',\_,\_>\in C$$

4.4 Forward submission

The forward submission deals with the same issue as the initiatives for individual repositories: submitting something to the group repositories. However, group repositories cannot take initiative for themselves, the initiative is taken by an individual repository, it is submitted to vote as usual, and, when accepted it is submitted to the group repository of the current one on behalf of all the subscribers.

As for knowledge submission, when a group repository receives a new proposal to submit to its own group repository, it issues a call for comments towards the subscribers (cfc-forward).

$$\text{(cfc-forward)}\qquad \frac{s \text{ — } \mathrm{forward}(n,p) \rightarrow g}{\begin{array}{c} C{:=}C\cup\{<s,n,!n,\mathrm{forward}(p),|S|>\}, \\ g \text{ — } \text{ask-all}(n',\mathrm{forward}(p)) \rightarrow S \end{array}}, \quad \begin{array}{l} G\neq\emptyset,\ p\neq \mathrm{subscribe}(g,B) \text{ or} \\ G=\emptyset,\ p=\mathrm{subscribe}(g,B) \end{array}$$

However, if the group repository does not have any group repository on its own, it issues an error message to the submitter (error-forward). This is only true if the proposal is not to subscribe to another repository which is dealt with below.

$$\text{(error-forward)}\ \frac{s \text{ — } forward(n,p) \to g}{g \text{ — } notify(n,error) \to s}\quad \begin{array}{c} G=\emptyset,\ p\neq subscribe(g,B) \\ \text{or } G\neq\emptyset,\ p=subscribe(g,B) \end{array}$$

Once all the subscribers have accepted to forward some proposal, this is notified to the subscribers and sent to the group repository (accept-forward).

$$\text{(accept-forward)}\ \frac{s \text{ — } reply(n,accept) \to g}{\begin{array}{c} C:=C\text{—}\{\langle s',n',n,forward(p),1\rangle\}, \\ g \text{ — } pnotify(n,accept) \to S, \\ A:=A\cup\{\langle !n'',p,n',s'\rangle\}, \\ g \text{ — } p(n'') \to G \end{array}}\quad \langle s',n',n,forward(p),1\rangle\in C$$

When the notification of the final issue of a submission (or an error) is issued from the group repository, since it concerns a proposal emitted by the group repository, it must be notified to the initial submitter in order for it to clean up its A table.

(group-handle-notify)

$$\frac{G \text{ — } notify(m,r) \to g}{A:=A\text{—}\{\langle m,p,n,s\rangle\},\ g \text{ — } notify(n,r) \to s}\quad \langle m,p,n,s\rangle\in A$$

An exception to these last rules concerns the case when the proposal to achieve is subscription. In such a case G is replaced by B (notify-register), and, when G=∅, there must be automatic subscription (accept-forward-register).

(accept-forward-register)

$$\frac{s \text{ — } reply(n,accept) \to g}{\begin{array}{c} C:=C\text{—}\{\langle s',n',n, forward(register(g,B)),1\rangle\}, \\ g \text{ — } pnotify(n,accept) \to S \\ A:=A\cup\{\langle !n'',register(g,B),n',s'\rangle\}, \\ g \text{ — } register(n'',g,B) \to B \end{array}}\quad \begin{array}{c} \langle s',n',n,forward(register(g,B)),1\rangle\in C, \\ G=\emptyset \end{array}$$

(group-notify-register)

$$\frac{B \text{ — } notify(m,accept()) \to g}{\begin{array}{c} G:=B,\ A:=A\text{—}\{\langle m, register(g,B),n,s\rangle\} \\ g \text{ — } notify(n,accept()) \to s \end{array}}\quad \langle m,register(g,B),n,s\rangle\in A,\ G=\emptyset$$

4.5 Broadcasting

So far, the messages which triggered the rules for group repositories were coming from upward (the subscribers). However, group repositories can also be subscribers and thus they are subject to the same messages as the individual repositories. The following rules handle the messages of the “automatic part of individual repositories” section (but the notify ones). They correspond to the phase 1(reception), 2(emission) and 5(reception) of the upward policy. They are received by the group repositories and are automatically processed (i.e. registered and automatically forwarded to the subscribers).

4.5.1 Group call for comments

When a group repository receives a call for comments issued by its group repository, it relays this call to its subscribers and processes it as usual. However, it notes that the call does not concern the repository but a forwarding to its group repository. Thus, upon acceptance or rejection, the notification results in replying to the group repository.

(broadcast-cfc)
$$\frac{G \text{ — ask-all}(n,p) \rightarrow g}{\begin{array}{c} P{:=}P \cup \{<n,p,@>\}, \\ C{:=}C \cup \{<G,n,!m, \text{forward}(p),|S|>\}, \\ g \text{ — ask-all}(m,\text{forward}(p)) \rightarrow S \end{array}}$$

(reject-forward-cfc)
$$\frac{s \text{ — reply}(n,\text{reject}(r)) \rightarrow g}{\begin{array}{c} C{:=}C \text{—} \{<G,n',n,p,x>\}, \\ g \text{ — pnotify}(n,\text{reject}(r)) \rightarrow S \\ P{:=}P +<n',p,@/R>, \\ g \text{ — reply}(n',\text{reject}(r)) \rightarrow G \end{array}} \quad \begin{array}{c} <G,n',n,\text{forward}(p),x> \in C, \\ <n',p,@> \in P \end{array}$$

(accept-forward-cfc)
$$\frac{s \text{ — reply}(n,\text{accept}) \rightarrow g}{\begin{array}{c} C{:=}C \text{—} \{<G,n',n,\text{forward}(p),1>\}, \\ g \text{ — pnotify}(n,\text{accept}) \rightarrow S, \\ P{:=}P+\{<n',p,@/A>\}, \\ g \text{ — reply}(n',\text{accept}) \rightarrow G \end{array}} \quad \begin{array}{c} <G,n',n,\text{forward}(p),1> \in C, \\ <n',p,@> \in P \end{array}$$

4.5.2 Poll notification

Whenever a proposal issued by the group repository achieves a particular status, this is notified to the group repository. Two cases can arise: either the vote for the current group is closed, in such a case the group has just to clean up its P table (group-handle-pnotify) or it is not closed (so the proposal is either denied or rejected) in which case the call for comments must be cancelled and the tables cleaned up (broadcast-notify).

(group-handle-pnotify)
$$\frac{G \text{ — pnotify}(n,r) \rightarrow g}{P{:=}P \text{—} \{<n,\_,\_>\}} \quad \begin{array}{c} <G,n,\_,\_,\_> \notin C, <n,\_,\_> \in P, \\ r \neq \text{challenge}(\_,\_,\_,\_) \end{array}$$

$$\text{(broadcast-pnotify)}\quad \frac{G \text{ — } \mathrm{pnotify}(n',r) \rightarrow g}{\begin{array}{c} C{:=}C\text{—}\{\langle G,n',n,\mathrm{forward}(p),\_\rangle\}, \\ P{:=}P\text{—}\{\langle n',\_,\_\rangle\}, \\ g \text{ — } \mathrm{pnotify}(n,r) \rightarrow S \end{array}} \quad \begin{array}{c} \langle G,n',n,\mathrm{forward}(p),\_\rangle \in C, \\ \langle n',\_,\_\rangle \in P \end{array}$$

4.5.3 Broadcasting repository content

When the group repository receives, from its own group repository, knowledge that it accepted to integrate in this last repository, it asks the right to do so to the subscribers. This is for dealing with the protocol whose idea is the following: When g receives from G a tell, it asks to everybody if they agree to include it. If this is agreed, then this is added and then re-told. If it is not, then the proposal is logged and that is all.

$$\text{(log-tell)}\quad \frac{G \text{ — } \mathrm{tell}(\_,p) \rightarrow g}{L{:=}L\cup\{p\}} \quad \neg K?p$$

$$\text{(cfc-tell)}\quad \frac{G \text{ — } \mathrm{tell}(n',p) \rightarrow g}{\begin{array}{c} C{:=}C\cup\{\langle G,n',!n,\mathrm{tell}(p),|S|\rangle\}, \\ g \text{ — } \mathrm{ask\text{-}all}(n,\mathrm{achieve}(p)) \rightarrow S \end{array}} \quad K?p$$

Once all the subscribers have accepted a tell issued from the group repository, this tell is added to the group repository and forwarded to the subscribers.

$$\text{(accept-tell)}\quad \frac{s \text{ — } \mathrm{reply}(n,\mathrm{accept}()) \rightarrow g}{\begin{array}{c} C{:=}C\text{—}\{\langle G,n,\mathrm{tell}(p),1\rangle\}, \\ g\text{— } \mathrm{pnotify}(n,\mathrm{accept}()) \rightarrow S, \\ K{:=}K{+}p,\ g \text{ — } \mathrm{tell}(p) \rightarrow S \end{array}} \quad \langle G,n,\mathrm{tell}(p),1\rangle \in C$$

$$\text{(reject-tell)}\quad \frac{s \text{ — } \mathrm{reply}(n,\mathrm{reject}(r)) \rightarrow g}{\begin{array}{c} C{:=}C\text{—}\{\langle G,n,\mathrm{tell}(p),x\rangle\}, \\ g\text{— } \mathrm{pnotify}(n,\mathrm{reject}(r)) \rightarrow S, \\ L{:=}L\cup\{p\} \end{array}} \quad \langle G,n,\mathrm{tell}(p),x\rangle \in C$$

4.6 Challenge management

Challenge management is very long to describe (and can be skipped on first reading) because the rule for challenge management does several things at once. It cares about who sent the initial proposal, issues call for comments, deals with the old call for comments… So the combination of all the possible exceptions to these tasks must be taken into account.

4.6.1 Basic challenge

However, the idea is very simple and makes use of what is already working for the regular votes. The challenges are made along a separation of the proposal into two complementary pieces: an accepted part (k+) and a rejected part (k-). To these two pieces is added a third part (k*) which aims at replacing the rejected part.

The consequences of the challenge are that the initial submission is reduced to its least still acceptable part (k+), that k- is rejected and a new call for comments is issued for k*.

(challenge-reply)

$$\frac{s \text{ — } reply(n,challenge(n'',k-,k+,k*)) \rightarrow g}{\begin{array}{c} C:=C+\{<s',n',n,forward^{i}(achieve(k/k\cap k+)),x/x-1>\}, \\ g \text{ —}pnotify(n,challenge(n'',k-,k+,k*)) \rightarrow S, \\ g\text{— } notify(n',challenge(n'',k-,k+,k*)) \rightarrow s', \\ C:=C\cup\{<s,n'',!m,forward^{i}(achieve(k*)),|S|>\}, \\ g\text{— } ask\text{-}all(forward^{i}(achieve(k*)),m) \rightarrow S \end{array}} \quad \begin{array}{c} <s',n',n,forward^{i}(achieve(k)),x>\in C, \\ x\neq 1,\ k\cap k+\neq \emptyset \end{array}$$

The two following rules consider the case when the challenger is the last subscriber to answer. In such a case, the k+ part is accepted. Thus these rules are a mix of the former one and (accept-achieve) and (accept-forward) respectively.

(challenge-accept)

$$\frac{s \text{ — } reply(n,challenge(n'',k-,k+,k*)) \rightarrow g}{\begin{array}{c} C:=C\text{—}\{<s',n',n,\ achieve(k),1>\}, \\ K:=K+(k\cap k+), \\ g \text{ —}pnotify(n,challenge(n'',k-,k+,k*)) \rightarrow S>\}, \\ g \text{ —}pnotify(n,accept) \rightarrow S, \\ g\text{— } notify(n',challenge(n'',k-,k+,k*)) \rightarrow s', \\ g\text{— } notify(n',accept()) \rightarrow s', \\ g \text{ —}tell(k\cap k+) \rightarrow S, \\ C:=C\cup\{<s,n'',!m,\ achieve(k*),|S|>\}, \\ g\text{— } ask\text{-}all(achieve(k*),m) \rightarrow S \end{array}} \quad \begin{array}{c} <s',n',n,achieve(k),1>\in C, \\ k\cap k+\neq \emptyset \end{array}$$

(challenge-accept-forward)

$$\frac{s \text{ — } reply(n,challenge(n'',k-,k+,k*)) \rightarrow g}{\begin{array}{c} C:=C\text{—}\{<s',n',n,forward^{i}(achieve(k)),1>\}, \\ g \text{ —}pnotify(n,challenge(n'',k-,k+,k*)) \rightarrow S, \\ g \text{ —}pnotify(n,accept) \rightarrow S, \\ g\text{— } notify(n',challenge(n'',k-,k+,k*)) \rightarrow s', \\ A:=A\cup\{<!m',\ forward^{i-1}(achieve(k\cap k+)),n',s'>\}, \\ g \text{ — } forward^{i-1}(m',achieve(k\cap k+)) \rightarrow G, \\ C:=C\cup\{<s,n'',!m,forward^{i}(achieve(k*)),|S|>\}, \\ g\text{— } ask\text{-}all(forward^{i}(achieve(k*)),m) \rightarrow S \end{array}} \quad \begin{array}{c} <s',n',n,forward^{i}(achieve(k)),1>\in C, \\ k\cap k+\neq \emptyset,\ i>0 \end{array}$$

If a vote for k (the previous proposal) arrives while it has been challenged, the vote is taken into account just as if it was a vote for k+ (by the rules reject-reply, accept-reply or accept-*). However, if this vote is a new challenge, it is also considered by challenge-reply (this is the reason for the intersection between k and k+: if this intersection becomes empty, the process is dealt with in §4.6.3).

This is also a bit different if the proposal came from the group base of the group. In such, a case the notification goes to this group base.

(challenge-forward-cfc)

$$\frac{s \text{ — } \mathrm{reply}(n,\mathrm{challenge}(n'',k\text{-},k\text{+},k^*)) \rightarrow g}{\begin{array}{c} C{:=}C{+}\{<G,n',n,\mathrm{forward}^{i}(\mathrm{achieve}(k/k\cap k\text{+})),x/x\text{-}1>\}, \\ g \text{ —} \mathrm{pnotify}(n,\mathrm{challenge}(n'',k\text{-},k\text{+},k^*)) \rightarrow S, \\ g\text{— } \mathrm{notify}(n',\mathrm{challenge}(n'',k\text{-},k\text{+},k^*)) \rightarrow G, \\ C{:=}C\cup\{<s,n'',!m,\mathrm{forward}^{i}(\mathrm{achieve}(k^*)),|S|>\}, \\ g\text{— } \mathrm{ask\text{-}all}(\mathrm{forward}^{i}(\mathrm{achieve}(k^*)),m) \rightarrow S \end{array}} \quad \begin{array}{c} <G,n',n,\mathrm{forward}^{i}(\mathrm{achieve}(k)),x>\in C, \\ x\neq 1,\ k\cap k\text{+}\neq\emptyset,\ i>0 \end{array}$$

Just like above, if the challenger is the last subscriber to answer, the k+ part must be accepted. Thus the following rule consider this as a mix of the former rule and (accept-forward-cfc).

(challenge-accept-forward-cfc)

$$\frac{s \text{ — } \mathrm{reply}(n,\mathrm{challenge}(n'',k\text{-},k\text{+},k^*)) \rightarrow g}{\begin{array}{c} C{:=}C\text{—}\{<G,n',n,\mathrm{forward}^{i}(\mathrm{achieve}(k)),1>\}, \\ g \text{ —} \mathrm{pnotify}(n,\mathrm{challenge}(n'',k\text{-},k\text{+},k^*)) \rightarrow S, \\ g\text{— } \mathrm{notify}(n',\mathrm{challenge}(n'',k\text{-},k\text{+},k^*)) \rightarrow G, \\ g \text{ —} \mathrm{pnotify}(n,\mathrm{accept}) \rightarrow S, \\ P{:=}P{+}\{<n,\mathrm{forward}^{i}(\mathrm{achieve}(k/k\cap k\text{+})),@/A>\}, \\ g \text{ — } \mathrm{reply}(n',\mathrm{accept}) \rightarrow G, \\ C{:=}C\cup\{<s,n'',!m,\mathrm{forward}^{i}(\mathrm{achieve}(k^*)),|S|>\}, \\ g\text{— } \mathrm{ask\text{-}all}(\mathrm{forward}^{i}(\mathrm{achieve}(k^*)),m) \rightarrow S \end{array}} \quad \begin{array}{c} <G,n',n,\mathrm{forward}^{i}(\mathrm{achieve}(k)),1>\in C, \\ k\cap k\text{+}\neq\emptyset \end{array}$$

4.6.2 Notification and table update

In order to maintain the A and P tables, the p-notification must be done first. The initial issuers of proposals are notified of the challenge and their A tables are modified in order to accommodate it exactly as if they had issued a proposal for k+. These two rules replace the (handle-notify) and (group-handle-notify) for individual and group repositories respectively.

(handle-challenge)

$$\frac{g \text{ — } \mathrm{notify}(n,\mathrm{challenge}(\_,\_,k\text{+},\_)) \rightarrow s}{A{:=}A{+}\{<n,\mathrm{forward}^{i}(\mathrm{achieve}(k/k\cap k\text{+})),\_,\_>\}} \quad \begin{array}{c} <n,\mathrm{forward}^{i}(\mathrm{achieve}(k)),\_,\_>\in A, \\ k\cap k\text{+}\neq\emptyset \end{array}$$

(group-handle-challenge)

$$\frac{G \text{ — } \text{notify}(n,\text{challenge}(\_,\_,k+,\_)) \rightarrow g}{\begin{array}{c} A{:=}A+\{\langle n,\text{forward}^{i}(\text{achieve}(k/k\cap k+)),n',s\rangle\},\\ g \text{ — } \text{notify}(n',\text{challenge}(\_,\_,k+,\_)) \rightarrow s \end{array}} \quad \begin{array}{c} \langle n,\text{forward}^{i}(\text{achieve}(k)),n',s\rangle \in A,\\ k\cap k+\neq\emptyset \end{array}$$

Nonetheless the notification can come from a subscriber base (because a subscriber has challenged the initial subscription during the vote process — see (challenge-forward-cfc)). Then, the base will change all the calls for vote it issued and also notify the initial sender. The new call for vote (for k*) will come later from the other base. Note that the number of subscribers to answer (x) is not changed and that the rule does not depend on it because the initial base has reduced the extent of the proposal but has not accepted it yet.

(broadcast-challenge)

$$\frac{s \text{ — } \text{notify}(n,\text{challenge}(\_,\_,k+,\_)) \rightarrow g}{\begin{array}{c} C{:=}C+\{\langle s',n,n',\text{forward}^{*}(\text{achieve}(k/k\cap k+)),x\rangle\},\\ g \text{ —} \text{pnotify}(n,\text{challenge}(\_,k,k\cap k+,\_)) \rightarrow S,\\ g\text{— } \text{notify}(n',\text{challenge}(\_,k,k+,\_)) \rightarrow s' \end{array}} \quad \begin{array}{c} \langle s',n,n',\text{forward}^{*}(\text{achieve}(k)),x\rangle \in C,\\ k\cap k+\neq\emptyset \end{array}$$

The subscribers are also notified of the change in the voting process. This allow them to update their P table in order to account for the new situation. Again, it is just like if the initial proposal had been k+.

(handle-pnotify-challenge)

$$\frac{g \text{ — } \text{pnotify}(n,\text{challenge}(\_,\_,k+,\_)) \rightarrow s}{P{:=}P+\{\langle n,\text{forward}^{i}(\text{achieve}(k/k\cap k+)),\_\rangle\}} \quad \begin{array}{c} \langle n,\text{forward}^{i}(\text{achieve}(k)),\_\rangle \in P,\\ k\cap k+\neq\emptyset \end{array}$$

(group-handle-pnotify-challenge)

$$\frac{G \text{ — } \text{pnotify}(n',\text{challenge}(\_,\_,k+,\_)) \rightarrow g}{P{:=}P+\{\langle n',\text{forward}^{i}(\text{achieve}(k/k\cap k+)),\_\rangle\}} \quad \begin{array}{c} \langle G,n',\_,\_,\_\rangle \notin C,\ k\cap k+\neq\emptyset,\\ \langle n',\text{forward}^{i}(\text{achieve}(k)),\_\rangle \in P \end{array}$$

(broadcast-pnotify-challenge)

$$\frac{G \text{ — } \text{pnotify}(n',\text{challenge}(\_,\_,k+,\_)) \rightarrow g}{\begin{array}{c} C{:=}C+\{\langle G,n',n,\text{forward}^{i+1}(\text{achieve}(k/k\cap k+)),\_\rangle\},\\ P{:=}P+\{\langle n',\text{forward}^{i}(\text{achieve}(k/k\cap k+)),\_\rangle\},\\ g \text{ — } \text{pnotify}(n,\text{challenge}(\_,k,k\cap k+,\_)) \rightarrow S \end{array}} \quad \begin{array}{c} \langle G,n',n,\text{forward}^{i+1}(\text{achieve}(k)),\_\rangle \in C,\\ \langle n',\text{forward}^{i}(\text{achieve}(k)),\_\rangle \in P,\\ k\cap k+\neq\emptyset \end{array}$$

4.6.3 When the initial proposal is rejected

When k∩k+=Ø in some of the rules above (note that this can happen at any moment), the initial proposal is totally rejected. However, in order to interact correctly with the above rules and to acknowledge the fact that a challenge has occurred, special rules are used. They just do what is done in the basic challenge rules but act just like rejection with regard to K.

(challenge-reject)

$$\frac{s \text{ — } reply(n',challenge(n'',k-,k+,k^*)) \rightarrow g}{\begin{array}{c} C:=C\text{—}\{<s',n,n',forward^i(achieve(k)),x>\}, \\ g \text{ —}pnotify(n',reject(challenge(n'',k-,k+,k^*))) \rightarrow S, \\ g\text{— } notify(n,reject(challenge(n'',k-,k+,k^*))) \rightarrow s', \\ C:=C\cup\{<s,n'',!m,forward^i(achieve(k^*)),|S|>\}, \\ g\text{— } ask\text{-}all(forward^i(achieve(k^*)),m) \rightarrow S \end{array}} \quad \begin{array}{c} <s',n,n',forward^i(achieve(k)),x>\in C, \\ k\cap k+=\emptyset \end{array}$$

(challenge-reject-forward-cfc)

$$\frac{s \text{ — } reply(n',challenge(n'',k-,k+,k^*)) \rightarrow g}{\begin{array}{c} C:=C\text{—}\{<G,n,n',forward^i(achieve(k)),x>\}, \\ g \text{ —}pnotify(n',reject(challenge(n'',k-,k+,k^*))) \rightarrow S, \\ g\text{— } notify(n,reject(challenge(n'',k-,k+,k^*))) \rightarrow G, \\ C:=C\cup\{<s,n'',!m,forward^i(achieve(k^*)),|S|>\}, \\ g\text{— } ask\text{-}all(forward^i(achieve(k^*)),m) \rightarrow S \end{array}} \quad \begin{array}{c} <G,n,n',forward^i(achieve(k)),x>\in C, \\ k\cap k+=\emptyset \end{array}$$

The handling of notification in case of null challenge are not presented here. As a matter of fact it consists in using (handle-notify), (group-handle-notify), (handle-pnotify), (group-handle-pnotify) and (broadcast-pnotify) instead of (handle-challenge), (group-handle-challenge), (broadcast-challenge), (handle-pnotify-challenge), (group-handle-pnotify-challenge) and (broadcast-pnotify-challenge) respectively. So these regular rules have to be slightly modified (by adding a guard to them testing if the content is a challenge and $k\cap k+=\emptyset$) since they compete with the challenge version. (broadcast-challenge) is replaced by (reject-reply).

These rules for challenging seems to be cumbersome since they almost duplicate the whole protocol. Moreover, it would be more concise to express the protocol with challenge as the central answer and `accept` (resp. `reject`) as a shortcut for `challenge(n,Ø,k,Ø)` (resp. `challenge(n,k,Ø,Ø)`). This has not been considered here in order to achieve a progressive and modular presentation of the protocol.

4.7 Miscellaneous

These rules are not vital for the protocol but are included here for the sake of completeness.

4.7.1 Evaluation

When a repository asks for an evaluation, the group repository replies immediately by the result of the evaluation of the proposal against its own repository (eval-reply).

(eval-notify)

$$\frac{s \text{ — } evaluate(n,p) \rightarrow g}{g \text{ — } notify(n,K?p) \rightarrow s}$$

4.7.2 Late messages

These messages correspond to the reception of a message concerning a submission which has already received a status (a vote for a proposal that has been rejected, the denial of a proposal which has achieved a status or the notification of status for a proposal that has been denied). They were left unspecified in the initial description of the protocol while it was clear that they could happen. The LOTOS simulation of the protocol identified them so they are presented here:

(late-reply) $$\frac{\mathrm{s} \text{ — } \mathrm{reply}(\mathrm{n},\_) \rightarrow \mathrm{g}}{} \quad <\mathrm{n},\_,\_,\_,\_> \notin \mathrm{C}$$

(late-deny) $$\frac{\mathrm{s} \text{ — } \mathrm{deny}(\mathrm{n}) \rightarrow \mathrm{g}}{} \quad <\mathrm{n},\_,\_,\_,\_> \notin \mathrm{C}$$

(late-notify) $$\frac{\mathrm{s} \text{ — } \mathrm{notify}(\mathrm{n},\_) \rightarrow \mathrm{g}}{} \quad <\mathrm{n},\_,\_,\_> \notin \mathrm{A}$$

5. Example and trace

In this simple example, the repository is an initially empty set of couples made of a letter (character) and a number (which can also be thought of as a letter indexed array of numbers). A piece of knowledge is such a couple. When confronted to the repository content it is declared as redundant if the piece of knowledge is syntactically contained in the repository; it is declared contradictory if the repository contains a couple with the same letter and a different number; and it is declared admissible if no couple with the same letter is in the repository content. This could be further complicated by allowing to replace a number by a greater one but the current setting is enough for demonstration purposes. Repositories will thus submit letter-number couples to other repositories (so they will be noted Letter=number).

Below is a trace of the protocol automatically provided by its description in LOTOS (a language especially designed for describing protocols [Bolognesi& 1987]). The LOTOS simulator having a restricted representation language [Pecheur 1997], the whole trace is the concatenation of three smaller tests whose output has been edited (with regular expressions) and commented for legibility purpose.

The convention used by the trace consists in noting on the right the name of the rule which sends the message (so, of course, the rule is triggered by the sender). The description is abstracted from the contents of the memory (only messages are traced). Rule names in parenthesis are processed automatically while these in brackets are generated by a user. The initial messages are those whose indentation is at the margin. The surrogates of messages have the form reply-with#/in-reply-to# in front of the message. They have been restricted to the strict minimum. The actions prefixed by a dollar sign ($) are external to the system.

The first track presents the creation of three repositories, one of which is a group repository. The two other repositories want to register in the group repository and succeed after voting. Then, one of them asks the group repository to compare some contents with its contents.

```
$ create b1
$ create b2
$ create b3
$ turn-to-group b3

/* simple subscription */
b1 - (1/)register -> b3                                         [register]
  b3 - (/1)notify(accept)) -> b1                          (reply-register)

/* subscription plus votes */
b2 - (1/)register -> b3                                         [register]
  b3 - (1/)ask-all(register(b2)) -> b1                      (cfc-register)
    b1 - (/1)reply(accept) -> b3                       (store-cfc)[accept]
  b3 - (/1)pnotify(accept) -> b1                         (accept-register)
  b3 - (/1)notify(accept) -> b2                                          —

/* simple confrontation */
b1 - (2/)evaluate({A=1}) -> b3                                  [evaluate]
  b3 - (/2)notify({}) -> b1                                  (eval-notify)
```

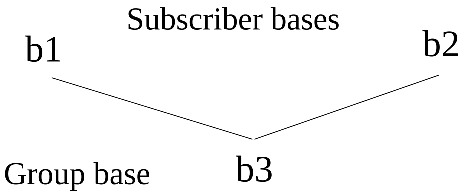

Figure 6. Here is a presentation of the connections at that point.

The repository b1 submits some contents (A=1) to the group repository which automatically issues a call for comments. The repository b2 offers a counter-proposal (A=3) which is integrated by the group repository to the call for comments. The repository b1 then cancels its first proposal and accepts the counter proposal which is accepted and broadcast to the subscribers.

```
/* submission with counter-proposal */
b1 - (3/)achieve({A=1,D=2}) -> b3                                [achieve]
  b3 - (2/)ask-all(achieve({A=1,D=2})) -> b2                 (cfc-achieve)
  b3 - (2/)ask-all(achieve({A=1,D=2})) -> b1                             —
    b1 - (/2)reply(accept) -> b3                       (store-cfc)[accept]
    b2 - (2/2)reply(challenge({A=1},{A=3},{D=2})) -> b3
                                                    (store-cfc)[challenge]
      b3 - (/2)pnotify(challenge({A=1},{A=3},{D=2})) -> b1
                                                        (challenge-accept)
      b3 - (/2)pnotify(challenge({A=1},{A=3},{D=2})) -> b2               —
      b3 - (/2)pnotify(accept) -> b1                                     —
      b3 - (/2)pnotify(accept) -> b2                                     —
      b3 - (/3)notify(challenge({A=1},{A=3},{D=2})) -> b1                —
      b3 - (/2)tell({D=2}) -> b1                                         —
      b3 - (/2)tell({D=2}) -> b2                                         —
      b3 - (3/)ask-all(achieve({A=3})) -> b2                             —
      b3 - (3/)ask-all(achieve({A=3})) -> b1                             —
        b1 - (/3)reply(accept({A=3}) -> b3             (store-cfc)[accept]
        b2 - (/3)reply(accept({A=3}) -> b3             (store-cfc)[accept]
      b3 - (/3)pnotify(accept) -> b1                      (accept-achieve)
      b3 - (/3)pnotify(accept) -> b2                                     —
      b3 - (/2)notify(accept) -> b2                                      —
      b3 - tell({A=3}) -> b1                                             —
      b3 - tell({A=3}) -> b2                                             —
```

The repository b1 is turned into a group repository and two new repositories (b4 and b5) are created. They subscribe successfully to b1 and the contents of b1 is broadcast to them.

```
/* transformation of an individual repository into a group repository */
$ turn-to-group b1 (with A=3 in its K)
$ create b4
$ create b5

/* simple subscription */
b4 - (1/)register -> b1                                               [register]
  b1 - (/1)notify(accept) -> b4                                 (reply-register)
  b1 - tell({A=3,D=2}) -> b4                                                   —

/* subscription plus votes */
b5 - (1/)register -> b1                                               [register]
  b1 - (4/)ask-all(register(b5)) -> b4                            (cfc-register)
    b4 - (/4)reply(accept) -> b1                             (store-cfc)[accept]
  b1 - (/4)pnotify(accept) -> b4                               (accept-register)
  b1 - (/1)notify(accept) -> b5                                                —
  b1 - tell({A=3,D=2}) -> b5                                                   —
```

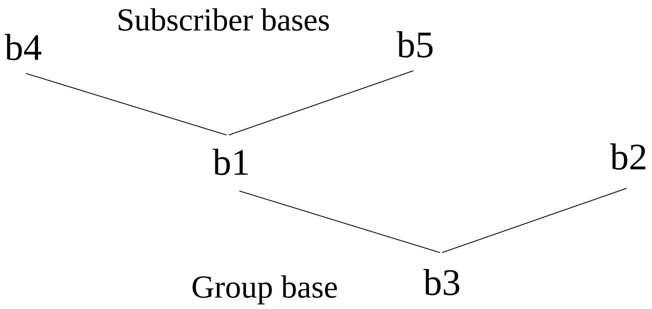

Figure 7. Here is a presentation of the connections at that point.

The repository b5 first compares some contents (B=2) with the contents of the repository b1. Then, it finds that another contents (B=4) is better and submits it to b1. This is accepted by b4 and included in b1. Then, the repository b5 asks b1 to submit B=4 to b3. After a call for comments accepted by the repository b4, this is submitted to the repository b3. As usual b3 issues a call for comments and the proposal is rejected by b2.

```
/* simple submission */
b5 - (2/)evaluate({B=2}) -> b1                                        [evaluate]
  b1 - (/2)reply({B=4}) -> b5                                      (eval-notify)

b5 - (3/)achieve({B=4}) -> b1                                          [achieve]
  b1 - (5/)ask-all(achieve({B=4})) -> b5                           (cfc-achieve)
  b1 - (5/)ask-all(achieve({B=4})) -> b4                                       —
    b5 - (/5)reply(accept) -> b1                             (store-cfc)[accept]
    b4 - (/5)reply(accept) -> b1                             (store-cfc)[accept]
  b1 - (/5)pnotify(accept) -> b4                                (accept-achieve)
  b1 - (/5)pnotify(accept) -> b5                                               —
  b1 - (/3)notify(accept) -> b5                                                —
  b1 - tell({B=4}) -> b4                                                       —
  b1 - tell({B=4}) -> b5                                                       —

/* submission to nested group repository, it fails */
b5 - (4/)forward(achieve({B=4})) -> b1                                 [forward]
  b1 - (6/)ask-all(forward(achieve({B=4}))) -> b5                  (cfc-forward)
  b1 - (6/)ask-all(forward(achieve({B=4}))) -> b4                         –
    b5 - (/6)reply(accept) -> b1                        (store-cfc)[accept]
    b4 - (/6)reply(accept) -> b1                        (store-cfc)[accept]
  b1 - (/6)pnotify(accept) -> b4                           (accept-forward)
  b1 - (/6)pnotify(accept) -> b5                                          –
  b1 - (7/)achieve({B=4}) -> b3                                           –
    b3 - (4/)ask-all(achieve({B=4})) -> b1                    (cfc-achieve)
    b3 - (4/)ask-all(achieve({B=4})) -> b2                                –
      b1 - (8/)ask-all(forward(achieve({B=4})) -> b4        (broadcast-cfc)
      b1 - (8/)ask-all(forward(achieve({B=4})) -> b5                      –
      b2 - (/4)reply(reject(r)) -> b3                   (store-cfc)[reject]
    b3 – (/4)pnotify(reject(r)) -> b1                        (reject-reply)
    b3 – (/4)pnotify(reject(r)) -> b2                                     –
    b3 – (/7)notify(reject(r)) -> b1                                      –
      b1 – (/4)notify(reject(r)) -> b5                (group-handle-notify)
      b1 – (/8)pnotify(reject(r)) -> b4                 (broadcast-pnotify)
      b1 – (/8)pnotify(reject(r)) -> b5                                   –
```

The repository b2 then submits a new proposal (B=3) to the repository b3 which issues a call for comments which is broadcast by the repository b1 to the repositories b4 and b5. They accept the proposal which is incorporated in b3. The repository b3 then broadcasts this new contents to the repository b1 which issues a new call for comments for incorporating B=3 in its own contents. This is accepted by the subscribers (b4 and b5) and then incorporated into the contents of the repository b1 and broadcast to the subscribers.

```
/* another submission to nested group repository, which succeed */
b2 - (3/)achieve({B=3}) -> b3                                     [achieve]
  b3 - (5/)ask-all(achieve({B=3})) -> b2                      (cfc-achieve)
  b3 - (5/)ask-all(achieve({B=3})) -> b1                                  –
    b2 - (/5)reply(accept) -> b3                        (store-cfc)[accept]
    b1 - (9/)ask-all(forward(achieve({B=3}))) -> b4         (broadcast-cfc)
    b1 - (9/)ask-all(forward(achieve({B=3}))) -> b5                       –
      b5 - (/9)reply(accept) -> b1                      (store-cfc)[accept]
      b4 - (/9)reply(accept) -> b1                     (store-cfc) [accept]
    b1 - (/9)pnotify(accept) -> b4                     (accept-forward-cfc)
    b1 - (/9)pnotify(accept) -> b5                                        –
    b1 - (/5)reply(accept) -> b3                                          –
  b3 - (/5)pnotify(accept) -> b1                           (accept-achieve)
  b3 - (/5)pnotify(accept) -> b2                                          –
  b3 - (/3)notify(accept) -> b2                                           –
  b3 - tell({B=3}) -> b2                                                  –
  b3 - tell({B=3}) -> b1                                                  –
    b1 - (10/)ask-all(achieve({B=3})) -> b4                      (cfc-tell)
    b1 - (10/)ask-all(achieve({B=3})) -> b5                               –
      b4 - (/10)reply(accept) -> b1                    (store-cfc) [accept]
      b5 - (/10)reply(accept) -> b1                    (store-cfc) [accept]
    b1 - (/10)pnotify(accept) -> b4                           (accept-tell)
    b1 - (/10)pnotify(accept) -> b5                                       –
    b1 - tell({B=3}) -> b4                                                –
    b1 - tell({B=3}) -> b5                                                –
```

Note that at the beginning, the user initiative seems to predominate while as the architectural complexity and the repositories grows, the automatic part becomes more important.

6. Properties

At first sight, the relevant properties for such a protocol are those of:

(0) having a protocol completely specified;
(1) not having a repository in an inconsistent state;
(2) having the opportunity to submit a proposal at any moment;
(3) having the acceptance of a proposal if and only if each subscriber — at the moment of the decision — agrees (and its reverse, not having a proposal if some subscriber disagrees);
(4) having an answer (accepted/refused/retracted) to a proposal in a finite amount of time;
(5) being minimal in the number of transactions.

Termination is not very useful in the case of the CO_4 protocol since the system could perfectly be used without end. Rather, (4) states a local termination property. The simplest property that can be put forth (0) is the fact that to any message issued corresponds a particular answer (this is important because, since non recognised messages are ignored it is easy to forget some rule):

PROPOSITION 0 (intelligibility). For each message sent with a particular performative and a particular type of receiver (group or individual repository), there is a rule triggered by that kind of message.

The proof can be carried out by simple enumeration. It has been achieved with the help of a Lisp simulator abstracting from the memory concerns (but taking conditions into account). This work has also been carried out with the (partial) LOTOS specification of the protocol.

The other properties are discussed below under additional assumptions. They are assumptions because the system has no means to enforce them.

ASSUMPTIONS. CO_4 is considered here under the following assumptions:

(a) there is always, in a finite amount of time, an answer to a query to an individual repository,
(b) there is no infinite number of counter-proposals for a proposal (which is entailed if the KRL language expressions (k) are non infinitely decomposable),
(c) there is not an infinite amount of subscriptions in a finite amount of time, and,
(d) different proposals examined concurrently are independent.

Each of the concerned assumptions are considered with the properties to which they are relevant. Obviously, the assumptions (b) and (c) are very reasonable ones and are not discussed. Assumption (a) is difficult to achieve but absolutely necessary for proposition 4; this is not a property of our voting scheme but is common to any scheme dedicated to solve the “transaction commit problem” in distributed database systems (even for majority votes) [Fischer& 1985]. Time-out are usual ways to overcome its failure. Assumption (d) is difficult to achieve too but necessary for achieving proposition 1 in the current state of the protocol. It will have to be relaxed but can already be enforced by detecting the dependencies and buffering queries as long as a decision has not been taken concerning the dependant proposals (thus, weakening proposition 2).

Let aside the problems of message order and parallelism, the only cause of non determinism is the intervention of individual users. It only appears in case of (1) subscription, (2) submission and (3) review.

Submission is the initial performative. It must be noted that a submission does not interfere with the examination of other submissions. Thus, the termination of the review process of a submission can be considered independently of other submissions.

PROPOSITION 1 (Repository consistency). Under assumption (d), the repositories associated with the group repositories are never in an inconsistent state.

proof. the consistency is a prerequisite to the addition of some knowledge (cfc-achieve being a prerequisite to accept-achieve) and concurrent proposals are independent (assumption d), the repositories are hence consistent. ◊

PROPOSITION 2 (Liveness and fairness). The subscribers can submit proposals to their group repository at any moment.

proof. there is no prerequisite for submission (achieve, forward), thus any subscriber can submit anything at any moment. Additionally, a repository which wants to subscribe to another one can do it at any moment as far as it has not subscribed to a base (G=Ø) and it has not initiated another subscription (A=Ø). ◊

Obviously, the subscription interferes with the review process. The following lemma allows the proofs below to consider the new subscriber exactly like if it were already subscriber at the beginning of the voting process.

LEMMA (Registration integration). The vote of the new subscribers (registrant between the initial submission of a proposal and its final vote) are taken into account exactly as if they were subscribers before the submission.

proof.

Let us see what happens when a new subscriber is accepted. With regard to the current proposals, three cases can happen:

1) The new subscriber is added before the submission (then it is in S and taken into account for the submission — it is thus taken into account in the proofs below);
2) The new subscriber is added during the call. Then this appears between two stages of the call. Through rule (accept-register) s is added to S and all the counters in the call for comments under examination are incremented. Thus, the opinion of the new subscriber is taken into account by the group repository exactly if it were there at the beginning. This is also true when challenges occur: the rejected part would have been rejected even with the opinion of the new subscriber and the accepted and new parts are still under examination so the new subscriber is taken into account.
3) The new subscriber is added after the call: the proposal has been either accepted, rejected, retracted or challenged. The subscriber is supposed to accept the result of the vote just as this was the case when it issued the subscription demand. ◊

The following propositions consider that the submission is an atomic submission (i.e. a submission which is not fragmented by challenges). This is equivalent to consider that all submissions of composite proposals are a shorthand for several atomic submissions (in such a case, no challenge is possible anymore).

PROPOSITION 3 (Consensus). Any submission is accepted if and only if all the subscribers (at the moment of its introduction in the repository) accept it.

proof.

if) all the subscribers (even if they are restricted to one) receive a call for comments (either from any rule whose name begins with “cfc-”, broadcast-cfc or accept-register in case of new subscribers); if all subscribers agree on the proposal, they all send a reply agreement. The first |S|-1 reply are recorded via the rule accept-reply. The |S|th reply fires one of the accept-* rules which accepts the proposal.

only if) the accept-* (and challenge-accept) rules are the only ones for the acceptance of a proposal. They all require the agreement of all the current subscribers (i.e. that the group repository has received as many accept reply as they are subscribers in S — and it is not possible to vote twice due to the conditions on P in the reply rules). ◊

PROPOSITION 4 (Termination). Under the assumptions (a-d) any submission reaches a status of either accepted or rejected proposal in a finite amount of time.

proof.

First, at the scale of the whole system, this can be proved by induction:

Base case: this is obviously the case for any individual subscriber (assumption a and b);

Inductive case: If a status of accepted or refused is achieved by every subscriber, it will be proved that such a status is also achieved by the group repository;

Thus, since the organisation of CO4 is hierarchical and the leaves are the individual subscribers, this induction property is sufficient for proving the property for each group repository. So the structure of our proofs is (1) the proof of the property at a group repository level without consideration of the nature of subscribers, (2) the proof at a group repository level which receives from downward the same solicitation in the same conditions and finally (3) the proof at the basic level of individuals.

At the initial stage, the set A of acceptors is Ø while the set B of subscribers is S. At each further stage (a new message is received), either the message is:

s — accept(n) → g and B≠{s} then A=A∪{s}, B=B–{s}, this case can only be dealt by the (accept-reply) rule which decrements the voter counter (which then corresponds to |S|-|A|);

s — accept(n) → g and B={s} then A=S, this case can only be dealt by the (accept-*) rules which consider the proposal as accepted: as a matter of fact all the subscribers have accepted;

s — reject(n) → g, this case can only be dealt by the (reject-reply) rule which considers the proposal as rejected: as a matter of fact, one of the subscribers has rejected it;

s — challenge(n,k^-,k^+,k^*) → g then, formally the whole proposal is rejected. However, this judgement can be refined. It can be considered that a part of the proposal (k^-) is rejected and, in fact, the challenge has exactly the effect of s — reject(n) → g for it. It can also be considered that the preserved part (k^+) is accepted and, again, the challenge has exactly the effect of s — accept(n) → g for it (and k^+ replaces the initial k). At last, the new submitted part (k^*) can be considered as a new submission. Hence, the challenge can be taken either as rejecting the proposal and it has then achieved a status or as accepting a subpart of it and rejecting another one which are dealt with according to the line above, so if they achieve a status, it is achieved in the present case too.

s — deny(n') → g, this case can only be dealt by the (deny-reply) rule which considers the proposal as denied;

Thus, at each stage, either p is rejected, accepted, re-examined with challengers or in a state such that A is the set of acceptors and B the state of those who have to answer.

From hypothesis (a), if there is no challenge, all the subscribers in S will have answered after a finite amount of time. Thus, the proposal is either accepted, rejected or included in a challenge. Since there is a finite number of challenges for a proposal, after a finite amount of time they will all be examined. They thus will all have a final status of accepted or rejected.

This was with the hypothesis that all the subscribers of a repository behave correctly. This is true of individual subscribers since they reply by accept, reject or challenge in a finite amount of time (assumption a) and, if they reply by challenge, the number of successive challenges is bounded by a finite number *n* (assumption b), after *n*+1 reply by subscribers the answer is either accept or reject. This is achieved in a finite time. This also holds true for group repositories, since the organisation of repositories is hierarchical and the proof above shows that the repository behaves correctly if its subscribers do (note that a subscriber can only answer once to a call for comments due to the condition on P in the [reply] rules).

So, this proposition holds for the achievement of submission of knowledge, subscription and forward submission. Moreover, the lemma above shows that the new subscription are correctly integrated in the voting process (just like if the subscribers were there at the beginning of the vote). The proposition thus holds. ◊

The protocol has also been partially described (without the challenges) in LOTOS and checked with a model checking strategy [Pecheur 1997]. This has led to the correction of many details in the protocol (taken into account in the description above) and the following main results:

- detection of deadlocks in case of synchronous use of the protocol;
- detection of messages left unanswered;
- complete description and formalisation of the management of surrogates and of the flushing of the various directories (A, P,…).

Note that some properties are not easily transferable to majority and intersection protocols. As a matter of fact, intersection and consensus are monotonous (if subscribers are added to a repository, this can only reduce the amount of knowledge stored) while majority is not.

Under the same assumptions, the CO4 protocol satisfies the requirements of [Gaspari& 1994] with regard to the users (they should not have to handle communication deadlocks, starvation issues and ressource management issues explicitly) in the asynchronous case if the memory in infinite (as the authors assume it). As a matter of fact, the strict protocol communications cannot loop because the protocol is organised along the tree of repositories and always proceeds either downward or upward.

7. Discussion and future work

The CO4 protocol as described above is based on several assumptions that are worth discussing. They are considered here from the more abstract to the more technical. They provide insight of the improvements of the protocol that can be taken into account.

Consensus vs. other aggregation policies. The choice of a consensus policy (as opposed to a majority policy for instance) is deliberate. It can be criticised because achieving consensus is not easy at all. However, there are several reasons for adopting this policy. From the technical viewpoint, consensus has very good properties: if a subscriber drops out, the repository is still consensual (this property is not shared by majority) and if every subscriber has a consistent view of the world, then consensus will be consistent. On the practical side, consensus has been used as a way to promote dialogue between subscribers instead of election tactics and minority gagging which is not in the ethic of the scientific community. Meanwhile, the protocol could easily be modified into intersection (no vote is necessary) or majority vote.

Consensus vs. private spaces. Many other systems offer private spaces rather than consensus. This allows to store in the repository proposals which are accepted by only some subscribers (with adequate labelling) and to bypass the consensual policy judged too strong. This should not be necessary in the CO4 system since the individual repositories do not have to contain the consensual one. On the opposite, we insist on the consensual aspect of the group base in order for subscribers to discover and acknowledge the disagreements instead of developing parallel and inconsistent private repositories. The strong policy could be softened by implementing discussion mechanisms such as those of gIBIS [Concklin 88].

Independence hypothesis. Hypothesis (d) states that the proposals must be independent. This is a very strong hypothesis since it prevents fairness to fully apply (one can submit a proposal which forbids other submissions by other subscribers). It is also problematic because independence concerns aspects which are not under the control of the protocol. Partial answers can be given to that problem: checking independence before submission, checking consistency only after a proposal has been accepted, providing a hook to the developers in order to deal with this on the application side. However, none of them is fully satisfactory and the protocol has been presented independently of that. It is noteworthy that the French national assembly should have such a way to deal with concurrent and dependant proposals when several amendments of a particular law are considered.

Complexity. The protocol relies on consistency tests which are known to be intractable in many representation languages (and evaluation can use revision which is far more complex). This is a matter of trade-off between consistency, expressiveness and complexity which is out of the scope of the CO4 protocol. If one wants to give up consistency, the protocol will work in the same way by implementing K?k so that it is always satisfied.

Acknowledgements and security. The current presentation of the protocol does not deal with issues such as security, authentication, acknowledgement and anonymity [Stodolski 1990]. Obviously they will have to be taken seriously if they are required. This does not seem to raise particular problems in the context of CO4. The call for comments procedure can easily ensure anonymous submission and anonymous replies while the accepted proposals can be credited to submitters.

Delay. Hypothesis (a) considers that answers are given in a finite amount of time. This is a very weak hypothesis and yet very difficult to ensure. Foreseen improvements of the protocol include the addition of time-out, payments and penalties which are often useful for constraining people to satisfy hypothesis (a) and getting rid of that hypothesis if necessary (by considering that subscribers which are too slow will have to register at new for instance). However, the protocol still does not care about social parameters such as hierarchic or consumer-producer relationships.

Verbosity. The protocol have been described in a form allowing to check rapidly if it is well-defined. Under that form it is quite verbose (e.g. the submitters of a proposal have to vote for it and to vote again in case of forward). It is planned to provide a language for expressing filters such that the user can decide to deal automatically with some routine messages (e.g. errors, notifications, call for comments on an irrelevant part of the repository). These filters should be integrated in such a way that they can be placed also on the group repositories through the same acceptance protocol. They have already been included in the current implementation.

8. Related works

The CO4 protocol can be characterised with regard to three ongoing research directions: software agents, groupware and distributed data servers. An extended comparison with other approaches can be found in [Euzenat 1995].

Software agents. The work on software agent is usually focused on how agents can make a deal (decide to put their force together) rather than how to supervise the behaviour of the agents. The “contract net” protocol [Smith 1980] is such a negotiation protocol. This led to theoretical and practical results about how to negotiate.

CO4 can be compared with SANP [Chang& 1994]. Both protocols are based on speech-acts plus a protocol and remain at the protocol level (they do not intend to model the intentions of the message senders). This approach allows to design an independent protocol and to prove it because it is not burdened by application details. The difference lays in the domains. SANP considers a protocol in an open world of negotiation. CO4 bounds the considered world by a registration protocol which identifies the actors as groups and/or individual repositories. Moreover, formalisms (rules instead of automaton) and implementations (speech-act dedicated transport layer instead of simple mail) differ. However, if the works are similar, here the formal properties of the protocol have been emphasised instead of its psychological or social relevance.

In summary, the CO4 protocol has the particularity of focusing on the maintenance of repositories and obeying very strict rules while in usual software agents the protocol is open to many tasks and thus observes more open rules. However, the present work opens interesting perspectives by providing built-in protocols which agents (software and human) may decide to fulfil and which can be supervised efficiently. There is room in software agent research for a very general interaction language with a denotational semantics independent from the protocols and libraries of specialised and protocols (using this language) that the agents may decide to use when necessary

Groupware. In contrast, groupware does not care about the collaboration between human and software agents and does not provide, in general, a policy for decision making, for instance, Lotus Notes or the last version of Microsoft Word includes the modification of a document by several users. However, the inclusion of these modifications is controlled by a very primitive policy:

- either there is no control and everyone can modify the document;
- or the control is the privilege of a particular user which can apply or not the modifications (with all the burden of consulting other people for clarification or conflict resolution for instance).

Several research projects (e.g. The coordinator [Flores& 1988], Treillis [Furuta& 1994] or Imagine [Haugeneder 1994]) investigate mediated communication between humans and computers according to formalised protocols. Trellis focuses on the tools for expressing these protocols through Petri nets. This system considers voting but in the context of decision making rather than constructing an artefact. An application closer to CO4 has been developed in the context of the Imagine project in order to manage appointments, but no details of the protocols are given.

Distributed servers and repositories. At last, there are two comparable lines of work concerning distributed data servers. In the distributed system and database community, important studies have been carried out in order to establish voting (or quorum consensus) protocols for authorising a modification in a repository [Garcia-Molina& 1985, Kumar& 1996]. This consists in requiring that a majority of tokens agrees on committing the modification for it to be accepted regardless to the other voters. This procedure is robust since it ensures that the modification is always accepted safely, but it cannot be proved that such a system works if one of the token-holders is deficient [Fisher& 1985]. Meanwhile, this work provides considerable theoretical results on the properties of voting procedures. However, it is different from the CO4 protocol because it only covers the voting aspect and the votes do not concern the content of a modification but the action of modifying (anyone is allowed to use a resource in any way as far as no one has previously reserved it).

The second line of work concerns the modification of knowledge bases by several users. The ontolingua server [Farquhar& 1997] is a system which allows users to modify a sort of repository called “ontology” through the World-wide web. The aim of the server is to achieve consensus between people about the vocabulary of a particular domain. However, the system has no tool for achieving consensus and the users can modify the repository content on an individual basis (with the same primitive policy as Lotus Notes). CGKAT [Martin 1996] provides a protocol for incrementally constructing a type hierarchy. This last protocol is provided as a set of algorithms without claim of completeness nor proof of correctness. Moreover, instead of consensus, the system tends to favour the “jardins secrets” (see §7) of the users by allowing them to express contradictory data on their sole behalf.

9. Conclusion

For the purpose of concurrently building an artefact (an article, a knowledge base, etc.) an architecture has been presented and a protocol has been set out. The protocol is closely tied to the way of interacting but totally independent from the artefact. It thus could be easily modified towards other ways of aggregating the artefacts and other groups of agents (e.g., in a scientific journal the voters and the subscribers are different). Such a protocol provides a way for contributors to modify the artefact though enforcing consistency and consensus of the choices made by the contributors. It is also fully functional and covers in a uniform way many aspects of the process (such as registering, cancelling or challenging contributions).

The protocol has been formally expressed with the help of simple rules. The benefits of the formal expression are the ease of understanding, modifying, simulating, testing and implementating of the CO4 protocol. The formal expression of the protocol allowed to prove that under particular assumptions, it enjoys the expected properties (anyone can submit a proposal at any moment, the proposal will reach a state of accepted or rejected in a finite amount of time and a proposal is accepted if and only if all the subscribers have accepted it). The next formalisation step could be to provide a denotational semantics for the protocol.

But such a protocol is not an end and it will have to prove its usefulness through real-life experiments.

If a consensual protocol could be acceptable in particular contexts, it is not universal. The CO4 protocol is not a protocol for any purpose. However, it is a first step towards the connection of human and software agents through a protocol which is monitored by software agents able to process automatically the routine messages. Such a technology can be applied to contract management with automatic payment, time-out messages, penalty management and processing of modification-clauses. Everything is possible if we care to include the human being in the protocol wherever there are decisions to be made.

10. Acknowledgements

This research has been supported by GREG (Groupement de Recherches et d'Études sur les Génomes) and by GdR CNRS «Informatique et Génomes» (CNRS: Centre National de la Recherche Scientifique) and a SERICS contract (SToRIA). The author thanks Alejandro Quintero (universidad de los Andes, Bogota, CO) who started developing the asynchronous extensions of KAPI, Loïc Tricand De La Goutte who finished it and developed the library and Charles Pecheur who helped clarifying many points left implicit when translating the protocol into LOTOS.

11. References

[Bolognesi& 1987] Tomaso Bolognesi, Ed Brinksma, Introduction to the ISO Specification Language LOTOS, *Computer networks and ISDN systems* 14(1):25-59, 1987

[Chang& 1994] Man Kit Chang, Carson Woo, A speech-act-based negociation protocol: design, implementation and test use, *ACM transactions on information systems* 12(4):360-383, 1994

[Cohen& 1995] Philip Cohen, Hector Levesque, Communicative actions for artificial agents, Proc. 1st ICMAS, Los Angeles (CA US), pp65-72, 1995

[Conklin& 1988] Jeffrey Conklin, Michael Begemann, gIBIS: A hypertext tool for explanatory policy discussion, *ACM transactions on office information systems* 6(4):303-331, 1988

[Euzenat 1995] Jérôme Euzenat, Building consensual knowledge bases: context and architecture, Proc. 2nd international conference on building and sharing very large-scale knowledge bases (KBKS), Enschede (NL), pp143-155, 1995

[Fagin& 1995] Ronald Fagin, Joseph Halpern, Yoram Moses, Moshe Vardi, Reasoning about knowledge, The MIT press, Cambridge (MA US), 1995

[Farquhar& 1997] Adam Farquhar, Richard Fikes, James Rice, The Ontolingua server: a tool for collaborative ontology construction, *International journal of human-computer studies* 46(6):707-727, 1997

[Finin& 1993] Tim Finin, Jay Weber, Gio Wiederhold, Michael Genesereth, Richard Fritzson, Donald MacKay, James MacGuire, Richard Pelavin, Stuart Shapiro, Chris Beck, Draft specification of the KQML agent communication language, 1993 `[ftp:ftp.cs.umbc.edu/]`

[Fischer& 1985] Michael Fischer, Nancy Lynch, Michael Paterson, Impossibility of distributed consensus with one faulty process, *Journal of the ACM* 32(2):374-382, 1985

[Flores& 1988] Fernando Flores, Michael Graves, Brad Hartfield, Terry Winograd, Computer systems and the design of organisational interaction, *ACM transactions on office information systems* 6(2):153-172, 1988

[Furuta& 1994] Richard Furuta, P. Stotts, Interpretated collaboration protocols and their use in groupware prototyping, Proc. 4th CSCW, Chapel Hill (NC US), pp121-131, 1994

[Garcia-Molina& 1985] Hector Garcia-Molina, Daniel Barbara, How to assign votes in a distributed system, *Journal of the ACM* 32(4):841-860, 1985

[Gaspari& 1994] Mauro Gaspari, Enrico Motta, Symbol level requirements for agent level programming, Proc. 11th ECAI, Amsterdam (NL), pp264-268, 1994

[Gaspari 1997] Mauro Gaspari, Knowledge level speech acts, Technical report 3, Department of computer science, University of Bologna, Bologna (IT), 1997

[Haugeneder 1994] Hans Haugeneder (ed.), IMAGINE (ESPRIT 5362) final project report, Siemens, München (DE), 1994

[Kumar& 1996] Akhil Kumar, Kavindra Malik, Optimizing the costs of hierarchical quorum consensus, *Acta informatica* 33(3):255-276, 1996

[Labrou& 1997] Yannis Labrou, Tim Finin, A proposal for a new KQML specification, Technical report 3, Computer science department, University of Maryland, Baltimore County (ML US), 1997

[Martin 1996] Philippe Martin, Exploitation de graphes conceptuels et de documents structurés et hypertexte pour l'acquisition de connaissances et la recherche d'information, PhD thesis, université de Nice-Sophia Antipolis, Nice (FR), 1996

[Narayanaswamy& 1992] K. Narayanaswamy, Neil Goldman, "Lazy" consistency: a basis for cooperative software development, Proc. 3rd CSCW, Toronto (CA), pp257-264, 1992

[Pecheur 1997] Charles Pecheur, Specification and verification of the CO4 distributed knowledge system using LOTOS, Rapport de recherche ####, INRIA Rhône-Alpes, Montbonnot (FR), 1997 (short version to appear in Proc. of the 12th IEEE international conference on automated software engineering, Incline Village (NE US), 1997)

[Peters 1995] John Peters, The hundred years war started today: an exploration of electronic peer review, 1995 `http://www.mcb.co.uk/literati/articles/hundred.htm`

[Rechenmann 1993] François Rechenmann, Building and sharing large knowledge bases in molecular genetics, Proc. 1st international conference on building and sharing very large-scale knowledge bases (KBKS), Tokyo (JP), pp291-301, 1993 `[ftp.imag.fr:/pub/SHERPA/articles/rechenmann93.ps.gz]`

[Smith 1980] Reid Smith, The contract net protocol: high level communication and control in a distributed problem solver, *IEEE transactions on computers* 29(12):1104-1113, 1980

[Stodolsky 1990] David Stodolsky, Consensus journals: invitational journals based upon peer consensus, *Datalogiske Skrifter* 29, 1990

12. Rule index

Rules are presented by their name followed by their triggering message sender and type.

Unité de recheche INRIA Rhône-Alpes
655 avenue de l'Europe, 38330 Montbonnot Saint-Martin (France)

Unité de recherche INRIA Lorraine - Technopôle de Nancy-Brabois - Campus scientifique
615, rue du jardin botanique - B.P. 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRA Rennes-IRISA - Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia-Antipolis - 2004 route des lucioles - B.P. 93 - 06902 Sophia-Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

http://www.inria.fr

ISSN 0249-6399

RR N°3260