Cm : A multitask communication package.

Version v8r1

Christian ARNAULT

Pierre MASSART

LAL - Orsay, France. arnault@lal.in2p3.fr

December 11, 2004

VIR-MAN-LAL-5600-107

1 Presentation.

The Cm package is an attempt to make simple and system independant the task-to-task communication problem. It covers the communication between tasks that operate on different operating systems, different architectures by hiding every detail of the use of TCPIP on which it is based.

These characteristics are quite important when one considers the domain of control-command in a real time environment, and in a more general meaning when one deals with distributed applications ans informations, the cooperative behaviour of applications permits to obtain a quite modular and structured architecture in the system design.

1.1 Document structure.

This document may be read at different levels, whether one merely wants to use Cm or if expertise is required in order to understand the detailed internal mechanisms.

So for a first approach the specification section, the section describing the CmMessage class and the one describing how to build a Cm application could be sufficient.

Then the CmConnect class description introduces some more internal mechanisms and permits to understand the detailed behaviour of the CmMessage objects.

The section on package installation and reconstruction is needed for the system manager who will install Cm , in particular for understanding how to configure the environment.

Lastly, a section is reserved for implementation details and other technical bits. Experienced programmers who want to work on additional layers on top on Cm (for instance) will require it.

Examples are provided at many places in the document. They all correspond to a real C file in the package distribution (the file name is mentionned with each example) and they may be compiled and linked to exercize Cm .

All possible remarks and suggestions either on the package (such as bugs, installation or behaviour problems) or on this document itself are welcome and will be addressed to the author :

Christian Arnault

email : arnault@lal.in2p3.fr

2 Specifications.

This section and all subsequent ones in this version of the document always take into account the most recent changes in the package.

`Cm` is meant to manage the task-to-task communications (sending or receiving messages) running on heterogeneous machines (with different architectures or operating systems) without limitations on the number of active connections (apart those induced by the operating systems).

The set of tasks (or applications) that may participate this network define a `Cm` *domain* managed by one special application - the `NameServer` - in charge of the physical addressing scheme, allowing several independant such domains to coexist.

An application with which a connection is requested is referenced by a **name**, that must be unique within one `Cm` domain and that doesn't need to mention anyhow the machine on which it runs, nor the transport characteristics (such as TCPIP parameters).

The central manager application `NameServer` is in charge of every mechanism for name registration, port number allocation and physical addressing operations transparently for the user applications.

Sending and receiving messages are managed asynchronously (without acknowledge management). This in particular implies that a special data framing protocol is added to the internal basic protocol (TCPIP) used for `Cm` .

The basic TCPIP protocol ensures the effective message transmission but not the arrival time nor the data packets organization (since successive message may be concatenated or split into pieces). It is therefore required to add a software layer on top of it in order to ensure the asynchronous data architecture. `Cm` provides this feature by the `CmMessage` package.

On the reception side, a *callback*-based mechanism is installed, so that user declared functions are triggered on message detection. This mechanism is combined with blocking (with transaction handling) or non-blocking capabilities in order to provide a rich integration scheme for interactive or real-time environments.

2.1 Implementation specifications.

`Cm` has been designed using a conceptual methodology using object oriented principles. The two main concepts manipulated by `Cm` are the *connection* that handles the path between two tasks and the *message* that manages structured information that can be worked on and transported along the connections. Implementation relies on the *classes* corresponding to these two concepts for organizing the functions presented to the users and data manipulated by `Cm` .

`Cm` exploits specifically the features and properties of the **TCPIP** protocol and for the implementation, the *socket* interface of the C language, and one of the main goals of `Cm` is to hide both of them to the user, showing only their major properties.

`Cm` is written with ANSI C and a set of principles based on object orientation. in order to achieve a good quality level for maintainance, portability on wide range of environments and easy evolutions (`Cm` has been ported to different operating systems such as DEC-ULTRIX, DEC-OSF1, HP-UX, LynxOS, OS9, SunOS, Solaris and DEC-VMS).

3 Cm architecture.

Cm is built around two main concepts the *connection* and the *message*. Two classes correspond to these concepts : the CmConnect class and the CmMessage class.

CmConnect manages the physical connections between applications by hiding the internal TCP-IP based mechanisms. It supports the packet-oriented data transfers and relies on the data-framing protocol provided by CmMessages in order to achieve fully asynchronous communications.

CmMessage gives the structuration to data needed for operating Cm in asynchronous mode. Many levels of security will guarantee internal integrity of message data as well as separation between messages themselves.

CmMessage objects generally hide the use of the CmConnect class and the user interface of CmMessage is sufficient to handle most of the operations.

The Cm package provides both a library meant to be linked to user applications and a set of predefined utility applications :

- the NameServer that manages TCPIP port allocation and associations between logical names and physical addresses. A private data base makes the configurations persistent.
- the cm tool that queries the NameServer about its internal data base, and can perform some interactive Cm activities.

3.1 The CmConnect class.

CmConnect handles the basic communication mechanisms. It is based on the use of the TCPIP protocol, and exploits the C socket interface. It manages (creates, opens, closes, destroys) the *sockets* and the primary level of unformatted data exchange.

The first role of CmConnect is to hide the C socket interface and the complexity of the TCPIP protocol, providing a simplified and secured environment.

Then it's able to handle the various management communications with the NameServer when connections are established. Therefore, a physical connection is closely related to the existence of one CmConnect instance, thus the construction method for such an object : CmConnectNew is used for initializing this connection by using the logical application name (received as its argument). Each CmConnect object is actually created only when the connection is possible (if the application is alive). Similarly, a connection loss (due to the application or the network) will yield a desactivation of the corresponding object and eventually its destruction.

Therefore properties of CmConnect objects may be summarized as follows :

- A connection is a peer-to-peer link between two applications (and is always associated with one TCPIP socket).
- Connection losses are automatically and transparently managed by Cm .
- Each application owns at least one connection : the one that will handle the connection requests from other applications. This particular CmConnect object can be accessed by the CmConnectWhoAmI function. The creation of this private connection is automatically taken into account by the CmMessageOpenServer function.
- Each application may manage one or several server connections (that may be addressed from other applications), each defined by one name which is the address by which a client will establish the connection.

3.1.1 Application initialisation.

Before any operation on `CmConnect` objects (and therefore on `CmMessage` objects) an application must declare one or several servers to the system by a *name* for each server. Each server may be declared either as a unique instance for this name with the `CmMessageOpenServer` function, or as a clonable server with the `CmMessageOpenMultipleServer` function. The actual name is a text string that must be unique in one `Cm` domain. Therefore, when used for a clonable server, the name is considered as a *generic* name and is suffixed by the `NameServer` by the sequence number of each particular instance.

The `NameServer` is requested to handle the name checking in both situations and to perform the port number allocation by the two open functions. One private connection is then created for each server if all required conditions are met.

An example of such a declaration that would like to be called **Toto** could be :

```
/*   File example1.c   */

#include <stdio.h>
#include <CmMessage.h>

main()
{
  if (!CmMessageOpenServer ("Toto"))
  {
    fprintf (stderr, "Declaration error.\n");
    return (0);
  }
}
```

Fig.1- Server name declaration.

```
/*   File example2.c   */

#include <stdio.h>
#include <CmConnect.h>

main()
{
  if (!CmMessageOpenServer ("Toto"))
  {
    fprintf (stderr, "Declaration error.\n");
    return (0);
  }
  if (!CmMessageOpenServer ("Titi"))
  {
    fprintf (stderr, "Declaration error.\n");
    return (0);
  }
}
```

Fig.2- Two-server name declaration.

3.1.2 Data conversion on machine architecture basis.

Each `CmConnect` maintains templates describing the byte ordering context characteristic of its originating machine. This context is visible through a converter object (instance of the `Cvt` class) which is able to convert to local representation any kind of received data. Each connection knows the converter required by the particular association of the two machines (possibly both the same) that are involved in a point-to-point communication.

At the connection time (thus only once in the connection life) this conversion information is exchanged between the two applications and from then on will be used for every message transfer (in both directions).

The current converter object of a given `CmConnect` may be accessed using the `CmConnectGetCvt` function, and conversions use the `CvtGetShort`, `CvtGetInt`, `CvtGetFloat`, `CvtGetDouble` functions. One should note that these translations are automatically performed by the `CmMessage` objects.

3.2 The `CmMessage` class.

This class exploits the connections in full asynchronous mode while keeping the `CmConnect` objects manipulations tranparent to the users. Although the direct access to the internally managed `CmConnect` objects is always possible it is never required to get such access for a normal user. Two specialized startup functions `CmMessageOpenServer` and `CmMessageOpenMultipleServer` must be used to initialize `Cm` . Once the appropriate startup is performed `CmConnect` objects are transparently managed by `CmMessage` objects both when sending a `CmMessage` (the `CmConnect` is created internally) and when receiving it (the `CmMessage` is created at connection request internally).

The CmMessage objects handle an extensible-array based mechanism to construct the message data, permitting to accumulate typed informations while the CmMessage object keeps the knowledge of the type structure provided by the user. Each CmMessage object built so is then transported across the connection, propagating its internal structure.

The internal data formatting takes care of the different machine architectures at each side of the connection by an automatic translation (using the Cvt converter object of the relevant CmConnect object).

Then CmMessage objects can be given a *type* (specified as a character string) that will be used at reception level to trigger dedicated activities or *handlers* (these handlers are declared by the user using the CmMessageInstallHandler function).

3.3 The internal protocol managed by CmMessage objects.

In order to maintain the global consistency of messages a framing format made of a prefix and a suffix is managed transparently for each CmMessage data. This protocol permits in particular to understand the global enveloppe of the CmMessage data, for detection of contiguous messages or for guaranteeing the completion of the data transfer for each CmMessage .

This mechanism makes the detection possible on message basis instead of on physical frame basis. Thus, a level of handlers is provided for CmMessages based on message types and specialized handlers are automatically triggered at each individual CmMessage reception.

3.4 CmMessage building and sending.

A CmMessage aimed at being sent is not associated to an existing connection (and to the CmConnect object). Instead, the CmMessage will be built (accumulating data in it) and then sent to some destination, or possibly several destinations (each of those having a dedicated CmConnect object transparently managed).

A CmMessage object must first be created using the CmMessageNew function that prepares it to act as an extensible structure for typed values such as numeric values, texts, arrays etc...

Several specialized functions are provided by Cm in order to construct the CmMessage structure dynamically :

- CmMessagePutChar
- CmMessagePutShort
- CmMessagePutInt
- CmMessagePutLong
- CmMessagePutFloat
- CmMessagePutDouble
- CmMessagePutText
- CmMessagePutBytes
- CmMessagePutArray
- CmMessagePutExtArray

The last two functions install arrays of values within the `CmMessage` structure either by copying it directly into the `CmMessage` data or by rather installing a description for it, avoiding the physical duplication of the array data. In this case, the array data is actually accessed only when the `CmMessage` is transfered. Arrays elements may be of any one of the simple types managed as single items and are converted the same way as simple items.

In both cases, the array is available at reception side the same way.

The conversion mechanism (using the `Cvt` converter object) takes care of byte ordering and word alignment according to the receiving machine. Control informations are installed along the `CmMessage` structure for guaranteeing the data integrity and to control the semantic of the data items while retrieving data from a received `CmMessage` .

Each `CmMessage` object may be given a type through the `CmMessageSetType` function. The type is provided as a free character string, and is meant to be used at the reception side to trigger dedicated handlers.

The function `CmMessageSend` is then used to send the `CmMessage` object to an application (specified by its name). This function takes care of the global message formatting, creates (or reuse) a `CmConnect` object and send the `CmMessage` object without any acknowledge manipulation.

The `CmMessageSend` operation completes in two phases : it first *posts* the message and then *waits* for its tranfer until it is finished. The `CmMessageWait` function is used internally for the second phase.

It is possible to control manually the two phases by using the `CmMessagePost` function which does not wait for the entire transmisson of the message. The use of this function must be done quite carefully since, for instance the `CmMessage` object is set in a special state (`Sending`) while packets are sent. While it is in this state, the `CmMessage` object cannot be used (`Cm` takes care of the internal protections) and for instance, trying to `Put` any item in a `CmMessage` while it is in this state has no effect on its internal state.

The `CmMessagePost` function receives as an argument a termination handler (which has the same syntax as the handlers declared for *receiving* messages) that will be called when the last packet of the `CmMessage` has been successfully sent, or if the connection is lost.

Each `CmMessage` object is controled internally by a finite set of possible states that are checked upon at each `CmMessage` operation. Actions are then either performed (yielding possibly a state transition) or ignored if the corresponding transition is forbidden. The following diagram shows the set of possible states and transitions permitted to a `CmMessage` object :

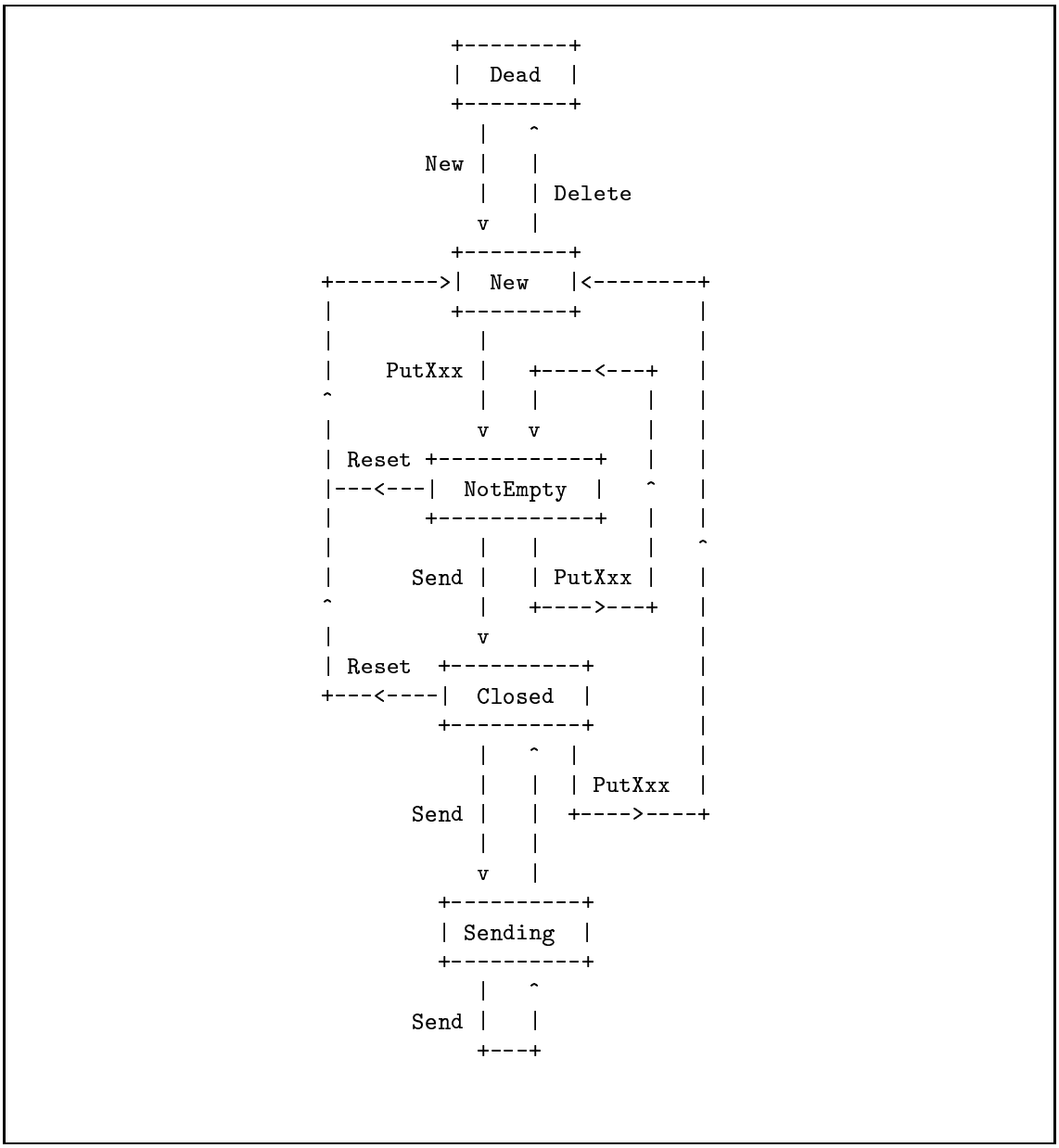

Fig.3- State diagram for the `CmMessage` objects.

Some explanation will help understanding this diagram :

- Actions correspond to methods of the `CmMessage` class which name is built by prefixing the action name by `CmMessage` such as `CmMessageSend` or `CmMessageDelete`.
- The `Send` operation can be repeatedly applied to a given `CmMessage` object, only the first of them will operate the `Close` operation.
- The first operation different than `Send` on a `Closed CmMessage` will perform a `Reset` action, which in particular will erase previous data in this `CmMessage` .
- The `Delete` action is actually available from any state and always correspond to the physical destruction of the object.

- The `Reset` initializes the data structure of the `CmMessage` and therefore loses all previously accumulated data.

The following example shows the steps of a `CmMessage` object building and how it is sent to applications with a type :

```
/* File example6.c */

#include <stdio.h>
#include <CmMessage.h>

void build_an_image (char** pixels, int* size)
{
  static char image[100];

  *pixels = image;
  *size = sizeof(image);
}

main()
{
  CmMessage message;
  int images = 3;
  int image;
  char* pixels;
  int imageSize;

  if (!CmMessageOpenServer ("Toto"))
  {
    fprintf (stderr, "Declaration error.\n");
    return (0);
  }

  message = CmMessageNew ();

  CmMessagePutText (message, "Hello you");
  CmMessagePutText (message, "I send you");
  CmMessagePutInt  (message, images);
  CmMessagePutText (message, "images : ");
  for (image = 0; image < images; image++)
  {
    /* something to build an image... */
    build_an_image (&pixels, &imageSize);
    CmMessagePutBytes (message, pixels, imageSize);
  }

  CmMessageSetType (message, "Image");

  CmMessageSend (message, "Client1");
  CmMessageSend (message, "Client2");
  CmMessageSend (message, "Client3");
}
```

Fig.4- Building and sending a `CmMessage` object.

3.5 Receiving a `CmMessage` .

`CmMessage` objects are received by type-dedicated handlers managed by the basic `Cm` engine.

The detection and assembly of individual message frames are performed internally and handlers are triggered when each complete message is available with an automatic detection of message types.

Handlers are installed using the `CmMessageInstallHandler` when they are associated with one particular type or using the `CmMessageInstallDefaultHandler` for handling unforeseen message type occurence.

The `CmMessage` data items are retrieved from it using the following functions :

- `CmMessageGetChar`
- `CmMessageGetShort`
- `CmMessageGetInt`
- `CmMessageGetLong`
- `CmMessageGetFloat`
- `CmMessageGetDouble`
- `CmMessageGetText`
- `CmMessageGetBytes`
- `CmMessageGetArray`

One should notice that retrieving arrays is done the same way whether they have been produced as embedded or external arrays. In both situations, data are received within the message frame.

The `CmMessageGetType` permits in addition to get the actual type of the received message (this feature is likely to be usefull in a *default handler*).

It is also possible to enquire the `CmMessage` object about the type of the next available item while retrieving the data, by using the `CmMessageGetItemType` function. The result of this function may be :

- `CmMessageItemTail`
- `CmMessageItemChar`
- `CmMessageItemShort`
- `CmMessageItemInt`
- `CmMessageItemLong`
- `CmMessageItemFloat`
- `CmMessageItemDouble`
- `CmMessageItemText`

- `CmMessageItemBytes`
- `CmMessageItemArray`

The first value, `CmMessageItemTail` means that the message's tail is met and no further data item is available. It is then useless (though harmless) to keep on *get*ting more data items.

On the other hand, trying to retrieve a data item with the wrong type yields an error message, and results in skipping the current item. Therefore, in case the type of each data item is not statically known in the handler's context, the use of `CmMessageGetItemType` is strongly encouraged.

An example of an application receiving messages from the one in the previous example. One uses here a dedicated handler that takes care selectively of the messages typed `"Image"` :

```
/* File example7.c */

#include <stdio.h>
#include <CmMessage.h>

CmMessageStatus my_image_handler (CmMessage message, char* sender,
                                  char* serverName);

void show_an_image (char* pixels, int size)
{
}

main()
{
  if (!CmMessageOpenServer ("Client1"))
  {
    fprintf (stderr, "Declaration error.\n");
    return (0);
  }

  CmMessageInstallHandler (my_image_handler, "Image");

  CmMessageWait ();
}

/* (to be continued) ... */
```

```
/* ... File example7.c (continued) */

CmMessageStatus my_image_handler (CmMessage message, char* sender,
                                  char* serverName)
{
  char* text;
  int images;
  int image;
  char* pixels;
  int imageSize;

  text = CmMessageGetText  (message);
  text = CmMessageGetText  (message);
  images = CmMessageGetInt (message);
  text = CmMessageGetText  (message);

  for (image = 0; image < images; image++)
  {
    pixels = CmMessageGetBytes (message, &imageSize);
    /* something to use this image... */
    show_an_image (pixels, imageSize);
  }

  return (CmMessageok);
}
```

Fig.5- Receiving `CmMessage` objects with a dedicated handler.

```
/* File example8.c */

#include <stdio.h>
#include <CmMessage.h>

CmMessageStatus my_image_handler (CmMessage message, char* sender,
                                  char* serverName);

void show_an_image (char* pixels, int size)
{
}

main()
{
  if (!CmMessageOpenServer ("Client1"))
  {
    fprintf (stderr, "Declaration error.\n");
    return (0);
  }

  CmMessageInstallHandler (my_image_handler, "Image");

  for (;;)
  {
    CmMessageCheck ();

    /*
       Here come actions to be executed repeatedly
       with no explicit wait for messages.
        The CmMessageCheck will only detects them and
       activate the appropriate handlers.
       ...
    */
  }
}

/* (to be continued) ... */
```

```
/* ... File example8.c (continued) */

CmMessageStatus my_image_handler (CmMessage message, char* sender,
                                  char* serverName)
{
  char* text;
  int images;
  int image;
  char* pixels;
  int imageSize;

  text = CmMessageGetText  (message);
  text = CmMessageGetText  (message);
  images = CmMessageGetInt (message);
  text = CmMessageGetText  (message);

  for (image = 0; image < images; image++)
  {
    pixels = CmMessageGetBytes (message, &imageSize);
    /* something to use this image... */
    show_an_image (pixels, imageSize);
  }

  return (CmMessageOk);
}
```

Fig.6- Non blocking detection of `CmMessage` objects.

4 Transaction management in Cm .

Cm is mainly designed to be operated asynchronously, and in an event driven non blocking mode. This in particular means that protocols installed between applications should not be based on a blocking wait for a dedicated answer.

However logic of the applications often require that a specific answer is expected after having send a request. Cm provides an optional transaction based mechanism to implement such a logic.

4.1 Description

A typical scenario showing how to use the Cm transaction could be :

1. A transaction is first opened, creating a unique identifier for it, using the `CmOpenTransaction` function.
2. This identifier is installed (by the user) in the message containing the request (thus the protocol is increased to include the transaction id).
3. The request is sent and the requestor enters a conventional wait loop (typically using the `CmMessageWait` function)
4. The application receiving the request is expected to answer is a finite amount of time. When the answer is sent back, it now includes the transaction id. Notice that the complete answer may include several messages, only the very last one has to include the returned transaction id.
5. The request sender eventually receives the answer, and among it the message containing the transaction id. It then terminates the transaction using the `CmTerminateTransaction` function (from within the massage handler). (Notice that in this case the handler will return `CmMessageOk` and that the traditional `CmMessageBreak` should not be used any longer).
6. The main wait loop is then stopped and the requestor can now close the transaction (using the `CmCloseTransaction` function). This last operation releases the allocated transaction id.

4.2 An example

An example of how to implement this scenario includes three code sections :

1. The request sending, the wait for the anser and the transaction cleanup,

```
int id;
CmMessage request;

id = CmOpenTransaction ("my request", NULL);

... ( now building the request )

CmMessagePutInt (request, id);
CmMessageSend (request, "server");

while (!CmIsTransactionTerminated (id))
  {
    CmMessageWait ();
  }

CmCloseTransaction (id);
```

Fig.7- Sending a request with a transaction id

2. The handler in the receiver application used to manage the request, compose and send the answer,

```
CmMessageStatus request_handler (CmMessage message, ...)
{
  int id;
  CmMessage answer;

  ... ( getting the request contents )

  id = CmMessageGetInt (message);

  ... ( working with the request )
  ... ( and building the answer )

  CmMessagePutInt (answer, id);
  CmMessageSend (answer, sender);
  ...

  return (CmMessageOk);
}
```

Fig.8- Building up an answer to a request with a transaction id

3. The handler in the requestor used to receive the request.

```
CmMessageStatus answer_handler (CmMessage message, ...)
{
  int id;

  ... ( getting the answer contents )

  id = CmMessageGetInt (message);

  CmTerminateTransaction (id);
  ...
  return (CmMessageOk);
}
```

Fig.9- .

One should notice the differences between this mechanism and the traditional use of a `return (CmMessageBreak)` used in the receiving handler:

- The break mechanism was global to the application, and thus it was impossible to distinguish between two break events produced from interlaced handlers (i.e. occurring recursively to each other).
- Similarly it was not possible to safely interlace protocols which used the same request-anwer protocol, since there was a risk for a confusion between two of them.
- Therefore the transaction-based protocol permits to interlace any protocol even recursively.
- One may also benefit from the possibilty to associate a user object with each transaction in order to safely pass its reference between the calling sequence and the handler, providing an unambiguous one-to-one relationship between the answer and the request.

4.3 Remote debugging facilities for transactions

`Cm` provides internal debugging facilities for transactions. Through sending a dedicated `Cm` messages, it is possible to

- Dump all existing transactions of an application,

```
> cm send -to=my_app -type=CmMessageDebug text=Transactions
```

Fig.10- Dump active transactions in an application

- Force an existing transaction to terminate, providing a kind of emulation of a failing server (typically to abort an infinite wait for answer)

```
> cm send -to=my_app -type=CmMessageDebug text=TerminateTransaction \
    int=<the transaction id>
```

Fig.11- Remote termination of a transaction

Fig.12- .

4.4 Programming interface

The following entry points are provided so as to manipulate transactions :

4.4.1 CmOpenTransaction

`Syntax` `int CmOpenTransaction (const char* info, void* user_object)`

Description This function opens a new transaction, allocating a free identifier for it. A free informational character string can be provided as well as a reference to a user object. This reference can be retrieved at any time using the CmGetTransactionObject function.

4.4.2 CmTerminateTransaction

`Syntax` `void CmTerminateTransaction (int id)`

Description This function terminates an opened transaction. The main effect of terminating a transaction is to stop the current waiting loop.

4.4.3 CmCloseTransaction

`Syntax` `void CmCloseTransaction (int id)`

Description This function definitively closes an existing transaction. The identifier will be released (and can be reused for another transaction).

4.4.4 CmIsTransactionTerminated

`Syntax` `int CmIsTransactionTerminated (int id)`

Description This function tests whether the referenced transaction is terminated.

4.4.5 CmGetTransactionObject

Syntax `void* CmGetTransactionObject (int id)`

Description This function retrieves the reference to an object associated with the specified transaction.

4.4.6 CmGetTransactionInfo

Syntax `char* CmGetTransactionInfo (int id)`

Description This function retrieves the informational character string associated with the specified transaction.

4.4.7 CmCloseTransaction

Syntax `CmTransactionStatus CmCloseTransaction (int id)`

Description Definitively close a transaction, releasing its identifier. No other operation are permitted on this transaction.

Possible values for the returned `CmTransactionStatus` are :

```
CmTransactionOk
CmTransactionNotFound
CmTransactionAlreadyClosed
```

4.4.8 CmTerminateTransaction

Syntax `CmTransactionStatus CmTerminateTransaction (int id)`

Description Terminate an opened transaction. This operation generally results in stopping the current wait loop, marking the transaction as *terminated*. The transaction may then be either *restarted* or *stopped*.

4.4.9 CmRestartTransaction

Syntax `CmTransactionStatus CmRestartTransaction (int id)`

Description Restart a transaction for a next occurence of the event associated with it.

4.4.10 CmGetTransactionState

Syntax `CmTransactionState CmGetTransactionState (int id)`

Description Returns the current state of the referenced transaction. Possible values are :

```
CmTransactionClosed
CmTransactionPending
CmTransactionTerminated
```

5 Handling errors in `Cm` .

Two mechanisms permit a user to detect misfunctions while using `Cm` . Firstly, most (if not all!) `Cm` entry points return a value which can be either a requested object (such as `CmMessageNew`) or a status value (such as `CmMessageSend` or `CmMessageWait`). Whenever a failure in doing the required operation occurs, the return value would be `NULL` (for returned objects) or `0` (*zero*) (for status values). The user should therefore test upon these returned values in order to avoid continuing an illegal `Cm` sequence. However, internal states are maintained and checked upon for every `Cm` object in order to keep some safety level.

Secondly, internally detected errors that would not be directly translated into returned values often produce *error messages* that are sent by default onto `stderr`. A user defined printer operator, syntacticly similar to the standard `vprintf` function (use `man vprintf` in order to get the exact and complete syntax for it) may be declared to `Cm` with the `CmMessageInstallPrinter` function. Typical usages for such printer operators are for putting error messages into log-files or displaying them into graphical windows.

6 The NameServer .

The NameServer is a special application (built with Cm) aimed at managing associations between logical names chosen by the Cm applications and their physical addresses (Internet host addresses and port numbers).

Each such address must be unique on the network, therefore, one of roles of the NameServer is to allocate one dedicated port number per application running on a given machine.

The mechanism used to ensure the correct allocation consists in the selection of a value within a specified range for the following entities :

- The host name where the NameServer runs.
- The port number allocated to the NameServer .
- The first port number that the NameServer may allocate for Cm applications within its domain.
- The number of port numbers that the NameServer may allocate within the domain.

One sees that the set of this four values is enough to specify a complete Cm domain or environment able to manage a coherent set of Cm servers, and opaque to any other Cm domain as far as there is no overlap on the specified address ranges. Within each such domain the server names should be unique, while a given name may exist in two different Cm domains.

The domains that correspond to an individual set of configuration parameters are described within one dedicated text file named $CMROOT/mgr/CmDomains. However, no mechanism or tool is provided yet to ensure that the definitions are consistent and yield no overlap. The greatest care is thus required by system managers while defining the address ranges in this database. Nevertheless, the NameServer applications that manage each individual domain are automatically configured from this database, giving some security level.

The format for this file is described as follows :

- Each domain is specified on one line in the file.
- The attributes of the domain are specified each on one word and separated by space characters.
- The format for the specification of one domain is :
  - The domain name.
  - The host where to run the NameServer .
  - The port assigned to the NameServer .
  - The first port number of the range used by the NameServer to allocate user's ports.
  - The size of this range.
  - The root location of the database repository. This directory specification is understood as a base for installing the database. However a subdirectory named with the domain name is added in order to manage several domains in the same environment.

An application (corresponding to one executable object) may declare several servers. The behaviour is strictly equivalent then as if the servers would be handled each by one application. The port numbers are allocated individually as well. Clients for both servers should never notice that they are physically installed in the same space.

6.1 The query operations for the NameServer .

A set of requests may be sent to the NameServer to get informations on the connections it is managing or to ask it to perform some operations.

Each of those requests has to be sent under the form of a CmMessage sent to the NameServer , with dedicated types for each kind of request. When needed request parameters are installed as the CmMessage contents. An answer is always returned to the caller, in the form of CmMessages with corresponding types, so that an application will declare dedicated *handlers* being in charge of understanding the result of the requests.

On another hand, the prebuilt application cm (shipped in the distribution kit) exploits this functionality and allows to interactively interrogate the NameServer using a shell command as follows :

```
Unix> cm <request>
```

The various requests understood by the NameServer can be summarized in the following table. Each request is described with the message type used and the data items provided. Answers are described with the message type and the data items returned.

Request	Parameters	Answer
NSGetAddress	Text <name>	NSAddress Text <original-name> Int <port> Text<new-name>
NSGetPort		NSPort Text <name> Int <port>
NSGetNames	Text <reg-expr>	NSNames Text <name> Text <name> ...
NSStop		NSStopped
NSRestart		NSStopped
NSGetConnects		NSConnects Text <name> Text <host> Int <port> ...
NSGetPorts		NSPorts <port> <port> ...
NSSetPeriod	Int <period>	NSPeriod Int <new value>
NSGetPeriod		NSPeriod Int <new value>

- The *NSStop* request will stop the NameServer itself (for exploitation purposes). The status value returned to the shell is such that the script NameServer.start used to drive the NameServer will itself terminate. The *NSRestart* request has a similar meaning except for the status value returned to the shell that makes the driver script to restart the NameServer instead.

- The *NSGetAddress* request returns a null port number (value `0`) when the specified name does not correspond to any active application. It also returns other informations that are (mainly internally) required to initialize the connections, such as the conversion format (for byte swapping numeric values). The set of informations returned by this request is used internally by `Cm` and is not meant to be used by a normal `Cm` user.
- The *NSGetNames* request looks for a set of names according to a pattern (a regular expression such as used in `egrep` or `sed`) and returns the list of active application names that match this pattern. The following example shows how to use this request :

```
      /* File example9.c */

      #include <stdio.h>
      #include <CmMessage.h>

      CmMessageStatus handler (CmMessage message);

      main ()
      {
        char* answer;
        char* destination;

        if (!CmMessageOpenServer ("Request"))
        {
          CmConnectExit ();
          return (1);
        }

CmMessageInstallHandler ((CmMessageHandler) handler, "NSNames");

message = CmMessageNew ();
CmMessageType (message, "NSGetNames");
CmMessagePutText ("Camera[0-9]*");

        destination = CmConnectGetName (CmConnectGetNameServer ());

CmMessageSend (message, destination);

CmMessageWait ();
      }

      CmMessageStatus handler (CmMessage message)
      {
char* name;

printf ("answer = ");
while (CmMessageGetItemType (message) == CmMessageItemText)
  {
    name = CmMessageGetText (message);
    printf ("%s ", name);
  }
printf ("\n");

        return (CmMessageOk);
      }
```

running this application will produce the following output :

```
answer = Camera12 Camera23
```

if the applications `Camera12` and `Camera23` were active when the request has been received by the `NameServer` .

- The *NSGetConnects* request returns a description of the complete set of connections currently served by the `NameServer` , including their host address and port number.
- The *NSGetPorts* request returns the complete set of allocated port numbers.
- The *NSTestPort* request checks the status of the given port number relatively to `Cm` : this port number may be :
  - outside the allowed range of port numbers for this particular `Cm` environment,
  - already allocated to one of the `Cm` applications,
  - available to a future allocation,
  - not available for allocation by a `Cm` application (likely it is used by a non-`Cm` application).
- The *NSSetPeriod* request changes the cleanup period used by the `NameServer` to check periodically the status of the `Cm` applications. The default value is 100 seconds.

The `NSGetAddress` and `NSGetPort` are meant for internal use only. A dedicated handler is declared within `Cm` and thus should never be overriden whithout unexpected results.

6.2 Data persistency in the `NameServer` .

Each `NameServer` activation maintains within dedicated files the whole set of informations on connections it manages. This permits in particular to *restart* it after it stopped (either on explicit user's request or due to a crash!!) while rebuilding the complete knowledge of active connections.

The connections between applications are not killed by a `NameServer` interruption, only new connections are impossible (and of course requests to `NameServer`). Eventually, when the `NameServer` is restarted, reconnection by the applications is performed transparently.

Thie database is maintained in a directory named with the *domain* name and installed in a root directory specified in the domain configuration file.

Each connection is stored in a specific file named with connection's name.

6.3 Activating the `NameServer` .

The `NameServer` application must be activated with the entire set of values that defines one `Cm` domain. Domains available to a site where `Cm` is installed are all specified in a single textual file :

```
$CMROOT/mgr/CmDomains
```

which contains one individual line per domain with the following format :

- The domain name.
- The host where to run the `NameServer` .
- The port assigned to the `NameServer` .
- The first port number of the range used by the `NameServer` to allocate user's ports.
- The size of this range.
- The root location of the database repository. This directory specification is understood as a base for installing the database. However a subdirectory named with the domain name is added in order to manage several domains in the same environment.

  The connection files are stored into the database directory and retrieved in subsequent reactivations of the `NameServer` .

The `Cm` distribution kit provides a tool (a Unix shell script located in `$CMROOT/mgr/NameServer.start`) that, in addition to performing some checks about the effectiveness of the current domain definitions (although these checks are also performed within the `NameServer` itself) would handle an automatic restart mechanism of the `NameServer` when it fails for strange reasons, guaranteeing a permanent life. This script is able to understand the various conditions that may occur for a termination of the `NameServer` , allowing to actually stop it when this is required by the manager, or when a non-recoverable error is detected (such as a bad environment definition or a failure to bind the TCPIP port).

It is important to understand that manually launching the `NameServer` *executable (i.e. without using the dedicated script) is generally a non-safe operation since this may not be done within the proper directory, or on the appropriate machine. The result of doing this is therefore unspecified.*

An example of a typical Unix session where the `NameServer` is first activated then stopped (by a request) and lastly restarted is shown as follows :

```
>##   One first selects a domain from the set specified
>##   in the $CMROOT/mgr/CmDomains file.

> more $CMROOT/mgr/CmDomains
LAL     as2.lal.in2p3.fr 22000 22001 999 $CMROOT/infos
LALDev  as2.lal.in2p3.fr 30000 30001 999 /tmp/Cm
> setenv CMDOMAIN LALDev
>
>##   Actual NameServer activation :
>
> $CMROOT/mgr/NameServer.start
>
>##   Then some applications are activated, using this
>##   Cm domain for a while.
>##   ...
>##   Then one decides to stop the NameServer.
>
> cm stop
>
>##   When the NameServer is reactivated, and as long as
>##  the same domain is still selected, the database will be
>##  restored, and still alive process will be re-connected to
>##  the NameServer.
>
>##  Reactivating the NameServer is just a matter of
>##  using the same shell script :
>
> $CMROOT/mgr/NameServer.start
>
```

Fig.13- Controling the `NameServer` by the special `NameServer.start` script.

The special script could be installed in the system's frame for automatic daemon activation tools (`rc.local` or `cron`) but it also provides an automatic reactivation mechanism of the `NameServer` in case of unexpected crash.

7 cm send : The interactive Cm exerciser.

The `cm send` utility available in the `Cm` package provides means of building and sending any kind of message to any application.

The message description is built using arguments given to the `cm send` command as follows :

- the destination

  `-to <name>`

- the message type

  `-type <type>`

- a possibility to receive one or several answers of given types

  `-handler <type>...`

- the message contents :

```
-char <character value>,...
-short <short value>,...
-int <int value>,...
-float <float value>,...
-double <double value>,...
-text <text value>,...
```

When a single value is given such as in :

`-int 10`

then a scalar value is installed in the message (using `CmMessagePutInt` here) and when a list of values is specified, such as in :

`-int 10,11,12,16`

then an array of values is installed in the message (using `CmMessagePutArray` here).

The following example shows how to send a request for comment to an application and receive the corresponding answer :

```
> cm send -to DbServerv4r3 -type CmRequestComment -handler CmComment
Message type CmComment received from DbServerv4r3
-text ServerAlive
```

Fig.14- Using the `Cm` exerciser.

8 Linking Cm to XWindows.

Linking Cm to XWindows, Motif or other such style of interactive environment requires to manage the *event*-loop coherently in the two worlds (since both are event-driven). Special modules are provided by the Cm distribution kit to handle this question properly either in a context of pure Xt/Motif manipulation or of the OnX package.

These modules provide each a setup function *CmXtSetup* or *CmOnxSetup* that will integrate the handler activation scheme of Cm and the *callbacks* mechanism through a common use of the general wait function CmMessageWait.

The integration of the two environments is shown in the following example :

```
/* File example10.c */

#include <stdio.h>
#include <Ci.h>
#include <OKit.h>
#include <OWidget.h>

#include <CmMessage.h>

Widget XtWidgetTop ();
int OXtDispatchEvent (XEvent*);
void CmSwitch ();

/* A Cm handler */
CmMessageStatus MyHandler (CmMessage message,
                           char* name,
                           char* serverName);

/* An OnX callback */
void SendMessage (char* to, char* text);


/* (to be continued) ... */
```

```
int main (int argc, char** argv)
{
  if (!CmMessageOpenServer ("Toto")) return (0);

  /* Declares the Cm handler */
  CmMessageInstallDefaultHandler (MyHandler);

  /* Install Cm in the C interpretor */
  CiPathNew ("CMSRC");
  CiBindClass ("CmSwitch",     (CiRoutine) CmSwitch);

  /* Declares the OnX callback */
  CiDo ("void SendMessage (char* to, char* text);");
  CiFunctionBind ("SendMessage", (CiRoutine) SendMessage);

  /* OnX is initialized */
  OKitInitClass (argc, argv);

  /* Install the Onx dispatcher inside Cm */
  {
    Display* display;
    Widget w;

    w = XtWidgetTop ();
    display = XtDisplay (w);
    CmOnxSetup (display, OXtDispatchEvent);
  }

  /*
    The CmMessageWait function is substituted to
    the usual OKitMainLoop function.
  */

  CmMessageWait ();

  OKitClearClass   ();
}

/* (to be continued) ... */
```

```
/* ... File example10.c (continued) */

CmMessageStatus MyHandler (CmMessage message,
                           char* name,
                           char* serverName)
{
  char* text;

  text = CmMessageGetText (message);
  printf ("Received from %s : %s.\n", Name, text);

  return (CmMessageOk);
}

void SendMessage (char* to, char* text)
{
  static CmMessage message = 0;

  if (!message) message = CmMessageNew ();

  CmMessageReset (message);
  CmMessagePutText (message, text);
  if (!CmMessageSend (message, to))
  {
    printf ("Error sending message to %s\n", to);
  }
}
```

Fig.15- Integration of `Cm` and OnX.

9 The Cm package : use and installation.

The Cm package is distributed with the following components :

- The tools required to develop a Cm application :
  - the libCm.a library.
  - the include file defining the public types corresponding to the public classes of the package, and the related functions prototypes :
    * CmMessage.h
- The NameServer application needed for the connection management.
- The cm application for interactively interrogate the NameServer .
- The NameServer.start shell script (only provided for Unix environment) for controling the NameServer 's activation.
- A shell script containing the definition of the various environment variables required both to access the package itself and to specify the Cm domains : setup.csh
- A template for the CmDomains configuration file.
- The documentation manuals :
  - This document (in Postscript format) : VIR-MAN-LAL-5600-107.ps
  - The architecture description document : VIR-MAN-LAL-5600-106.pdf

The general management of Cm (both for its building up and its usage) is handled through the CMT environment. This in particular controls the way environment variables and make macros are generated.

9.1 Configuring Cm .

The system configuration for Cm is two-sided : on the one hand, some environment variables are used to access the package (sources, binaries or management tools) and on the other hand Cm domains are specified using a special environment variable. Let's first look at the configuration variables :

`$CMROOT`	The root directory where `Cm` is installed. For instance, at LAL, the version `v8r1`is installed in : • `Unix> /lal/Cm/v8r1`
`$CMROOT/src`	The directory where sources and header files are stored.
`$CMROOT/cmt`	The directory where administration and reconstruction files (*Makefile*) are stored.
`$CMROOT/mgr`	The directory where scripts (start up scripts for instance) are stored.
`$CMROOT/$CMCONFIG`	The directory where package binaries are stored : the various libraries and executables (`NameServer` and `cm.exe`). Usual values are : • `$CMROOT/alpha` • `$CMROOT/HP-UX` • `$CMROOT/SunOs` • `$CMROOT/LynxOS` • `$CMROOT/Linux`
`$CMROOT/doc`	The directory where manuals are stored (LaTeXsource, postscript or HTML documents).

Then the current `Cm` domain is specified using the `CMDOMAIN` environment variable. The set of possible domains is described in the file `CmDomains` (installed in the management directory of `Cm`) and contains one line per domain with each having the format (items are separated by spaces) :

- The domain name.
- The host where to run the `NameServer` .
- The port assigned to the `NameServer`
- The first port number of the range used by the `NameServer` to allocate user's ports.
- The size of this range.
- The root location of the database repository. This directory specification is understood as a base for installing the database. However a subdirectory named with the domain name is added in order to manage several domains in the same environment.

9.2 Building a `Cm` application and using it.

The `Cm` application code should first get access to both the `Cm` functions definitions and to the `Cm` types. This is done in C language by including the `Cm` header file :

```
#include <CmMessage.h>
```

Fig.16- Using the `Cm` header file in an application.

The first operation installed in the application's code is to initialize one `Cm` personnal connection, giving the selected server's name. The important decision that must be taken at this level is whether it will be clonable or not. Making a server clonable means that its name cannot be hard-coded in any of it's client, since its name will be suffixed by a sequence number by the `NameServer` . Only non clonable servers can be referred to explicitely by clients. Clonable ones may only be answered on requests.

The functions :

- `CmMessageOpenServer`
- `CmMessageOpenMultipleServer`

permit one of the two possible initialisation modes of `Cm` servers. They all receive the server's name as their argument, and can be tested upon normal completion.

During the application's life, several servers may be activated or disactivated. Disactivation of a server is done using the function :

- `CmMessageCloseServer`

An old style of server activation was used in the previous versions of `Cm` (up to version v5rx). The corresponding entry points have been marked as 'obsolete'. The developpers are thus strongly encouraged to switch to the new style. The entry points concerned with this remark are :

- `CmConnectStartupAsServer`
- `CmConnectStartupAsMultipleServer`
- `CmConnectOpenServer`
- `CmConnectOpenMultipleServer`
- `CmMessageStartup`
- `CmMessageStartupMultiple`

9.2.1 Compiling and linking a `Cm` application.

Developping an application using the `methods` environment automatically provides all required connections and references to include search paths or library usage related with `Cm` .

This will be obtained if one specifies in the `requirements` file of the developped package :

```
use Cm v8r1

...

application MyApp MyApp.c
```

Fig.17- The `use` statement for `Cm` .

Then using the usual `pck genmake` utility will provide all required make macro definitions so as to properly compile and link your application.

9.2.2 Running a `Cm` application.

The proper `Cm` context has to be correctly setup prior any application can be started. This consists in the definition of some environment variables (usually by executing the shell script `$CMROOT/mgr/setup.csh`) and launching the `NameServer` application (only if thhis is required and by executing the shell script `$CMROOT/mgr/NameServer.start`).

One way to ensure the proper installation is done one could be to execute the following sequence of commands :

```
  Unix> source .../Cm/../cmt/setup.csh

Check the NameServer :
  Unix> cm names
  CmNameServer
  cm_1

  (Minimal answer when NameServer is already active)

or :
  Unix> cm names
  Cm> Impossible to reach the NameServer.

  (Answer when NameServer is unavailable)

Start the NameServer (when needed) :
  Unix> $CMROOT/mgr/NameServer.start

Run the application :
  Unix> toto.exe
```

Fig.18- Setting and checking the `Cm` context.

9.3 Porting the `Cm` package on a new system.

In order to be properly rebuilt and operated on a given system, one should ensure the availability of an ANSI C compiler with the socket interface environment supporting TCPIP.

The `Cm` package is distributed as a compressed Unix `tar` file comprising the sources, the manual and the various administration or reconstruction files.

Once untarred, one should have a look at the file `./Cm/v8r1/mgr/requirements`. It contains definitions for various environment variables, that may need to be adjusted. Once this is done, the following sequence of action should be performed :

```
Unix> cd <some root>
Unix> ftp lalftp.in2p3.fr
anonymous
> cd pub/Cm
> get CMTv3r1.tar.Z     ! if needed
> get CSetv2r5.tar.Z        ! if needed
> get Cmv7r7.tar.Z
> quit
Unix> mkdir Cm
Unix> cd Cm
Unix> uncompress ../Cmv7r7.tar.Z
Unix> tar xvf Cmv7r7.tar
Unix> cd v7r7/mgr
Unix> vi requirements
...
Unix> vi CmDomains
...
Unix> pck config
Unix> source setup.csh
Unix> [g]make
Unix> setenv CMDOMAIN ...
Unix> $CMROOT/mgr/NameServer.start &
Unix> cm names
...
```

9.4 How to figure out the actual situation of a `Cm` context.

Often, one has to understand on a given machine or environment whether `Cm` is configured, whether the `NameServer` is running or not, and which domains are available or used.

The following points can be easily checked upon in order to figure out what is the current setup of `Cm` :

- Check the installation directory for the `Cm` package.

  The environment variables `CMROOT` and `CMVERSION` must be defined. When this is not true you should run the setup script (you need to explicitely know the exact location where `Cm` is installed). At LAL you should do :

  ```
  csh> source /lal/Cm/v8r1/setup.csh
  ```

- Select one of the possible domains to live with.

  The set of domains is described in the basic configuration file located in `$CMROOT/mgr/CmDomains`. the `cat` Unix command) one sees the list of available domains.

  Selecting one of the domain is done by setting the environment variable `CMDOMAIN` the selected domain name.

- Check the `NameServer` .

  Each domain is assigned one host address where the `NameServer` should run and a directory where the `NameServer` database is maintained. Only one `NameServer` may exist in the context of a particular domain. Therefore, before trying to run a new `NameServer` , one should check its existence. On the machine assigned to it, the Unix `ps` command can be used as follows :

```
home> ps -e | grep NameServer
 8812 ttyp3    S  +     0:00.01 grep NameServer
 8762 ttyp4    S        0:00.10 csh -f NameServer.start
 8808 ttyp4    S        0:00.05 NameServer.exe LAL v7r7
```

  This shows that one `NameServer` is currently running in the context of the domain named `LAL` and under the version `v8r1`.

- If the `NameServer` is *not* running for the selected context, one can try to launch it using the `$CMROOT/mgr/NameServer.start` shell script. However, this operation may require some access privileges in order tolet the `NameServer` write out its database.

  On the other hand, running this script must be done on the machine assigned to the selected domain.

- Check the state of a runnning `NameServer` .

  The typical application that can be run in order to verify the correct behaviour of the `NameServer` is `cm`.

  Typical use of it are :

```
home> cm <command> [args...]
command :

names                     : Gets the names of active Cm applications
connects                  : Gets the descriptions of active connections
comments                  : Gets the comments of active servers
ports                     : Get the list of allocated ports
period [<new period>]     : Get or set the cleanup period of the
                            NameServer
reconnect <name> <port> <host> <owner>   : Try to reconnect NameServer
                                           to a lost application
restart_server            : Restart the NameServer
stop_server               : Stop the NameServer
domain                    : Show domain info
domains                   : Show all domains
version                   : Show Cm version
trace <level> <name>      : Trace messages in application <name>
                             0     : off
                             1/2/3 : on
debug [on/off] <name>     : Set/reset debug mode in application <name>
check_base                : Perform various checks on the database
send ....                 : Build and send a message to an application
receive ....              : Receive messages
cleanup <name>            : cleanup <name> in the Cm connects list
                            (process does not exist anymore)
print <name>              : print out all connections currently opened
                            with <name>
kill <connect>            : close <connect> Cm connect opened with the
                            NS
kill <connect> <name>     : close <connect> Cm connect opened with
                            <name>
```

```
home> cm names
CmNameServer
cm_1
```

10 Application programming interface for Cm .

This section presents the functions available on the CmConnect and CmMessage classes.

10.1 The CmMessage functions.

This class handles a structured protocol within data frames that have to be sent over the network through CmConnect objects. The mechanism they provide permits to operate the connection asynchronously since the data itself knows about their internal structure. In addition to the control operations, a general typing mechanism is added in order to install specialized behaviour on specific message receptions.

In most of the CmMessage 's methods, the first argument is a reference to the CmMessage object acted upon in this method. For all these situations, the corresponding argument is not documented explicitely.

10.1.1 CmMessageOpenServer - CmMessageOpenMultipleServer

Syntax
```
int CmMessageOpenServer (char* name)
int CmMessageOpenMultipleServer (char* genericName)
```

Description These functions declare a server as a participant to a Cm domain defined by its context and selects a-priori an asynchronous operating mode for incoming connections. One of these two functions should be selected on the basis of whether the server should be unique in the Cm domain or clonable.

This is the very first operation to do within a Cm application before any attempt to communicate with another server.

For clonable servers, each clone is given a sequence number corresponding to the instance number for this server. Clone numbers are reused when clones disappear.

Parameters

name	The name under which the server will be known by the system for unique applications.
genericName	The generic name used to form the actual server's name for clonable ones. Names of clones are built by suffixing the generic name by an underscore followed by the sequence number of the clone.

Returned value An integer value (1 or 0) describing the correctness of the initialisation procedure. A null value means the required name is not available or the NameServer is not reachable.

10.1.2 CmMessageNew

Syntax
```
CmMessage CmMessageNew (void)
```

Description Creates a `CmMessage` object. It is meant to be filled up with typed items using the `CmMessagePutXxx` functions. Once filled up it can be sent to a server using the `CmMessageSend` function.

A given `CmMessage` object may be sent consecutively to several servers.

A `CmMessage` is not a-priori associated to any connection. Instead, either it is used for sending informations, and in this case connections (using `CmConnect` objects) will be created (or referenced) dynamically or if it used for receiving informations, the `CmMessage` itself is brought along with the transfered data (and automatically instanciated locally in the application's space).

Returned value The reference of the newly created `CmMessage` .

10.1.3 CmMessageDelete

`Syntax` `void CmMessageDelete (CmMessage message)`

Description Destroys the `CmMessage` object.

Greatest care should be taken not to destroy `CmMessage` objects received from another application : the `Cm` engine is taking care of it automatically. Killing them would damage the system.

10.1.4 CmMessageReset

`Syntax` `void CmMessageReset (CmMessage message)`

Description This functions sets the `CmMessage` back to the original state it had just after its creation, losing in particular all of its previously stored informations.

10.1.5 CmMessageSetType

`Syntax` `void CmMessageSetType (CmMessage message, char* type)`

Description Defines the message type. The type can be redefined as many times as wished, only the last value at sending time is relevant.

Types are used for selecting at the reception side a specialized handler declared in the client application using the `CmMessageInstallHandler` function.

Parameters `type` The type given to this message.

10.1.6 CmMessageSend

Syntax `int CmMessageSend (CmMessage message, char* name)`

Description This function sends the specified `CmMessage` object to a server. The internal mechanism is such that one may consider that one copy of the `CmMessage` is actually sent over the network, and will be received as it is by the server, thus preserving the accumulated informations.

Once a `CmMessage` has been sent, it is logically closed for further data accumulations. Thus any further `CmMessagePutXxx` operation will automatically re-open it, clearing the internal buffers, and accumulate again into a fresh area.

Parameters `name` The name of the server.

Returned value A boolean value (1 or 0) describing the correctness of the sending operation. A null value indicates an error while sending the message, showing usually that the client application is not currently active.

One should notice that in case of error, the close operation that would occur after any send is not performed, permitting in particular to try again to send it.

An internal timeout check is performed while sending the message. If the timeout duration is exceeded a error value 1 is returned. The present value of the timeout duration is 10 seconds (except for OS9: 1 second).

10.1.7 CmMessagePutXxx

Syntax Several versions of this function are available, according to the type of the value that is to be accumulated within the `CmMessage` :

- `void CmMessagePutChar (CmMessage message, char value)`
- `void CmMessagePutShort (CmMessage message, short value)`
- `void CmMessagePutInt (CmMessage message, int value)`
- `void CmMessagePutLong (CmMessage message, long value)`
- `void CmMessagePutFloat (CmMessage message, float value)`
- `void CmMessagePutDouble (CmMessage message, double value)`
- `void CmMessagePutText (CmMessage message, char* value)`
- `void CmMessagePutBytes (CmMessage message,`
  `char* value, int length)`
- `void CmMessagePutArray (CmMessage message,`
  `CmMessageArrayType type,`
  `int elements, void* address)`
- `void CmMessagePutExtArray (CmMessage message,`
  `CmMessageArrayType type,`
  `int elements, void* address)`

Description These functions accumulate typed values within the internal data buffers of the specified `CmMessage` object. The values are internally formatted in a machine-independant way for transparent transfer.

Arrays can be installed either with direct copy into the `CmMessage` 's frame or by referencing it. In the latter case, data will be used only when the message is sent to some application, putting the array at that time only onto the network. The `CmMessage` internal buffers are not extended either in this case.

Parameters

`value`	The value to be installed.
`length`	The size of the byte array (only for the `CmMessagePutBytes` function).
`type`	The element type for arrays (only for the `CmMessagePutArray` functions). The value may be chosen among one of the following constant values : • `CmMessageChar` • `CmMessageShort` • `CmMessageInt`
`elements`	The number of elements in the array (only for the `CmMessagePutArray` functions).
`address`	The address of the first element of the array.

10.1.8 CmMessageGetType

`Syntax` `char* CmMessageGetType (CmMessage message)`

Description This function reads the type of the specified message. It is typically used within a reception handler and is often usefull when the handler is a *default* handler.

Returned value A character string representing the type. Be carefull not to modify in any manner the return string this a direct pointer to the actual message type is returned.

10.1.9 CmMessageGetXxx

`Syntax` Several versions of this function are available, according to the type of the value that has to be extracted from the message.

- `char CmMessageGetChar (CmMessage message)`
- `short CmMessageGetShort (CmMessage message)`
- `int CmMessageGetInt (CmMessage message)`
- `long CmMessageGetLong (CmMessage message)`
- `float CmMessageGetFloat (CmMessage message)`
- `double CmMessageGetDouble (CmMessage message)`
- `char* CmMessageGetText (CmMessage message)`
- `char* CmMessageGetBytes (CmMessage message, int* returnedLength)`
- `void* CmMessageGetArray (CmMessage message,`
  `CmMessageArrayType* type,`
  `int* elements)`

Description These functions sequentially decode values from the internal message buffer. Values are not actually copied into the user's space, except for simple numeric ones. Arrays and strings are merely returned as a reference to the actual value within the message buffer. Their life time is therefore limited to the duration of the reception handler from which these functions are called.

For functions that accept additionnal information (such as `CmMessageGetBytes` or `CmMessageGetArray`) a reference to the variable that is meant to receive the value is given in the argument list. If the `NULL` poiter value is provided, then this argument is ignored.

Parameters

`returnedLength` The effective length of the byte array retrieved (Only for the `CmMessageGetBytes` function).

`type` The element type of the returned array (only for the `CmMessageGetArray` function). Its value can be one of :

- `CmMessageChar`
- `CmMessageShort`
- `CmMessageInt`

`elements` The number of elements in the returned array (only for the `CmMessageGetArray` function).

Returned value The value retrieved from the message : either the value itself for simple numeric values or a reference to it for arrays or strings.

10.1.10 CmMessageGetItemType

`Syntax` `CmMessageItemType CmMessageGetItemType (CmMessage message)`

Description This function accesses the item type of the current item within the data buffer of the specified message.

This information may be used for instance to check whether no more information is available from this message.

Returned value A value describing the type of the current item. It may present one of the following constant values (available from the `CmMessage.h` header file) :

- `CmMessageItemChar`
- `CmMessageItemShort`
- `CmMessageItemInt`
- `CmMessageItemLong`
- `CmMessageItemFloat`
- `CmMessageItemDouble`
- `CmMessageItemArray`
- `CmMessageItemText`
- `CmMessageItemBytes`
- `CmMessageItemTail`

10.1.11 CmMessageInstallPrinter

Syntax `int CmMessageInstallPrinter (CmConnectPrinter printer)`

Description This function installs a printer operator

10.1.12 CmMessageInstallDefaultHandler

Syntax `void CmMessageInstallDefaultHandler (CmMessageHandler handler)`

Description This function installs a reception handler that will be activated (by the `Cm` engine) for any message which type is not handled.

The handler should likely be able to cope with any unforeseen message type (and its data format too !). The use of the `CmMessageGetType` is usually recommanded in such a handler.

Handlers should be prototyped as follows :

```
CmMessageStatus handler (CmMessage message, char* sender, char* serverName);
```

10.1.13 CmMessageInstallHandler

Syntax `void CmMessageInstallHandler (CmMessageHandler handler, char* type)`

Description This function installs a reception handler meant to be activated (by the `Cm` engine) when the message type matches the character string specified here.

The handler specified here is assumed to know about the data format within messages of that particular type.

Handlers should be prototyped as follows :

```
void handler (CmMessage message, char* sender, char* serverName);
```

10.1.14 *any-message-handler*

Syntax `CmMessageStatus` *any-message-handler* `(CmMessage message,`
`char* sender,`
`char* serverName)`

Description This is how a user defined reception handler should be defined for a proper activation by the `Cm` engine.

The handler must return either `CmMessageOk` or `CmMessageBreak` when the innermost wait loop must be broken.

Parameters

`sender`	The application's name that sent this message. This name can be used to (for instance) answer directly to the caller (this may be done right from the handler itself).
`serverName`	The server's name to which the message was sent.

10.1.15 CmMessageUninstallHandler

`Syntax` `void CmMessageUninstallHandler (char* type)`

Description This function uninstalls a reception handler previously declared for the specified message type.

10.1.16 CmMessageCheck

`Syntax` `CmConnectCondition CmMessageCheck (void)`

Description This function checks in non-blocking mode whether any of the existing connection has a pending message. If any, the associated handler (when defined) will be executed.

On another hand, this function checks the internal service messages such as connection requests, connection losses and previously inactivated connection destructions (deleting the associated `CmConnect` objects).

The critical role played by this function for physical connection management makes it strongly required whenever a long loop is to be performed in the application. In such a case, the developper should ensure that `CmMessageCheck` is regularly called within such a loop, specially if messages are sent within the loop.

Returned value The condition under which the check completed. On of the following values may be returned :

- `CmConnectNormalCompletion`
- `CmConnectNoHandler`
- `CmConnectBreakDetection`
- `CmConnectErrorCondition`

The role and behaviour of this function is strictly the same as those of the `CmConnectCheck` function.

Example

```
/* File example12.c */

/*
  Application looping on operations while being
  listening to incoming messages.
*/

#include <stdio.h>
#include <CmMessage.h>

main ()
{
  CmMessageOpenServer ("Toto");

  for (;;)
  {
    CmMessageCheck ();

    /*
       Any message received prior to the check function
       will trigger the activation of the corresponding
       handler (if any).
    */

    /* Actual processing loop ... */
  }
}
```

Fig.19- Non-blocking message reception by `CmMessage` objects.

Example

```
/* File example11.c */

#include <stdio.h>
#include <CmMessage.h>

CmMessageStatus my_handler (CmMessage message, char* sender,
                            char* serverName);

main()
{
  CmConnectCondition condition;

  if (!CmMessageOpenServer ("Client1"))
  {
    fprintf (stderr, "Declaration error.\n");
    return (0);
  }

  CmMessageInstallHandler (my_handler, "Image");

  condition = CmMessageWaitWithTimeout (1.0);

  switch (condition)
    {
    case CmConnectTimeoutDetection :
      /* The image did not come */
      break;
    case CmConnectBreakDetection :
      /* The image is arrived */
      break;
    case CmConnectNoHandler :
      break;
    case CmConnectErrorCondition :
      break;
    }

}

/* (to be continued) ... */
```

```
      /* ... File example11.c (continued) */

      CmMessageStatus my_handler (CmMessage message, char* sender,
                                  char* serverName)
      {
        char* text;

        text = CmMessageGetText  (message);

return (CmMessageBreak);
      }
```

Fig.20- Interrupting a wait operation from a handler using `CmMessageBreak`.

10.1.17 CmMessageServerCheck

`Syntax` `CmConnectCondition CmMessageServerCheck (char* namePattern)`

Description This function checks in non-blocking mode whether any of the existing servers whose name match the pattern has a pending message. If any, the associated handler (when defined) will be executed.

Otherwise, the behaviour is similar to the `CmMessageCheck` function.

Parameters `namePattern` The regular expression pattern used to select the servers.

Returned value The condition under which the check completed. On of the following values may be returned :

- `CmConnectNormalCompletion`
- `CmConnectNoHandler`
- `CmConnectBreakDetection`
- `CmConnectErrorCondition`

Example

```
/* File example13.c */

/*
  Application looping on operations while being
  listening to incoming messages.
*/

#include <stdio.h>
#include <CmMessage.h>

main ()
{
  CmMessageOpenServer ("A");
  CmMessageOpenServer ("B");
  CmMessageOpenServer ("C");

  for (;;)
  {
    CmMessageServerCheck ("[BC]");

    /*
       Any message received prior to the check function
       will trigger the activation of the corresponding
       handler (if any) if they had been sent to a server
       whose name contain B or C.
    */

    /* Actual processing loop ... */
  }
}
```

Fig.21- Non-blocking message reception for selected servers..

10.1.18 CmMessageWait

Syntax `CmConnectCondition CmMessageWait (void)`

Description This function is similar to the `CmMessageCheck` function except that it waits continuously instead of just checking for the messages. Handlers are activated and internal service activities are performed. However this function may complete when a message is detected on a connection that has no declared handler.

Returned value The condition under which the wait completed. One of the following values may be returned :

- `CmConnectNoHandler`

- `CmConnectTimeoutDetection`
- `CmConnectBreakDetection`
- `CmConnectErrorCondition`

The role and behaviour of this function is strictly the same as those of the `CmConnectWait` function.

10.1.19 CmMessageWaitWithTimeout

`Syntax` `CmConnectCondition CmMessageWaitWithTimeout (double seconds)`

Description This function is similar to the `CmMessageWait` function except that it waits up to a given number of seconds. The timeout is given as a floating point value of seconds.

Parameters `seconds` The floating point value of seconds until the wait is forced to complete.

Returned value The condition under which the wait completed. One of the following values may be returned :

- `CmConnectNoHandler`
- `CmConnectTimeoutDetection`
- `CmConnectBreakDetection`
- `CmConnectErrorCondition`

10.1.20 CmMessageServerWait

`Syntax` `CmConnectCondition CmMessageServerWait (char* namePattern)`

Description This function is similar to the `CmMessageServerCheck` function except that it waits continuously instead of just checking for the messages for the selected servers. Handlers are activated and internal service activities are performed. However this function may complete when a message is detected on a connection that has no declared handler. The messages sent to servers whose name does not match the pattern are left pending and stacked until a non-selective wait function is used.

Parameters `namePattern` The regular expression pattern used to select the servers.

Returned value The condition under which the wait completed. One of the following values may be returned :

- `CmConnectNoHandler`
- `CmConnectTimeoutDetection`
- `CmConnectBreakDetection`
- `CmConnectErrorCondition`

10.1.21 CmMessageServerWaitWithTimeout

Syntax `CmConnectCondition CmMessageServerWaitWithTimeout (char* namePattern, double seconds)`

Description This function is similar to the `CmMessageServerWait` function except that it waits up to a given number of seconds. The timeout is given as a floating point value of seconds.

Parameters

`namePattern`	The regular expression pattern used to select the servers.
`seconds`	The floating point value of seconds until the wait is forced to complete.

Returned value The condition under which the wait completed. One of the following values may be returned :

- `CmConnectNoHandler`
- `CmConnectTimeoutDetection`
- `CmConnectBreakDetection`
- `CmConnectErrorCondition`

10.2 The `CmConnect` functions.

The `CmConnect` class implements the basic connection mechanisms, hiding the internal transport layer (based itself on TCPIP and the C socket interface).

Generally, the `CmMessage` interface is enough to access the `Cm` features. However, some general requests about the knowledge base of `Cm` are performed using the direct `CmConnect` interface.

In most of the `CmConnect` 's methods, the first argument is a reference to the `CmConnect` object acted upon in this method. For all these situations, the corresponding argument is not documented explicitely.

10.2.1 CmConnectNew

Syntax `CmConnect CmConnectNew (char* name)`

Description This function establishes a new connection to another server (found in the same `Cm` domain). If the connection already exists, it is merely reused.

This function is useful when one wants to establish at a desired time the connection to a given server, avoiding for instance any undesirable time penalty in a real-time application when the first message would be sent or received to/from it.

Parameters

`name`	Server name which one wants to communicate with.

Returned value The `CmConnect` object that supports the connection.

The NULL value is returned in case of error (likely, there is no such server active under this name).

10.2.2 CmConnectGetReference

Syntax `CmConnect CmConnectGetReference (char* name)`

Description This function looks for a `CmConnect` object supporting an active connection to the specified server.

Parameters `name` The logical `Cm` name of the server.

Returned value The `CmConnect` object supporting the connection or the `NULL` value.

10.2.3 CmConnectGetName

Syntax `char* CmConnectGetName (CmConnect connect)`

Description Yields the server's `Cm` name currently connected through the specified `CmConnect` object. This is often usefull within a handler since a reference to the `CmConnect` object that produced data is provided as the first argument to any handler.

Returned value The server's name or the `NULL` value if the `CmConnect` object is no longer active.

10.2.4 CmConnectGetHost

Syntax `char* CmConnectGetHost (CmConnect connect)`

Description Provide the host name of the specified connection. Use it in conjunction with the `CmConnectGetReference` in a `CmMessage` handler.

Returned value The host's name from which the connection is originating or the `NULL` value if the `CmConnect` object is no longer active.

10.2.5 CmConnectGetOwner

Syntax `char* CmConnectGetOwner (CmConnect connect)`

Description Provide the owner name of the specified connection. Use it in conjunction with the `CmConnectGetReference` in a `CmMessage` handler.

Returned value The user's name who created this connection or the `NULL` value if the `CmConnect` object is no longer active.

10.2.6 CmConnectGetServer

Syntax `CmConnect CmConnectGetServer (CmConnect connect)`

Description Returns the `CmConnect` object representing the server which actually handles this connection.

Returned value The `CmConnect` object representing the server.

10.2.7 CmConnectCleanup

Syntax `void CmConnectCleanup (void)`

Description This function closes all currently active connections, deletes all objects managed by `Cm` and cleans up the dynamically allocated memory used by `Cm` .

This is clearly a function that has to be called only at the very end of a `Cm` application. It may also show various information messages about possibly uncorrectly cleaned up items. These messages are only informational but might be sent to the `Cm` 's author for debugging purposes.

10.2.8 CmConnectSelectServer

Syntax `CmConnect CmConnectSelectServer (char* name)`

Description This function permits to select one of the servers managed within the current application to become the *current* server. This notion has no real internal meaning except that it can be inquired using the `CmConnectWhoAmI` function.

Parameters

`name`	The name of one of the servers actually managed by the application. If the name is not recognized, the *current* server is not changed.

Returned value A reference to the private `CmConnect` object of the current application.

10.2.9 CmConnectWhoAmI

Syntax `CmConnect CmConnectWhoAmI (void)`

Description This function permits to access the personnal connection of the current application. This particular connection is used to handle the connection requests from other applications. The `CmConnect` object may then be used for management purposes (such as retrieving the actual `Cm` name of the application using the `CmConnectGetName` function).

Returned value A reference to the private `CmConnect` object of the current application.

10.2.10 CmConnectGetNameServer

`Syntax`	`CmConnect CmConnectGetNameServer (void)`
Description	This function permits to access the personnal connection established with the `NameServer` . This `CmConnect` object may then be used for communicating with it (together with the `CmConnectGetName` function).
Returned value	A reference to the `CmConnect` object used for the `NameServer` .

11 History of changes

11.1 Changes since the version v8r1.

Add the possibility to kill a CmConnect NULL. cm kill "¡null¿".
Add CmMessageLong in CmMessageArryType enum to be used in CmMessagePutArray.
Add the possibility to send long type with "cm send " command (it was before possible with int type (implemented with long actually).
Add defined (CM_LINUX) in Block () DontBlock () in order to avoid problem for large data transfer on Linux.
Add CmConnectKill (c) in CmServerInfosHandler: this handler is activated.
each time a server S1 tries to communicate with another one S2 (at the first communication try ONLY). Hence if S2 has kept in memory an old obejct connect corresponding to S1, it will be cleanup before creating the new object Connect in S2.

11.2 Changes since the version v7.

These versions fix several internal problems found in the database file management, and introduce the transaction management mechanisms.

The package has also been cleaned up using the `Insure++` tool.

The interactive `Cm` exerciser (the `cm send` utility) is also introduced.

The configuration management is now entirely and explicitely managed within the context of the `methods` environment.

11.3 Changes since the version v6r1.

This version introduces quite major changes, especially in the internal protocol management, since all synchronous aspects of the internal management are now removed. The complete protocol, be it between applications and the `NameServer` or between applications is now asynchronous and fully based on `CmMessages` objects. In particular, sending `CmMessages` is now asynchronous, and although the `CmMessageSend` function still provides the same functionality as before, its internal behaviour relies now on a loop over `CmMessageWait` allowing the reception of messages while the transmisson of individual packets is processed.

A new function `CmMessagePost` is provided in order to start the transmission of a (maybe long) message without blocking the calling application. Termination is acknowledged by specifying as an argument to this function a *termination handler*.

One should note that the `NameServer` is now a full `CmMessage` oriented application and that the various service messages exchanged between applications or between applications and the `NameServer` are all based on `CmMessage` objects. The result of this new mechanism is that even

the connection requests are non-blocking and can be managed entirely while a real-time activity is performed.

This should not imply major changes in the user programming interface for the vast majority of normal `Cm` users since only the internal *acknowledge* control mechanisms have been removed.

The only major change in the user programming interface concerns the syntax of message *handlers* which should now return a value (that can be either `CmMessageOk` or `CmMessageBreak`) that indicates whether the current wait loop must be broken or not. The previous mechanism was using the `CmMessageBreak` function which is now removed from the `Cm` interface. The main impact of this change is that handlers may now have wait loops themselves and that the break operations act upon the most inner wait loop.

Besides the fact that this version was a major change, a few clean-up actions have been performed on the implementation, that deal with :

- Internal error management : the production and printing of internal error messages is now handled by a customizable *printer* operator (set by default to be the C standard function `vprintf`). A special function (`CmMessageInstallPrinter`) permits to specify a user defined printer operator, that will be called whenever an error message has to be delivered from the internals of `Cm` .
- The startup functions like `CmMessageStartup`, `CmMessageStartupMultiple`,

  `CmConnectStartupAsServer` and `CmConnectStartupAsMultipleServer` are now really obsolete, and will produce a informational message and return *zero* when called. The recently introduced functions `CmMessageOpenServer` and `CmMessageOpenMultipleServer` are now the unique way to initialize a `Cm` server.
- Several remote control mechanisms have been introduced in order to make applications produce debug or trace informations. Debug control is meant for quite low-level debugging purposes since it would trace the primary calls to the TCPIP entry points. Trace actions permit to show the contents of `CmMessages` exchanged by applications. Three levels of tracing permit to show either simple informations up to complete and detailed dump of message data (the latter should be used carefully since huge prinouts may result from this action).
- The communication with the `NameServer` is now based on asynchronous `CmMessages` . Therefore the `CmConnectAskNameServer` has been removed. It must be replaced by a `CmMessage` built with the request, and a message handler installed according to the required request type (see the section on the `NameServer` for a detailed description of the message formats).
- The internal package that handle the basic collection management or memory management is now based on the version `v2` of `CSet` which thus should be installed accordingly.

11.4 Changes since the version v6.

This new version adds the capability of waiting selectively for messages sent to one of the servers declared within one `Cm` application. This is achieved only within the `CmMessage` environment by providing the new wait functions to the `CmMessage` package :

- `CmMessageWait`
- `CmMessageWaitWithTimeout`
- `CmMessageServerWait`
- `CmMessageServerWaitWithTimeout`

The first two functions duplicate the two wait functions that were already available in the Connect package (with identical argument syntaxes) and the two others perform a selective wait (without or with a timeout specified). For these two selective wait functions, a pattern for selecting one or several server names is provided as the first argument.

The same duplication has been offered (for the sake of symetry!) for the non-blocking check functions with :

- `CmMessageCheck`
- `CmMessageServerCheck`

The following example shows how to wait for messages sent to servers which names start with an 'A' then wait for any message :

```
main()
{
   ...

   CmMessageServerWait ("^A");

   CmMessageWait ();

}
```

Fig.22- Selective wait on servers.

A change in the parameter syntax for the user defined message handlers reflects now the fact that a particular message has been sent to a particular server. Therefore, a third argument receives the server name to which this message was addressed (The first argument is still the reference of the message and the second argument is still the name of the sender).

Connections are now selectively handled by the different servers that a given application declares. Therefore, several connections may have the same name as long as they are managed by different servers. On the other hand, one may now retrieve the server that handles a particular connection, using the `CmConnectGetServer` function. The server that handles the servers themselves is always the `NameServer` .

Few fixes have been done in this release in the mechanisms involved when deleting a message object, in order to effectively release the memory exploited by this object. The consequence of this is that the `CmMessageCleanup` operation should end up now with a completely released memory.

11.5 Changes since the version v5r1.

This new version introduces some major changes in the protocol between applications and the `NameServer` that will increase the security level or even bring new functionnalities.

New features for management issues are supported here that introduce in particular the notion of *domain* in order to give a more centralized vision of `Cm` environments.

A summary of the new features or changes is :

- Introduction of a mechanism for starting several *servers* from within one single application. This mechanism permits to define servers (each still defined by its name - and corresponding to one port number) independently from the fact they would be managed in one application.

They are viewed from their clients as separate objects. This is handled by *opening* or *closing* the servers (entry points that can be used in place of the previous *startup*xxx old ones - still available however for some time).

- The protocol between applications handles now the user name of the application's owner. This owner name can be checked upon in order to introduce security controls using the `CmConnectGetOwner` function. It will be possible later on to configure the `NameServer` for some level of automatic security checking.
- The port number allocation algorithm has been enhanced by taking the machine spaces explicitely into account. Then, the busy port numbers are now detected and automatically discarded from the resource set.
- Domains are introduced for describing each particular `Cm` environment. Each domain corresponds to a particular set of environment variables that describe uniquely the address domains that can be reached by `Cm` applications, and are identified by a unique name. A dictionnary of the defined domains is maintained within the `Cm` management area, and the package will check at run time whether one of these domains is actually selected and setup.
- The `NameServer` database is now internally handled by the `NameServer` application (instead of the driver script) which will ensure a greater lelel of reliability.
- The `NameServer` is now able to control and limit the expansion of its log file (that used to produce system hang up by disk space exhaust).
- The cleanup mechanisms used while quitting the `Cm` package has been re-designed, taking care of thoroughly flushing the allocated spaces when it's no more needed or when the application finishes.

Some defficiencies or bugs have also been fixed up :

- It is possible to uninstall a message handler using the `CmMessageHandlerUninstall` function.
- The `CmMessageCleanup` or `CmConnectCleanup` functions are available for cleaning up the memory allocated during a `Cm` session. (In principle, after calling this cleanup function, one should be able to re-open a server...)

11.6 Changes since the version v5.

This release deals mainly with bug fixing but improves slightly the reporting facilities for the `NameServer` :

- A few checks have been added for allocating port numbers by the `NameServer` , with the detection of busy port numbers (even for number chosen within the right range defined in a `Cm` environment).
- Some new commands are understood by the `NameServer` (and served by the `AskNameServer` application) that help getting statistics informations.
- A correction solves a problem that sometimes forced `Cm` applications to exit after a restart of the `NameServer` .

11.7 Changes since the version v4.

Several important features have been introduced in this `Cm` version :

- Full support of binary exchange format for any C type (floating values in particular)

- Management of multiple instances of identical tasks. The generic task is declared with a `Cm` name, and an actual name is built by suffixing the sequence number of this task instance to the generic name.
- Possibility of calling the main wait loop with a timeout.
- Possibility of requesting a break in the main loop from within a Message handler.
- Queries about the items in a message : a function asks about the type of the next item while retrieving message data.

11.8 Changes since the version v3.

This is a reminder of the changes between the version v3 and the version v4.

- `NameServer` data persistency. This feature makes the active `Cm` tasks unaffected by a stop (or a crash) of the `NameServer` , since it can be reactivated with the knowledge of the system state previously to the stop (particularly the set of allocated TCPIP port numbers). Active tasks are transparently reconnected to the `NameServer` .
- Extension of the supported item types within a `CmMessage` to arrays for handling large chunks of data without actually duplicating them.
- Improvements in the exchange mechanism when different machine architecture are involved (no intermediate ascii format for instance).
- Pesistency of the `CmMessage` data beyond the transfer, allowing in particular multiple transfers of one `CmMessage` .

List of Figures

1 Server name declaration. 4
2 Two-server name declaration. 5
3 State diagram for the `CmMessage` objects. 8
4 Building and sending a `CmMessage` object. 10
5 Receiving `CmMessage` objects with a dedicated handler. 12
6 Non blocking detection of `CmMessage` objects. 14
7 Sending a request with a transaction id 16
8 Building up an answer to a request with a transaction id 16
9 17
10 Dump active transactions in an application 17
11 Remote termination of a transaction 17
12 18
13 Controling the `NameServer` by the special `NameServer.start` script. 28
14 Using the `Cm` exerciser. 29
15 Integration of `Cm` and OnX. 32
16 Using the `Cm` header file in an application. 35
17 The `use` statement for `Cm` 35
18 Setting and checking the `Cm` context. 36
19 Non-blocking message reception by `CmMessage` objects. 47
20 Interrupting a wait operation from a handler using `CmMessageBreak`. 49
21 Non-blocking message reception for selected servers.. 50
22 Selective wait on servers. 57

Contents

1 Presentation. 1

1.1 Document structure. 1

2 Specifications. 2

2.1 Implementation specifications. 2

3 `Cm` architecture. 3

3.1 The `CmConnect` class. 3

3.1.1 Application initialisation. 4

3.1.2 Data conversion on machine architecture basis. 5

3.2 The `CmMessage` class. 5

3.3 The internal protocol managed by `CmMessage` objects. 6

3.4 `CmMessage` building and sending. 6

3.5 Receiving a `CmMessage` 10

4 Transaction management in `Cm` . 15

4.1 Description 15

4.2 An example 15

4.3 Remote debugging facilities for transactions 17

4.4 Programming interface 18

4.4.1 CmOpenTransaction 18

4.4.2 CmTerminateTransaction 18

4.4.3 CmCloseTransaction 18

4.4.4 CmIsTransactionTerminated 18

4.4.5 CmGetTransactionObject 19

4.4.6 CmGetTransactionInfo 19

4.4.7 CmCloseTransaction 19

4.4.8 CmTerminateTransaction 19

4.4.9 CmRestartTransaction 19

4.4.10 CmGetTransactionState 20

5 Handling errors in `Cm` . 21

6 The `NameServer` . 22

6.1 The query operations for the `NameServer` 23

6.2 Data persistency in the `NameServer` 26

6.3 Activating the `NameServer` 26

7 `cm send` : The interactive `Cm` exerciser. 29

8 Linking `Cm` to XWindows. 30

9 The `Cm` package : use and installation. 33

9.1 Configuring `Cm` 33

9.2 Building a Cm application and using it. 34
9.2.1 Compiling and linking a Cm application. 35
9.2.2 Running a Cm application. 36
9.3 Porting the Cm package on a new system. 36
9.4 How to figure out the actual situation of a Cm context. 37

10 Application programming interface for Cm . 40
10.1 The CmMessage functions. 40
10.1.1 CmMessageOpenServer - CmMessageOpenMultipleServer 40
10.1.2 CmMessageNew 40
10.1.3 CmMessageDelete 41
10.1.4 CmMessageReset 41
10.1.5 CmMessageSetType 41
10.1.6 CmMessageSend 42
10.1.7 CmMessagePutXxx 42
10.1.8 CmMessageGetType 43
10.1.9 CmMessageGetXxx 43
10.1.10 CmMessageGetItemType 44
10.1.11 CmMessageInstallPrinter 45
10.1.12 CmMessageInstallDefaultHandler 45
10.1.13 CmMessageInstallHandler 45
10.1.14 *any-message-handler* 45
10.1.15 CmMessageUninstallHandler 46
10.1.16 CmMessageCheck 46
10.1.17 CmMessageServerCheck 49
10.1.18 CmMessageWait 50
10.1.19 CmMessageWaitWithTimeout 51
10.1.20 CmMessageServerWait 51
10.1.21 CmMessageServerWaitWithTimeout 52
10.2 The CmConnect functions. 52
10.2.1 CmConnectNew 52
10.2.2 CmConnectGetReference 53
10.2.3 CmConnectGetName 53
10.2.4 CmConnectGetHost 53
10.2.5 CmConnectGetOwner 53
10.2.6 CmConnectGetServer 54
10.2.7 CmConnectCleanup 54
10.2.8 CmConnectSelectServer 54
10.2.9 CmConnectWhoAmI 54
10.2.10 CmConnectGetNameServer 55

11 History of changes 55
11.1 Changes since the version **v8r1**. 55
11.2 Changes since the version **v7**. 55

11.3 Changes since the version **v6r1** 55
11.4 Changes since the version **v6** 56
11.5 Changes since the version **v5r1** 57
11.6 Changes since the version **v5** 58
11.7 Changes since the version **v4** 58
11.8 Changes since the version **v3** 59