Co4 protocol 1.0a reference manual

(Modified: 30/06/98 15:24, Printed: 30/06/98 15:30)

Sherpa project

655 avenue de l'Europe, 38330 MONTBONNOT-SAINT MARTIN, France

troeps@inrialpes.fr

Foreword

The CO_4 protocol provides a way to build a consensual artefact through submission of part of it to a common description. This reference manual provides an in depth presentation of the CO_4 protocol. It can be found on-line at `http://co4.inrialpes.fr/docs/co4-manual.html`.

About the CO4 protocol

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.

In order to extend such an organisation towards the knowledge itself (i.e. the declarative and digital representation of executable knowledge), a computer environment called CO_4 (for collaborative construction of consensual knowledge) is presented in [Rechenmann 1993, Euzenat 1995]. CO_4 is dedicated to the incremental and concurrent building of a knowledge base organising a set of various annotations around formalised knowledge. Annotations can be text, bibliography, image, experimental data from which the knowledge originates.

The CO_4 protocol allows the communication of pieces of the artefact between various repositories and their integration into consensual repositories. It 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 CO_4 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. 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 repository and so on, and so forth. For instance, a consensual representation could be achieved inside a particular firm before being submitted to the inter-institution repository. However, the behaviour of such a group repository is still subject to the consensual approbation of its subscribers.

CO_4 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 CO4 protocol is a result of the Sherpa project headed by François Rechenmann since 1988. The protocol has been designed by Jérôme Euzenat, model checked by Charles Pecheur and implemented by Loïc Tricand de La Goutte under the supervision of Jérôme Euzenat.

Beware: although other documents, anterior or posterior, can help to understand particular aspects of the system, **this reference manual is the only trustworthy documentation of the CO4 protocol 1.0 made available from July 1998**.

Organisation and notation

An interface to the structure manipulated by the protocol is provided through HTTP. Particular figures called Screendumps display this interface. They provide a picture of the HTTP browser screen together with the indications of the next screens when clicking on an active element (link or button). Some screens (mainly dialogues) are omitted since they do not carry a lot of information and other ones are reduced through white stripes across them.

Screendump <number>: <Description of the screen function>. <Additional information>
<link>: ..... see Screendump <other number>, p. <pagenumber>
…
<Button>:..... see Screendump <other number>, p. <pagenumber>
…

Plan

The manual is made of four main chapters. The first one, *Introduction to the CO4 protocol*, describes the philosophy of the system and its general behaviour. The second one, *User reference manual*, is dedicated to the presentation of the actual behaviour of the system including various practical things not specified in the protocol. In particular it is the reference manual for the CO4 HTML interface. The third one, *Understanding the CO4 behaviour*, provides some insight of the internal behaviour of the protocol. The last chapter, *Developer reference manual*, describes how to embed CO4 in a particular application. This will be turned into a tutorial.

A set of appendices is added to these four main chapters. They concern:

1) how to get and install the CO4 protocol;
2) the KAPI 2.6e LICENSE file;
3) the example of sessions;
4) the error messages used by the system;
5) the glossary of used terms;
6) an index of notions presented in the manual

The user is invited to read only the two main sections. The installer and/or newcomer should consult the Appendix 1 for a quick start. The programmer will read the two first sections and the last one. She may have to consult the third section only in case of deep modification of

the protocol. The programmer will access directly the manual from the function index at the end of the manual or by browsing the on-line manual through a WWW client (see Appendix 1).

Montbonnot, le 1er juillet 1998

Table of contents

FOREWORD **3**

ABOUT THE CO4 PROTOCOL 3
ORGANISATION AND NOTATION 4
PLAN 4

TABLE OF CONTENTS **6**

INTRODUCTION TO THE CO_4 PROTOCOL **8**

THE NETWORK OF REPOSITORIES 8
A GLIMPSE AT SUBMISSION FROM THE USER VIEWPOINT 9
PAPER SUBMISSION METAPHOR AS IMPLEMENTED BY CONSENSUAL REPOSITORIES 9

USER REFERENCE MANUAL **11**

VOCABULARY 11
REPOSITORY 11
PRIMITIVES 11
CUSTOMISING 13
INTERFACE 14

UNDERSTANDING THE CO_4 BEHAVIOUR **18**

CO_4 AND KAPI 18
ADDRESSING 18
STORAGE 19
WEB COMMUNICATION 20

DEVELOPER REFERENCE MANUAL **22**

CONNECTION WITH REPOSITORY MANAGEMENT SYSTEMS 22
BASIC APPLICATION PROGRAMMING INTERFACE 23
BUILDING AN HTTP SERVER 26
EXAMPLES 27

APPENDIX 1. HOW TO GET, INSTALL AND RUN THE CO_4 PROTOCOL **28**

APPENDIX 2. KAPI 2.6E LICENSE FILE **31**

APPENDIX 3. SCRIPT EXAMPLES **32**

SCRIPT SYNTAX 32
MINI 32
ERROR 32
ACCEPT 33
FORWARD_REG 33
MAIL 34

APPENDIX 4. ERROR MESSAGES **35**

APPENDIX 5. GLOSSARY..... 37

APPENDIX 6. FUNCTION INDEX..... 39

Introduction to the CO4 protocol

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 before providing the principles of the CO4 protocol form the viewpoint of the individual users and from that of consensual repositories.

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.

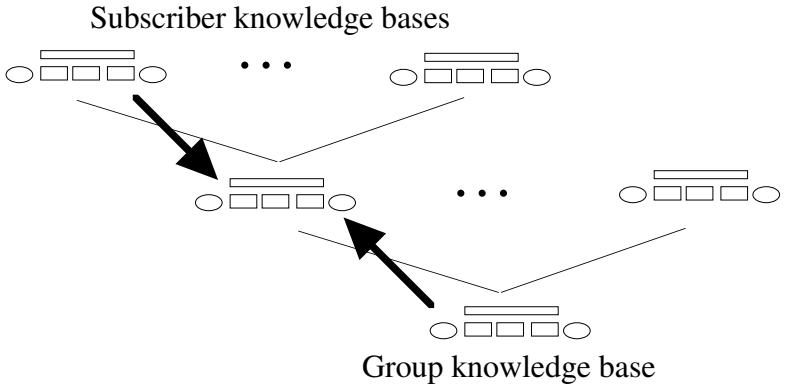


Figure 1. The hierarchical architecture and message flow (dark arrows). The repositories are organised in a tree whose leaves are individual repositories and nodes represent the consensus between of connected individuals. The downward types of messages include the submission of a proposal and the reports 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.

Some independent repositories can subscribe to group repositories as observers: the group repository sends to these repositories whatever is introduced in the repository they but cannot modify the group repository. 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.

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.

Paper submission metaphor as implemented by consensual repositories

Any system allowing to build some artefact must have a particular change policy. The CO4 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 receives 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 CO_4 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.

User reference manual

The user reference manual describes on a practical mode how to use the protocol by manipulating the various entities constructed by the protocol. This description is provided first abstractly before detailing the graphic user interface.

Vocabulary

The CO4 protocol makes use of various notions that must be understood in order to manipulate them. These definitions can also be found in the glossary.

Repository: a container of the description of the artefact to be built. In the following, the repository is a running process.

Group repository: a repository that is fully automated and that is meant to contain the consensual representation among its subscribers.

Agent name server (ANS): a process which has in charge the identification (and management of addresses) of the various repositories. It is identified by an URL. In CO4, the ANS also manages the data of the various repositories.

Mailbox: the set of messages received by a repository. It is stored by the ANS when the repository is shut down.

Initiative: a query issued by the current repository to its group repository.

Proposal: under the interface, a proposal is a query that has been sent by the group repository (as a CFC), to be accepted, rejected or challenged by its subscribers.

Call for comments (CFC): when the group repository receives an initiative, it sends it as a proposal to all the subscribers for a vote, and gathers all the answers. A CFC is an abstract representation of this voting procedure, and will be accepted, rejected or challenged.

Log: the set of proposals that have been accepted in the group repository but have not been introduced into the current repository for various reasons.

Surrogate: number identifying in a unique manner the message exchanged between repositories (the numbers are not unique, but the context — i.e. the kind of message and who issued it — warrants the identification of the message).

Repository

A CO4 repository is defined by its name. It must be declared to an agent name server (ANS). A repository is thus created by declaring:

— the ANS URL (e.g. `http://co4.inrialpes.fr:1080`);

— the name of the repository (e.g. `myrep`);

— the URL of the repository (e.g. `tcp://lotse.inrialpes.fr:5551`).

The content of the repository and all the data necessary for CO4 are stored by the ANS (see *Storage* in *Understanding the CO4 behaviour*).

Primitives

Once the repository has been created, there are various kinds of actions which can be performed. They are presented below.

General

General primitives (Screendump 1) deal with the system issues of starting and stopping a repository. These actions are not part of the CO_4 protocol but they are necessary to its everyday work.

`CREATE`: the repository is created and known to the ANS.

`DELETE`: the repository is discarded from the ANS together with its mailbox and content.

`SHUTDOWN`: the repository is no more active. Its content is saved by the ANS, and from now on, a message addressed to it will be stored in its mailbox.

`WAKE UP`: the repository is reactivated. Its content is read from the ANS, and its mailbox too.

`TURN TO GROUP REPOSITORY`: the repository is changed from an individual repository into a group repository.

Initiatives

Initiatives concern the CO_4 actions that can be taken by an individual repository. Once issued, they are stored in the set of initiatives of the repository. All but "evaluate" and "deny" are subject to acceptation by the other subscribers (thus require the use of a CFC).

`REGISTER` The repository subscribes to another repository (see Screendump 1), that will become its group repository. This can only happen if the individual repository does not already have a group repository, if the asked repository had been turned into a group repository, and if all its subscribers (if any) accept the new subscriber. Once accepted, the current repository will receive the complete content of the group repository and will have to answer all the proposals concerned by pending CFC issued by the group repository.

`EVALUATE` The repository asks its group repository to evaluate a particular proposal. The answer is "unanswered" until the group repository answers by "admissible", "contained" or "contradictory" (see `TEST` below).

`ACHIEVE` The repository asks its group repository to add a particular assertion to its content (see Screendump 5).

`FORWARD` The repository asks its group repository to issue a particular query. For example, forwarding a registration will cause the group repository to try to register. Forward actions can be nested.

`DENY` The repository tells its group repository to cancel a previously issued query. This can only apply if the concerned CFC has not yet been accepted or rejected.

Answers

Each proposal, initiative and CFC has a status which can be (see Screendump 4):

`UNANSWERED`	initial state;
`DENIED`	(only for initiative) when the issuer denied its initiative;
`ACCEPTED`	when it has been accepted;
`REJECTED`	when it has been rejected (the reason is then provided);
`CHALLENGED`	when it has been challenged (the alternative is then provided);
`ERROR`	when an error occurred (the error message is then provided).

Only proposals can be directly answered, and the answers will affect the corresponding CFC and initiatives. The possible answers are (see Screendump 4):

`ACCEPT`: the repository accepts the proposal.

REJECT: the repository rejects the proposal. It provides a comment explaining the rejection.

CHALLENGE: the repository accepts a part of the proposal, rejects another part and adds a new proposal to it.

Content management

Since the messages exchanged between the repository and its group repository contain some assertions, it is possible to manipulate them and to add them into its own repository content. Thus when some content is carried by a message, it is possible to apply the following actions to it (see Screendump 2):

TEST: the repository confronts the content of the message to its content. The result can be one of the following:

CONTAINED	when the assertion is already in the repository;
CONTRADICTORY	when the assertion is contrary to the repository;
ADMISSIBLE	when the assertion can be added to the repository.

APPLY: the content of the message is added to the repository content.

LOG: the content of the message is not added to the repository content. It is logged in a logbook.

Customising

There are several possibilities for modifying the default behaviour of the Co4 protocol. These simple customisations are currently handled by environment variables considered by the system.

Under UNIX, a variable *var* can be turned on by typing

```
setenv var 1
```

and turned off by typing

```
setenv var 0
```

These options will be saved, but the existence of the variables will override the saved options. The various options, detailed below, can also be accessed by the HTML interface (see Screendump 6).

Tracing

The default behaviour of the Co4 protocol is silent. However, it is possible to have all the messages sent by a repository printed. This is achieved by setting the CO4_TRACE variable to 1. Each message sent is printed under the following format:

```
B2 --(.... W=00)-register(tcp://lotse.inrialpes.fr:5552) --> B3  [register]
                B3 --(T=00 ....)-notify(accept) --> B2
```

meaning that the repository B2 sent to repository B3 a query corresponding to the register rule of the Co4 specification and that it received from B3 the notification of its acceptation.

Moreover, a [mbox] printed at the end of a line signifies that the receiver repository is down and that the message has been stored in its mailbox.

Setting CO4_TRACE to 0 will prevent the messages to appear.

Housekeeping

When a CFC, a proposal or an initiative has reached a final status, it is discarded from CO4 data structures, to save space. Setting `CO4_KEEP_ALL` environment variable to 1 will keep them, while setting it to 0 will discard them (by default).

Anonymity

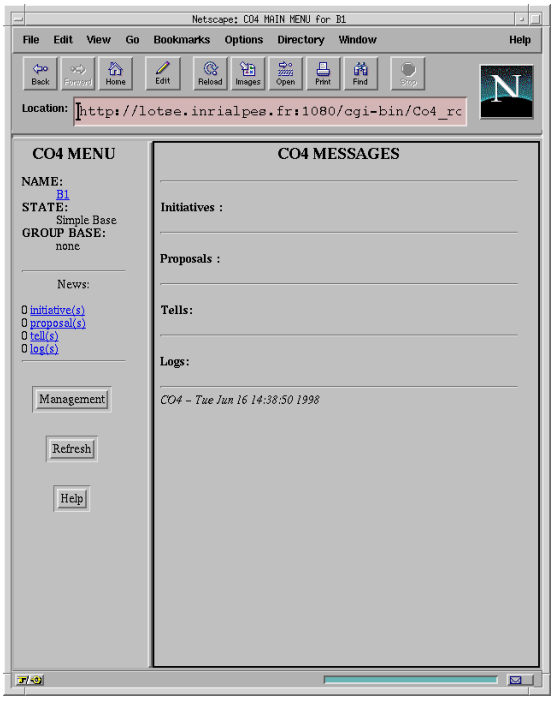
The default behaviour, when issuing a CFC consists in transmitting the initial proposal (containing the identification of the issuer). It is possible to mask this information during the voting process by setting the `CO4_ANONYMOUS` environment variable to 1. A value of 0 will show this information. This only matters in group repositories.

Automatism

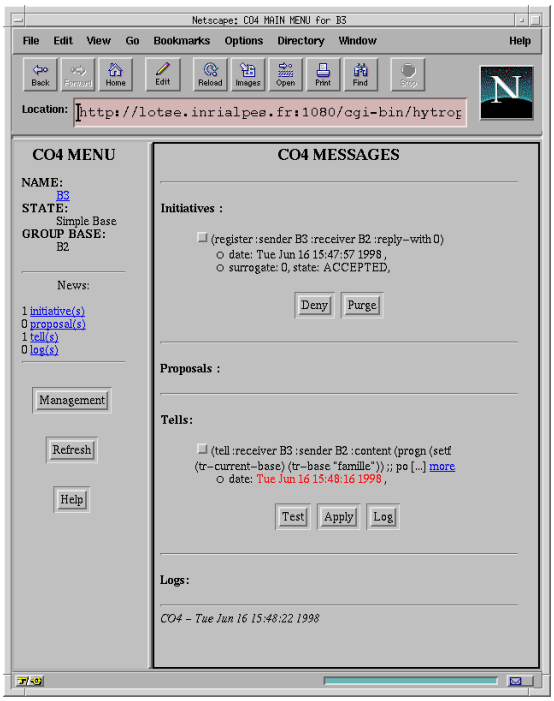
An individual repository can prefer to automatically accept propositions, or to automatically accept proposals due to itself, instead of voting every time. Just setting `CO4_ACCEPT_ALL` or `CO4_ACCEPT_MINE` to 1 will make CO4 work more automatically, while setting them to 0 will give more control to the user.

Interface

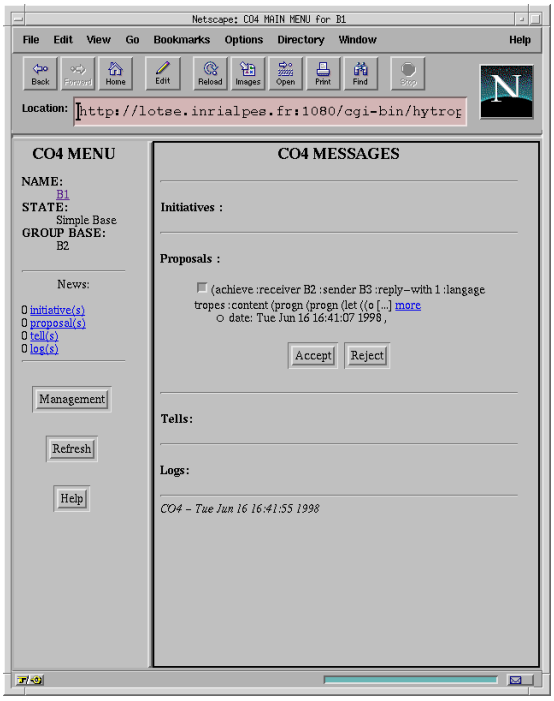
The CO4 package contains a Web browser interface, that provides two frames, one to display CO4 main menu, the other to display either CO4 messages, or the repository content part. Below the various panels of the graphic user interface are presented. They concern three repositories (whose content is a knowmedge base expressed in the TROEPS system) named B1, B2 and B3. B2 is a group repository to which B1 and B3 will subscribe.



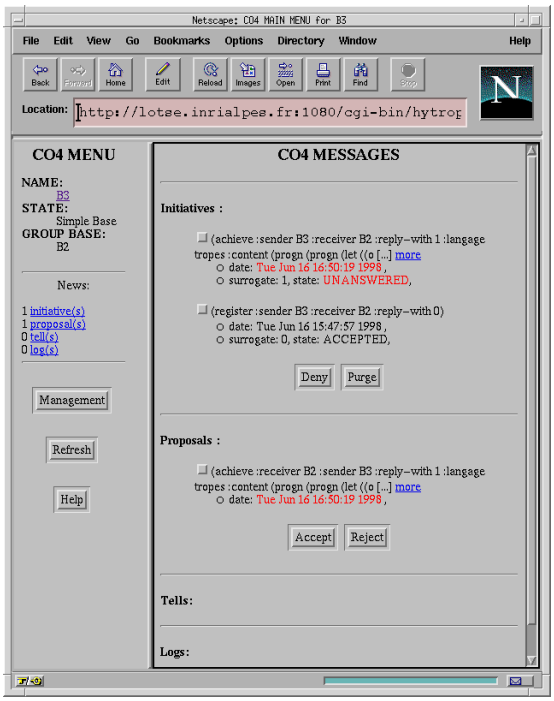
Screendump 1: Initial CO4 panels. The repository B1 has no group repository. Thus it must first register to a group repository.



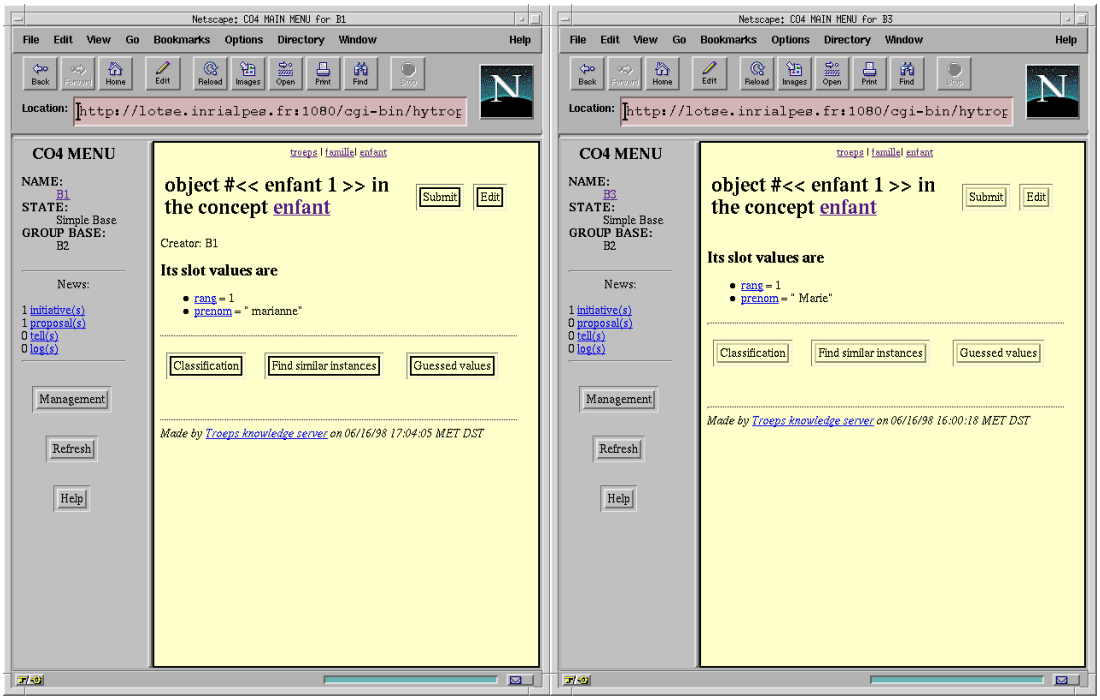
Screendump 2: After that B1 has subscribed. B3 subscribes and then receives information from its new group repository: answer to the initiative (subscription), proposals to answer and tells to integrate in its content.



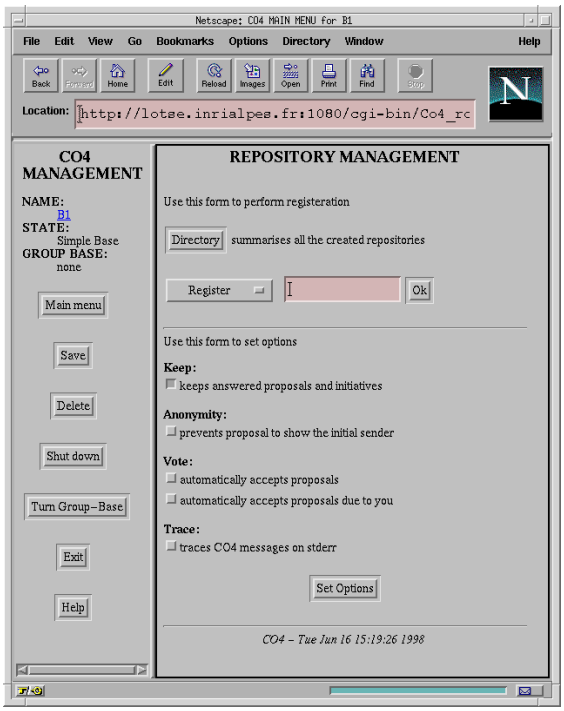
Screendump 3: More proposals to answer will appear into B1 interface as long as B3 sends new ones.



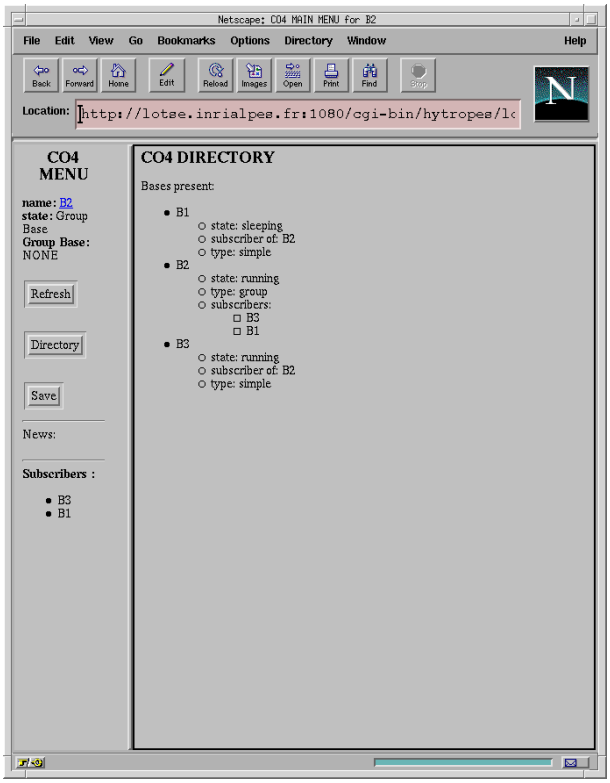
Screendump 4. More and more information is communicated to the repository (B1) included new status of initiatives (ACCEPTED) and new proposals.



Screendump 5: The content of the repositories can be displayed in the CO4 panel. The content is a knowledge base from the TROEPS system. One can observe the ”Submit“ button allowing to submit the knowledge to the group knowledge base and the name of the creator of the objects (B1) in the repository B3. Since the content of the repository can be different, ne “prenom“ of the displayed object is indeed not the same.



Screendump 6: Configuration of an individual repository (B1). It is possible to register to a group repository or to set the various values presented above. It is also possible to access all the system primitives (“delete”, “shutdown”, “turn-to-group”, etc.).



Screendump 7: Management of repositories. It is possible from the HTTP interface of a group repository to access the state of the subscriber repositories.

Understanding the CO_4 behaviour

This section presents the system behaviour of CO_4. It can help to understand the actual implementation and installation.

CO_4 and KAPI

The CO_4 protocol takes advantage of the KAPI messaging services, and like KAPI uses an Agent Name Server (ANS) supported by an HTTP server to manage clients as agents. It thus requires KAPI to be installed. The CO_4 package is made of two libraries, `libCo4`, the main one, and `libKapi2`, built on top of KAPI which adds some functions like mailboxes, so KAPI (`libkapi.a`) is still required. The package also contains 6 cgi-scripts, to deal with ANS requests:

`Co4_register` and `Co4_unregister` to manage agent.

`Co4_deleteFile` and `Co4_writeFile` to manage saved files.

`Co4_addMail` to save mail

`Co4_directory` to retrieve existing repositories

`Co4_routingScript` to route HTTP requests from a browser.

These scripts are installed in the CGI directory of the HTTP server.

Addressing

A CO_4 repository is a process running as a demon, exchanging KQML messages with other CO_4 repositories, for example a demand of registration. A TCP port ("KQML port" in the remainder) is dedicated to this repository to communicate with its peers and the corresponding URL (host and port) is recorded by the ANS at the creation of the repository. This allows KQML messages to only deal with repository nicknames, without knowing their physical address.

Before each emission of a message, the ANS is requested for the URL of the target, as shown in Figure 2. Several repositories can run on the same host, by choosing different KQML ports.

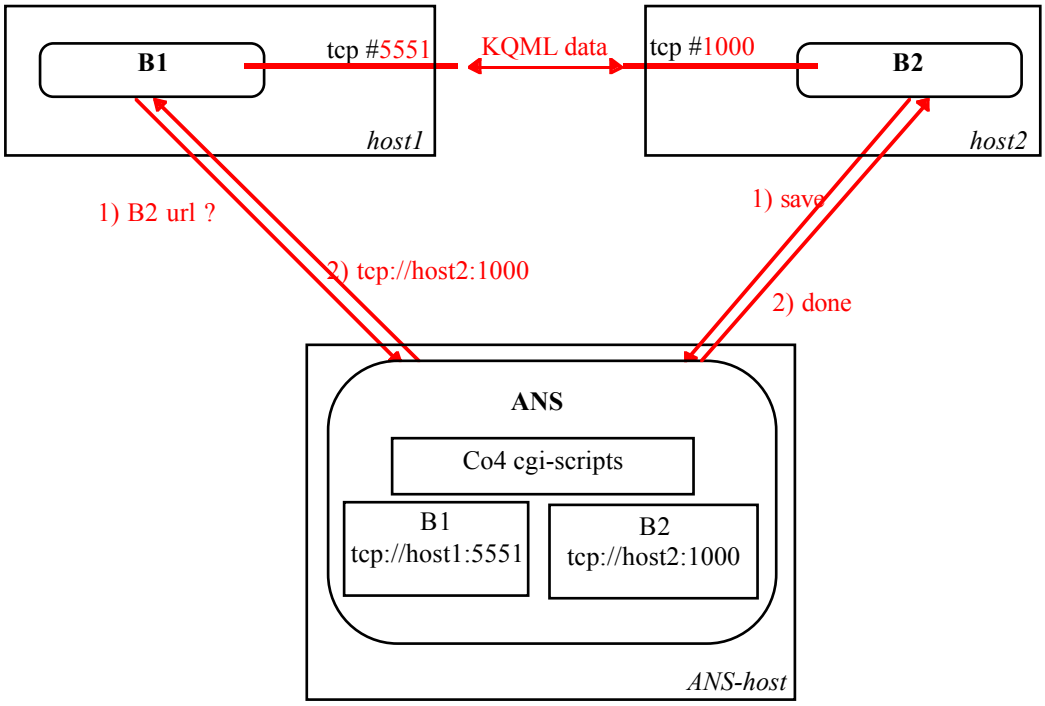


Figure 2. Communication scheme of the Co4 protocol. Two repositories B1 (located on host1) and B2 (located on host2) communicate through an ANS (located on ANS-host). B1 requests the URL of B2, so the ANS retrieves in the files of B2 its URL, tcp://host2:1000, and returns it to B1. On its side, B2 asks to save data, so the ANS returns to it the result of the corresponding script execution (here *done*).

Storage

The ANS also deals with repository management requests, because it stores repository data, by the way of the cgi-scripts presented above. The ANS used by CO4 has been enriched to store all data relating to the repositories. So, let a HTTP server be on a host, with a scripts directory located in `$HTTP_BIN_ROOT`, and a documents directory located in `$HTTP_DOC_ROOT`. CO4 installation will copy the 6 `Co4_` scripts in `$HTTP_BIN_ROOT`, and make a directory named `CO4` in `$HTTP_DOC_ROOT`. Each registered repository will have an account in this directory `CO4`, to store:

- a file named `url` containing the URL used by KAPI (see KAPI documentation for more information);
- a file named `infos` containing all the information needed by CO4 for a successful wake up;
- a file named `mbox` containing pending mails;
- a file named `active`, whose presence indicates it the repository is active or not;
- a directory named `knowledge` where all the content is stored.

Group and individual repositories are not distinguished in the architecture.

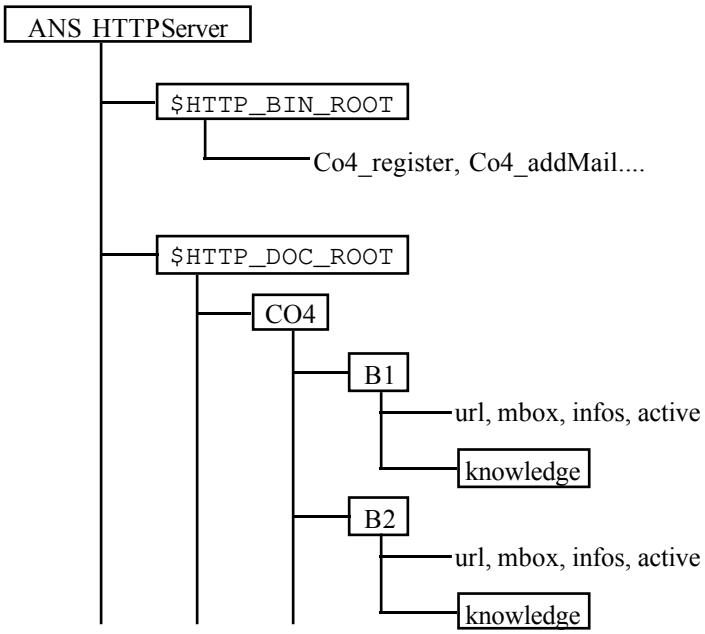


Figure 3. The various directories and files that can be found in the ANS HTTP server.

These files and directories are created and deleted by the various cgi-scripts that have been added in the server (see installation). These modifications correspond to CO_4 orders:

— a `CREATE` will create a new directory named after the new repository, and create `url` and `active`.

— a `DELETE` will erase all the repository directories.

— a `SHUTDOWN` will delete the `active` file, save a new `infos` file, and save all the `knowledge` files.

— a `WAKEUP` will read `infos`, read and delete `mbox` if it is not empty, load `knowledge` files, and create `active`.

Web communication

CO_4 provides a HTML interface to visualise the repository and to manipulate the CO_4 primitives. This repository behaves as a server for some HTTP clients (web browsers). It uses another TCP port (server port in the remainder), different from the KQML port, to communicate with its clients, via a routing script, named `Co4_routingScript` and located in `$HTTP_BIN_ROOT`. This script receives a URL, extracts the server port from it, sends and receives data from the repository through that port, and finally, returns HTML code to the browser to be displayed, as shown in Figure 4.

Several servers can run on a same host, by choosing different server and KQML ports.

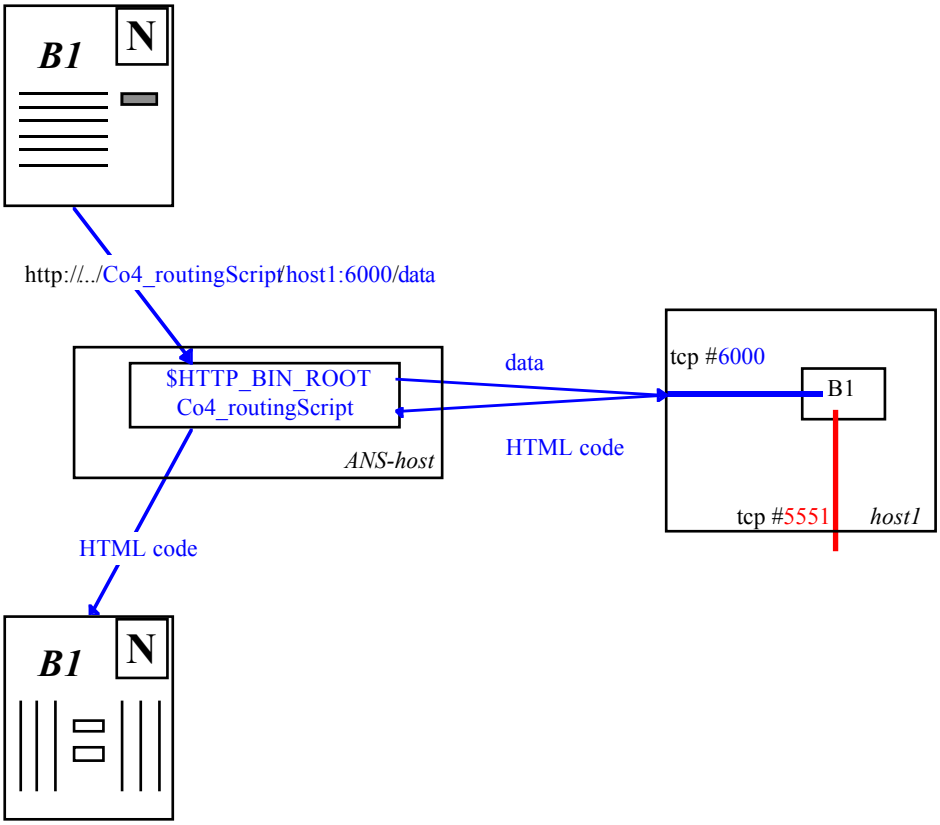


Figure 4. A click on the HTML interface of B1 displayed by a browser causes a call to the routing script, that retrieves the server port of B1 (tct://lotse:6000) from the URL, then sends data to B1, gets some HTML code from it, and finally returns this code to the browser to be displayed. The KQML port (tcp:5551) is always active in order to let B1 receive KQML messages send by other repositories.

Developer reference manual

This section concerns the integration of the CO4 protocol, as a library, in a new application. It is a C developer manual. You must keep in mind that because of the use of KAPI, a KAPI error can occur. They are not listed below, and you must refer to the documentation available in the KAPI directory.

Connection with repository management systems

In order to build an application using CO4, the developer must provide some functions defined as extern in `libCo4`. These functions deal with content management, while CO4 only deals with transport. This covers initialisation, loading, saving, addition of content. To help writing these functions, CO4 provides utilities functions to manage repository files located in the ANS.

Content management

extern int **InitKnowledge**()

called by `Co4CreateRepository` and `Co4WakeUpRepository`.

Initialises the content of the repository. Returns `CO4_OK` if successful.

extern int **LoadKnowledge**()

called by `Co4WakeUpRepository`.

Recovers the content of the repository from the ANS. Returns `CO4_OK` if successful.

extern int **SaveKnowledge**()

called by `Co4ShutDown`.

Saves the content of the repository on the ANS. Returns `CO4_OK` if successful.

extern char* **DumpKnowledge**()

Returns a string representing the complete content of the repository. This string will be sent by CO4, then freed, so it must have been allocated.

extern int **AddKnowledge**(char**contentString*)

Extracts the content from its string representation *contentString* and adds it to the current repository. By extension, must be able to treat the string returned by `DumpKnowledge`. Returns `CO4_OK` if successful.

extern int **ConfrontKnowledge**(char**contentString*)

Extracts the content from its string representation *contentString* and confronts it with that of the current repository. Returns as `Co4Test`:

`CO4_STATUS_ADMISSIBLE`,

`CO4_STATUS_CONTAINED`,

`CO4_STATUS_CONTRADICTORY`.

extern char* **IntersectKnowledge**(char* *k1*, char* *k2*)

Returns an allocated string denoting the intersection of two pieces of knowledge, contained in two strings *k1* and *k2*. This string will be freed by CO4.

extern char* **MergeKnowledge**(char* *k1*, char* *k2*)

Returns an allocated string denoting the merge of two pieces of knowledge, contained in two strings *k1* and *k2*. This string will be freed by CO4.

File management

Some more functions might be necessary in order to deal with files stored in the ANS server and to retrieve and store them. They are provided here.

int **Co4GetFile**(char* *ANSfilename*, char* *localFilename*)

Copies the file *ANSfilename* located in the directory of the current repository by the ANS, to *localFilename*. Returns

CO4_OK: the file *ANSfilename* has been successfully transferred.

CO4_NO_FILE: the file *ANSfilename* cannot be accessed by the ANS.

CO4_ERROR: an error occurred.

int **Co4SendFile**(char* *localFilename*, char* *ANSfilename*)

Copies the local file *localFilename* to the file ANS *filename* located in the directory of the current repository by the ANS. Returns

CO4_OK: the file local*Filename* has been successfully transferred.

CO4_NO_FILE: the file *localFilename* cannot be accessed.

CO4_ERROR: an error occurred.

int **Co4DeleteFile**(char* *ANSfilename*)

Deletes the file ANS *filename* located in the directory of the current repository by the ANS. Returns:

CO4_OK: the file ANS*filename* has been successfully deleted.

CO4_NO_FILE: the file ANS*filename* does not exist by the ANS.

CO4_ERROR: an error occurred.

int **Co4DeleteAllFiles**()

Deletes the whole directory of the current repository by the ANS. Returns:

CO4_OK: the directory has been successfully deleted.

CO4_ERROR: an error occurred.

Basic application programming interface

The main header file co4.h defines the CO4 protocol application programming interface (API):

Repository management

int **Co4CreateRepository**(char* *ANSurl*, char* *name*, char* *KqmlUrl*, int *erase*)

Tries to create the repository, via the given ANS. If it already exists (same ANS and same name), its behaviour depends on *erase* : if it is false, an error is returned. If it is true, all data are erased, and the repository created again. The ANS will communicate with the repository trough the URL given by *KqmlUrl*.

Returns CO4_OK if successful, otherwise one of the following error codes is returned:

CO4_NO_ANS_RESPONSE: the ANS does not respond.

CO4_NO_CONNECTION: *KqmlUrl* does not respond

CO4_IS_ACTIVE: the repository is still active.

CO4_ALREADY_EXISTS: *erase* is 0, and the repository already exists.

CO4_ERROR another error occurred.

int **Co4WakeUpRepository**(char* *ANSurl*, char* *name*, char* *KqmlUrl*)
Wakes up an existing repository, i.e. fetches repository content, info and processes pending messages. If *KqmlUrl* is NULL, uses the previous existing URL.
Returns CO4_OK if successful, otherwise one of the following error codes is returned:
CO4_NO_ANS_RESPONSE: the ANS does not respond.
CO4_NO_CONNECTION: *KqmlUrl* does not respond
CO4_IS_ACTIVE: the repository is still active.
CO4_DOESNT_EXIST: the repository does not exist.
CO4_ERROR: another error occurred.

int **Co4SaveRepository**()
Saves the repository.
Returns CO4_OK if successful, CO4_ERROR otherwise.

int **Co4ShutDownRepository**()
Calls Co4SaveRepository and shuts the repository down.
Returns CO4_OK if successful, CO4_ERROR otherwise.

int **Co4DeleteRepository**()
Discards the repository.
Returns CO4_OK if successful, CO4_ERROR otherwise.

void **Co4TurnIntoGroup**()
Turns the current repository into a group repository.

Enquirers

char* **Co4GetGroupName**()
Returns the name of the group repository of the current repository (default NULL).

int **Co4IsGroupRepository**()
Returns 1 if the current repository is a group one, 0 otherwise.

Message management

int **Co4GetMessage**(char **msg*)
Tries to read a waiting message. If any, it is copied in *msg*, and the function returns CO4_OK, CO4_ERROR otherwise.

void **Co4ProcessMessage**(char* *msg*)
Processes a message.

Orders

Here are the functions for performing CO_4 orders like “register”, “evaluate”, “accept”, etc. All functions can return CO4_UNKNOWN_RECEIVER if the receiver is not known by the ANS, CO4_UNREGISTRED if the current repository has not registered yet (all but Co4Register) and CO4_ERROR, denoting an internal error. Others errors are listed below.

Beware of the fact that return status only concerns the request emission and not the result of the request.

int **Co4Register**(char* *grpname*)

The current repository wants to register to *grpname*. Returns CO4_OK if the demand is successful, or an error code:

CO4_ALREADY_REGISTERED: already has a group repository.

CO4_BEING_REGISTERED: no answer yet

CO4_SELF_REGISTERED: a repository cannot register to itself.

int **Co4Achieve**(char* proposal)

Submits a new piece of knowledge to the group repository.

Returns CO4_OK, or one of the common errors.

int **Co4Evaluate**(char* proposal)

Makes the group repository evaluate a piece of knowledge.

Returns CO4_OK, or one of the common errors.

int **Co4Forward**(char* *order*, int *depth*)

Forwards a KQML *order* to *depth* levels forward, a *depth* of 0 meaning to the group repository, a *depth* of 1 meaning to the group repository of the group repository, and so on.

Returns CO4_OK, or one of the common errors.

int **Co4Deny**(Co4Surrogate *n*)

Denies a previous ACHIEVE initiative or REGISTER, and so makes the group repository cancel its CFC. Returns CO4_OK, or an error code:

CO4_NO_SUCH_INITIATIVE: there is no initiative corresponding to the surrogate *n,*

CO4_CANT_DENY: the initiative does not correspond to an ACHIEVE or a REGISTER,

CO4_ALREADY_ANSWERED: the initiative is no more in a waiting state.

int **Co4Accept**(Co4Surrogate *n*)

Accepts a proposal. Returns CO4_OK, or an error code:

CO4_NO_SUCH_PROPOSAL: there is no proposal corresponding to the surrogate *n*,

CO4_ALREADY_ANSWERED: the initiative is no more in a waiting state.

int **Co4Reject**(Co4Surrogate *n*, char* *comment*)

Rejects a proposal with an explaining comment. Returns CO4_OK, or an error code:

CO4_NO_SUCH_PROPOSAL: there is no proposal corresponding to the surrogate *n*,

CO4_ALREADY_ANSWERED: the initiative is no more in a waiting state.

int **Co4Challenge**(Co4Surrogate *n*, char* *new_proposal*)

Challenges a proposal with an other KQML one. Returns CO4_OK, or an error code:

CO4_NO_SUCH_PROPOSAL: there is no proposal corresponding to the surrogate *n*,

CO4_ALREADY_ANSWERED: the initiative is no more in a waiting state.

int **Co4Apply**(Co4surrogate *n*)

Applies a new piece of knowledge sent by the group. Returns CO4_OK, or an error code:

CO4_NO_SUCH_TELL: there is no tell corresponding to the surrogate *n*,

CO4_CONTAINED if the knowledge is already present, or

CO4_CONTRADICTORY if it contradicts the repository.

int **Co4Log**(Co4surrogate *n*)

Logs a new piece of knowledge sent by the group. Returns CO4_OK, or an error code:

CO4_NO_SUCH_TELL: there is no tell corresponding to the surrogate *n.*

int **Co4Test**(Co4surrogate *n*)

Tests a new piece of knowledge sent by the group. Returns `CO4_NO_SUCH_TELL`, or one of the following

`CO4_CONTAINED`: the knowledge is already present,

`CO4_CONTRADICTORY`: the knowledge contradicts the repository,

`CO4_ADMISSIBLE`: the knowledge is admissible.

Debugging and options

extern int **Debug**

Forces KAPI to display low level information concerning TCP or HTTP communication (KAPI variable). Setting it to `NoDebug` (default) will hide these information. (see `kapi.h` for its possible values)

int **oTrace**
int **oKeepAll**
int **oAnonymous**
int **oAcceptAll**
int **oAcceptMine**

These variables are related to the `CO4_*` environment variables (see *Customising* in the *User reference manual*). If the repository is waken up, the values of these variables are restored. If it is created, they are all set to 0 (false). Then they are re-computed from their related environment variables, if present.

int **oTellKnowledge**

In some cases, depending on knowledge, only already existent knowledge can be submitted. So, when the knowledge has been accepted by all, it is normally told to each subscriber, but here not to the initial repository, because it already exists. To tell CO4 whether it must tell or not the knowledge, set `oTellKnowledge` to 1 or 0.

The default behaviour is to tell knowledge. This variable is also saved and restored.

Building an HTTP server

CO4 provide a HTML interface to be used by a browser via the Web. To do so, a CO4 server must be created to communicate with the browser via a routing cgi-script. The file `server.h` contains the necessary functions.

void **Co4ServerInit**(char* *routingScriptUrl*, int *port*, int (**hook*)(char*), char* *knowledgeHome*)

Initialises the server with the following parameters:

routingScriptUrl: the URL of the routing script called by the browser (see Figure 4). For instance, "`http://lotse.inrialpes.fr:1080/cgi-bin/co4exemple_server`";

port: the TCP port used by the server to communicate with the routing script, e.g. 6666;

hook: the name of the function that will handle the queries to the server (e.g. `co4hook`). Its argument string is the remainder of the URL passed to the script. It is formatted as "`<keyword>/<rests>`". If the hook function does not recognise `<keyword>`, it returns 0, otherwise it takes the action corresponding to `<rests>`, then returns 1;

knowledgeHome: a string "`<keyword>/<orders>`", that the hook function will recognise, to display the content the repository.

```
Co4ServerInit("http://lotse.inrialpes.fr:1080/cgi-bin/co4exemple_server",
              6666,
              co4hook,
              "troeps/home-page");
```

void **Co4ServerLoop**()

Prints the initial URL to be browsed, corresponding to CO_4 home page, then loops through CO_4 messages, but also requests from the routing script, answering to them.

void **Co4ServerStop**()

Stops the loop initiated by `Co4ServerStart`.

Char ***gUrlPrefix**

The prefix to be concatenated to the string `"/<keyword>/<orders>"` to make a valid URL understandable by the routing script first, and then by the hook *hook*.
For example, the CO_4 home page URL is "*gUrlPrefix*/CO4/home".

Examples

Two examples are available, one basic, the other using a HTTP server. Both use the same content representation and same content management functions, that can be found in `examples/common/knowledge.c`. The interest of these examples is mainly to show the use and implementation of the functions of the API presented above.

Basic example

An example of the basic capabilities (no server) is available in `examples/basic`. This example launches repositories, that will communicate between them, tracing on a window their queries and answers. Some examples correspond to the CO_4 specification examples, the rest demonstrate other aspects of CO_4: use of mail boxes and error management. See the `README` file and the appendix 3.

Server

An example of server is available in `examples/server`. This example launches a repository as server, to let a browser communicate with it. The server can be created or waken up, and implements a small hook to perform various actions. See the `README` file.

Appendix 1. How to get, install and run the CO_4 protocol

Requirements

The CO_4 protocol is being developed on Sun SparcStations under SunOS 4.4. It runs and has been tested under SunOS 4.3, 4.4, 5.4 (a.k.a. Solaris) and 5.5; it can be ported to other platforms either by INRIA or by yourself by recompiling the sources. It has to be used as a library linkable to other programs.

The CO_4 protocol has been developed with KAPI and C. It is thus made of a set of C programs that should be compilable with any C compiler as soon as KAPI libraries are available.

In order to use KAPI and its extension, there must be a HTTP server running somewhere for which the installer has the privilege to add new cgi-scripts and new directories.

Getting the CO_4 protocol and its documentation

The CO_4 protocol and its documentation are available for public distribution under the terms of the COPYRIGHT file provided with the distribution. They can be retrieved through anonymous FTP (file transfer protocol) on the machine `ftp.inrialpes.fr` under the names:

```
/pub/sherpa/rapports/co4-manual.ps.gz
/pub/sherpa/logiciels/troeps/co4.tar.gz
```

Our HTTP server provides various information (among which the latest release and the on-line manual). The URLs are:

```
http://www.inrialpes.fr/sherpa/
http://co4.inrialpes.fr/docs/co4-manual.html
```

Do not hesitate to browse anywhere on our site (beginning with `http://co4.inrialpes.fr`); a lot of information about new developments is available there.

Installing the CO_4 protocol

The CO_4 protocol can be installed anywhere in your system provided that you follow the instructions below. The files can be obtained by:

```
$ gunzip co4-manual.ps.gz
$ gunzip co4.tar.gz | tar xf -
```

In the following, let `$CO4DIR` be the location of CO_4 directory created by this installation. The content of the `$CO4DIR` (so the content of the tar-file) consists in the following files and directories:

`COPYRIGHT`	the INRIA Copyright notice;
`README`	the name says it;
`examples`	contains some sample repositories (useful for building your own);
`sparc_4_4.0`	contains C sources and libraries for SunOS 4.3/4.4 port;
`sparc_5_gnu`	contains C sources and libraries for SunOS 5.4/5.5 (Solaris) port with the gnu compiler;
`src`	contains the C sources for the CO_4 protocol library;

`kapi2.6e`	contains the complete package for the KQML Application Programming Interface release 2.6e from EIT/Lockheed (see inside LICENSE notice);
`config`	contains configuration files for compilation;
`doc`	user and developers last minute documentation;

There are two specific installation steps since they can be carried out on independent different machines.

Setting up the ANS

— log to the HTTP server, where you want to set the CO_4 ANS.

— edit `config/http_locations` to set:

`HTTP_BIN_ROOT` to the cgi-scripts root path of the HTTP server, where the CO_4 scripts will be copied.

`HTTP_DOC_ROOT` to the document path of the HTTP server, where the `CO4` directory containing all data will be created.

— type in:

```
$ cd $CO4DIR/src
$ make http
```

The required cgi-scripts are then compiled and copyied to the CGI directory and the `CO4` directory is created under the server root.

Compiling the CO_4 library

— log to the client host, where CO_4 has to be compiled, and type:

```
$ cd $CO4DIR/src
$ make libco4
```

Running the examples as scripts

The scripts are provided in a following appendix 3.

```
$ cd $CO4DIR/examples/basic/src
$ make
$ cd ..
$ ./run-script <CO4_ANS_URL> SCRIPTS/<file>
```

The system will open several X-terminals for displaying the result of each repository execution (in the trace format, see p.13).

Running the example as a HTTP server

```
$ cd $CO4DIR/examples/server/src
$ make
$ cd ..
$ ./create-server <CO4_ANS_URL> server_name
or
$ ./wakeup-server <CO4_ANS_URL> server_name
```

Several repositories can be created with these actions. The system provide the URLS of the created repositories. These repositories contain an integer value as content. They can be manipulated through a HTTP client (see *Interface*).

Using the CO_4 protocol libraries

The Co_4 protocol can be used from an alien language (C, C++, etc.). For that purpose, the Co_4 protocol is supplied with several libraries (`libkapi`, `libKapi2` and `libCo4`) along with the corresponding include file (`co4.h`) which enables its use from these languages.

Reporting bugs/getting advice

This can be done preferably through mail to `troeps@inrialpes.fr`.

Appendix 2. KAPI 2.6e LICENSE file

Here is the content of the KAPI 2.6e LICENSE file:

Copyright 1994, 1995 Enterprise Integration Technologies Corporation and Lockheed Missiles and Space Company, Inc.

Enterprise Integration Technologies Corporation (EIT) and Lockheed Missiles and Space Company, Inc. (Lockheed) grant to the possessor of this copy (the Licensee) a royalty-free, nonexclusive license to use, copy, modify and distribute both the binary and source code of the KQML KAPI and any related documentation (the Software) for academic, research and internal business purposes only and subject to the following conditions:

1. By using or copying the Software, Licensee agree to abide by the terms of this Agreement.

2. Title and copyright to the Software remain with EIT and Lockheed and Licensee agrees to preserve the same.

3. Licensee acknowledges that the Software is still in the development stage and that it is being supplied "as is", without any accompanying services from EIT or Lockheed. EIT and Lockheed MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. By way of example, but not limitation, EIT and Lockheed MAKE NO REPRESENTATIONS OR WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE THE SOFTWARE WILL NOT INFRINGE ON ANY PATENTS OR OTHER RIGHTS. EIT and Lockheed shall not be held liable for any liability nor for any direct, indirect or consequential damages with respect to any claim by Licensee or any third party with respect to this Agreement or any use of the Software.

4. Licensee may distribute the Software or any derivative work based on the Software to third parties provided (1) all copyright notices and this Agreement appear on all copies, and (2) no charge is associated with such copies. Unless Licensee labels the derivative work with the notice "No Grantback", Licensee agrees to grant EIT and Lockheed a royalty-free nonexclusive license to use, copy, modify and distribute any modifications to the Software owned by Licensee and distributed to third parties as a derivative work under this Agreement.

5. If Licensee distributes a derivative work, then Licensee also agrees to (1) notify EIT and Lockheed of the distribution, and (2) clearly notify users that such derivative work is a modified version and not the original EIT and Lockheed Software.

6. Any Licensee wishing to make commercial use of the Software should contact EIT and Lockheed to negotiate an appropriate license. Commercial use includes (1) integration of all or part of the source code into a product for sale or license by or on behalf of Licensee to third parties, or (2) distribution of the binary or source code to third parties that need it to utilise a commercial product sold or licensed by or on behalf of Licensee.

7. Licensee agrees to abide by all applicable laws of the United States, including, but not limited to, copyright and export control laws.

8. EIT and Lockheed have the right to terminate this license immediately by written notice upon Licensee's breach of, or non-compliance with any of its terms. Licensee may be held legally responsible for any copyright infringement that is caused or encouraged by licensee's failure to abide by the terms of this license.

Appendix 3. Script examples

These scripts can be found in `co4/examples/basic/SCRIPTS`. They can be invoked in any order. They trace (using `CO4_TRACE`) the communication between the various repositories involved in several windows. Some examples corresponds to the CO4 specification examples (they concern repositories containing numbers). The syntax for running a particular script is:

```
$ ./run-script ANSUrl SCRIPTS/name_of_script
```

where *ANSUrl* is the URL of the CO4 ANS, that must have been installed before.

Script syntax

The script syntax is very raw. It is given here only for inteligibility purposes and not as a scripting manual.

The first line of the script is relevant since it allows the system to compute the size that the X-terminal will occupy. Otherwise, whatever is placed after a sharp (`#`) is considered as a comment and not futher interpreted.

The script is interpreted line after line. Each line consists in three parts: the repository which is concerned by the line, the triggering message and the action. The repository is identified by its name. The triggering message is made of two part separated by a colon (`:`): the name of the repository which sent the message and the three first letters of the performatives. This message can be void identified by void fields (filled by underscores, _) in which case the action takes place immediatly. The action is specified by an identifier and arguments in a row.

Mini

The `Mini` script tests the registration of one base to another and closing of the connexion.

```
# 2 repositories
# this test shows the communication between 2 base"

B1  __:___     group 2  #B1 becomes group rep., with 2 as knowledge

B2  __:___     register B1
B2  B1:tel     eval 2

B2  B1:not     signal
B2  __:___     exit

B1  B2:sig     exit
```

Error

The `Error` script shows various error messages in the protocol.

```
# 2 repositories
# this test shows the errors

B2  __:___    register B2
B1  __:___    register B4
B2  __:___    register -- # user doesn't exist
B2  __:___    register B1

B1  B2:reg    exit
B2  B1:not    exit
```

Accept

The `Accept` script carries on evaluation, submission and acceptation of knowledge to the consensual repository. The comments display

```
# 3 repositories

B1  __:___    group 124   # become THE group repository

B2  __:___    register B1    #
B2  B1:tel    apply          # perform B1's content, given by the TELL

B3  __:___     register B1     #
# B1 -ask-all(register B3)--> B3
   B2  B1:ask     accept     # B1 asks B2 -> B2 accept B3's registeration
B1  B2:rep     print
B1  __:___     print      # just to print 2 blank lines

B2  B1:pno     eval 154      # after pnotify, evaluate A=1
B1  B2:eva     print

B2  B1:not     achieve 154
# B1 --(13/)ask-all(154)--> B2, B3
   B3  B1:ask     accept
   B2  B1:ask     accept
# B1 --pnotify(accept)-> B3, B2
# B1 --notify(accept)-> B2
# B1 --tell(154)-> B3, B2

B3  B1:tel    apply

B3  __:__    exit
B2  B1:tel   signal
B2  __:___   exit

B1  B2:sig   sleep
B1  __:__    exit
```

Forward_reg

The `Forward_reg` script tests the forward registration of a repository's repository to a third repository.

```
# 3 repositories

B1  __:___     group   124
B2  __:___     group   100

B3  __:___     register B2     #
B3  B2:tel     print B3 registers to B2

B3  __:___     forward register B1   # to have B2 registered to B1
# B2 --()ask-all(forward(register))--> B3
   B3  B2:ask     accept
# B2 --pnotify(accept)-> B3,
# B2 --notify(accept)-> B3

# B2 --register-> B1
#  B1 --notify(accept)-> B2
#  B1 --tell(124)-> B2
#     B2 --ask-all(achieve(124))-> B3
B3  B2:ask  accept
#     B2 --tell(124)-> B3
B3  B2:tel    signal
B3  __:___    exit

B2  B3:sig    exit
B1  B3:sig    exit
```

Mail

The `Mail` script tests the ability of the system to work through shutdown of repository and take advantage of the mailbox system.

```
# 2 repositories
# this test shows the communication between B2 and B1 by theirs MBox."

B1  __:___    group 124     #B1 becomes group rep., with 124 as knowledge

B2  __:___     register B1
B2  B1:tel     eval 3     # B1 is up, so the msg is sent to it.

B1  B2:eva    signal      # as soon as B1 receives EVAL from B2, it signals
B1  __:___    shutdown    # its disconnection

B2  B1:sig    sleep 2     # give time to B1 for deconnection,
B2  __:___    eval 4      # B1 is down -> B2's request is sent to B1's mbox
B2  __:___    signal      # and B2 signals
B2  __:___    shutdown    # its deconnection


B1  B2:sig   print B1 wakes up
B1  __:___   sleep 5
B1  __:___   wakeup # B1 reads it's MBOX,  answers to 'eval', but to B2's MBOX
B1  __:___   signal # wakeup done.

B2  __:___   print initiatives    # to see that 'eval' isn't answered yet
B2  __:___   print Wait for B1's reconnection
B2  B1:sig   wakeup

B2  __:___    exit
B1  __:___    exit
```

Appendix 4. Error messages

`CO4_ADMISSIBLE`
The tested knowledge is admissible for the repository.

`CO4_ALREADY_ANSWERED`
The requested initiative has already been answered, so it can not be answered another time.

`CO4_ALREADY_EXISTS`
The repository to create already exists and Co4 is not allowed to erase it.

`CO4_ALREADY_REGISTERED`
The current repository already has a group repository.

`CO4_BEING_REGISTERED`
The current repository is still waiting for a previously registration answer. So it cannot register again now.

`CO4_CANT_DENY`
The requested initiative does not correspond to an `ACHIEVE or a REGISTER`, so it cannot be denied.

`CO4_CONTAINED`
The tested knowledge is already present in the repository.

`CO4_CONTRADICTORY`
The tested knowledge is contrary to the repository,

`CO4_DOESNT_EXIST`
The requested repository does not exist.

`CO4_ERROR`
Another error occurred.

`CO4_IS_ACTIVE`
The requested repository is still active, it cannot be deleted nor waken up.

`CO4_NO_ANS_RESPONSE`
The ANS does not respond, its URL may be invalid.

`CO4_NO_CONNECTION`
The URL of the required repository does not respond.

`CO4_NO_FILE`
The required file does not exist, or is inaccessible.

`CO4_NO_SUCH_INITIATIVE`
There is no initiative corresponding to the given surrogate.

`CO4_NO_SUCH_PROPOSAL`
There is no proposal corresponding to the given surrogate

`CO4_NO_SUCH_TELL`
: There is no tell corresponding to the given surrogate.

`CO4_RECEIVER_UNKNOWN`
: The target repository for a KQML message does not exist for the ANS.

`CO4_SELF_REGISTERED`
: The current repository tried to register to itself.

`CO4_UNREGISTERED`
: The current repository must have a group repository to perform orders such as `ACHIEVE`, `DENY`, etc.

Appendix 5. Glossary

Agent name server (ANS)

An agent name server is a process which has in charge the identification (and management of addresses) of the various repositories. It is identified by an URL. In CO4, the ANS also manages the data of the repositories.

ANS

See agent name server.

API

Application Programming Interface. A set of primitives which allow to manipulate the implementation of the TROEPS model.

Call for comments (CFC)

A call for comments is used by the group repository to gather the answers to the proposal sent to each subscriber. The CFC will be accepted, rejected or challenged, depending of each individual answer.

CFC

See call for comments.

CGI (script)

CGI (Common Gateway Interface) is a technique for generating dynamic HTML pages in a HTTP server. A program called CGI script is launched by the HTTP server when particular URLs are requested and its output is a HTML page which is redirected on the server output.

CO4

CO4 is a system and protocol allowing several users to modify concurrently a ositoryrepository. It modifies the ositoryrepository only when all its subscribers have decided to do so.

Group repository

A group repository is a repository that is fully automated and that is meant to contain the consensual representation among its subscribers.

HTML

Hyper-text mark-up language. A format for encoding hypertext documents so that they are visible and accessible through a network. HTML is based on the SGML (standard generalised mark-up language) syntax.

HTTP

Hyper-text transmission protocol. The protocol used for dealing with HTML documents through TCP/IP networks.

Initiative

An initiative is a query issued by the current repository to its group repository.

IP

Stands for Internet protocol.

KAPI

KQML application programming interface.

KQML

Knowledge query and manipulation language.

Log

The log is the set of proposals that have been accepted in the group repository but have not been introduced into the current repository for various reasons.

Mailbox

A mailbox stores the set of messages received by a repository. It is stored by the ANS when the repository is shut down.

Proposal

A proposal is a query that has been sent by the group repository, to be accepted, rejected or challenged.

Repository

A repository is a container of the description of the artefact to be built. In the following, the repository is a running process.

Subscriber

A subscriber is a repository subscribing to a group repository.

Surrogate

A surrogate is a number identifying in a unique manner the message exchanged between repositories (the numbers are not unique, but the context — the kind of message and who issued it warrant the identification of the message).

Troeps

Has no particular meaning.

URL

A URL (Uniform Resource Locator) is the address by which the documents are identified in the World-wide web.

WWW

Acronym of World-wide web.

X-terminal

A “window“ of the X-window system.

Appendix 6. Function index

By using the on-line version of the manual, you should be able to retrieve the function definitions through the “Find” facility of your HTML browser.

AddKnowledge(char* contentString): int 22
Co4Accept(Co4Surrogate n): int 25
Co4Achieve(char* proposal): int 25
Co4Apply(Co4surrogate n): int 25
Co4Challenge(Co4Surrogate n, char* new_proposal): int 25
Co4CreateRepository(char* ANSurl, char* name, char* KqmlUrl, int erase):int 23
Co4DeleteAllFiles(): int 23
Co4DeleteFile(char* ANSfilename): int 23
Co4DeleteRepository(): int 24
Co4Deny(Co4Surrogate n): int 25
Co4Evaluate(char* proposal): int 25
Co4Forward(char* order, int depth): int 25
Co4GetFile(char* ANSfilename, char* localFilename): int 23
Co4GetGroupName(): char* 24
Co4GetMessage(char *msg): int 24
Co4IsGroupRepository(): int 24
Co4Log(Co4surrogate n): int 25
Co4ProcessMessage(char* msg): void 24
Co4Register(char* grpname): int 25
Co4Reject(Co4Surrogate n, char* comment): int 25
Co4SaveRepository(): int 24
Co4SendFile(char* ANSfilename, char* localFilename): int 23
Co4ServerInit(char* routingScriptUrl, int port, int (*hook)(char*), char* knowledgeHome): void 26
Co4ServerLoop(): void 27
Co4ServerStop(): void 27
Co4ShutDownRepository(): int 24
Co4Test(Co4surrogate n): int 26
Co4TurnIntoGroup(): int 24
Co4WakeUpRepository(char* ANSurl, char* name, char* KqmlUrl): int 24
ConfrontKnowledge(char* contentString): int 22
Debug: int 26
DumpKnowledge(): char* 22
gUrlPrefix: char* 27
InitKnowledge(): int 22
IntersectKnowledge(char* k1, char* k2): char* 22
LoadKnowledge(): int 22
MergeKnowledge(char* k1, char* k2): char* 22
oAcceptAll: int 26
oAcceptMine: int 26
oAnonymous: int 26
oKeepAll: int 26

oTellKnowledge: int 26
oTrace: int 26
SaveKnowledge(): int 22