CubeMOM
Programming Guide

Copyright

Document version	date	Software version
1.3	07/2025	CubeMOM Release 2.2.30

Contents

CubeMOM Programming Guide

This document is a guide for developing CubeMOM application, and explains CubeMOM and CubeMOM functions (API). Readers of this book should have basic knowledge of UNIX or Linux systems and C programming.

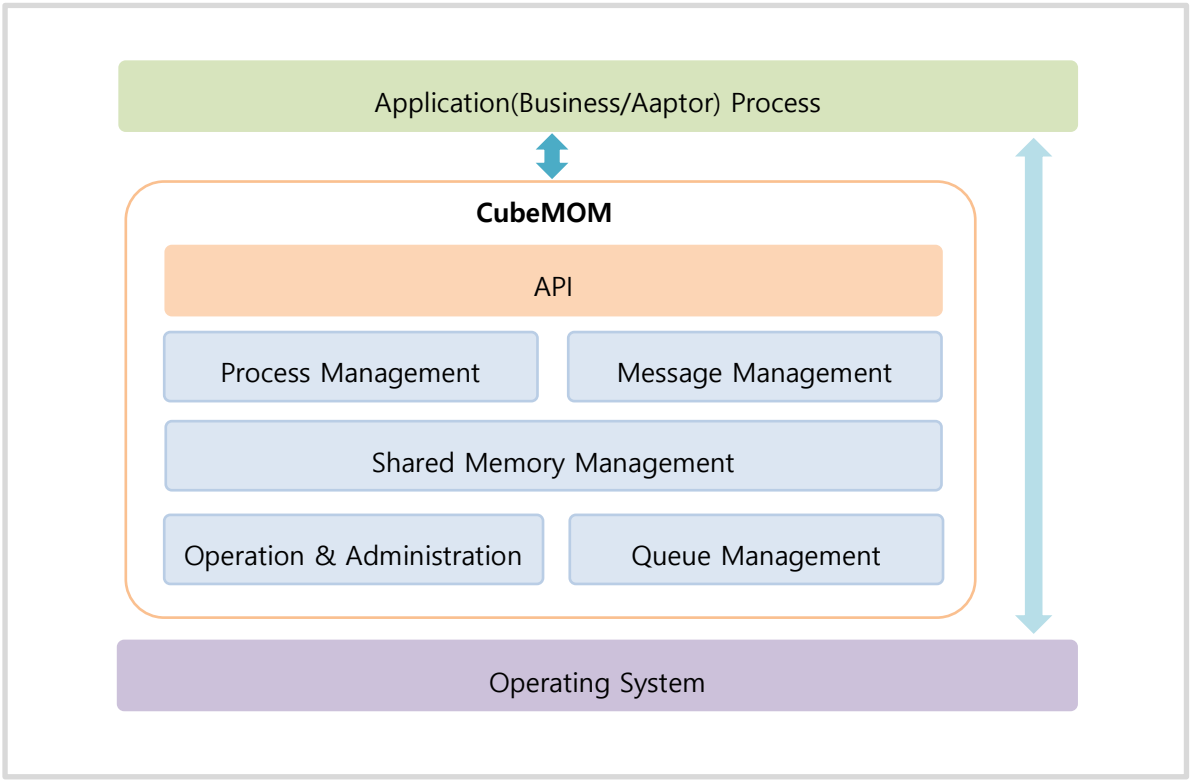
The meta characters in this document are listed in the table below.

Character	Description
< >	Mandatory
[ ]	Optional
|	Exclusive selection delimiter among multiple specified values
*****	Zero or more characters
?	One character
-	Number range
.	Separate objects
,	Separate items
…	Repeat item zero or more times

CubeMOM understanding

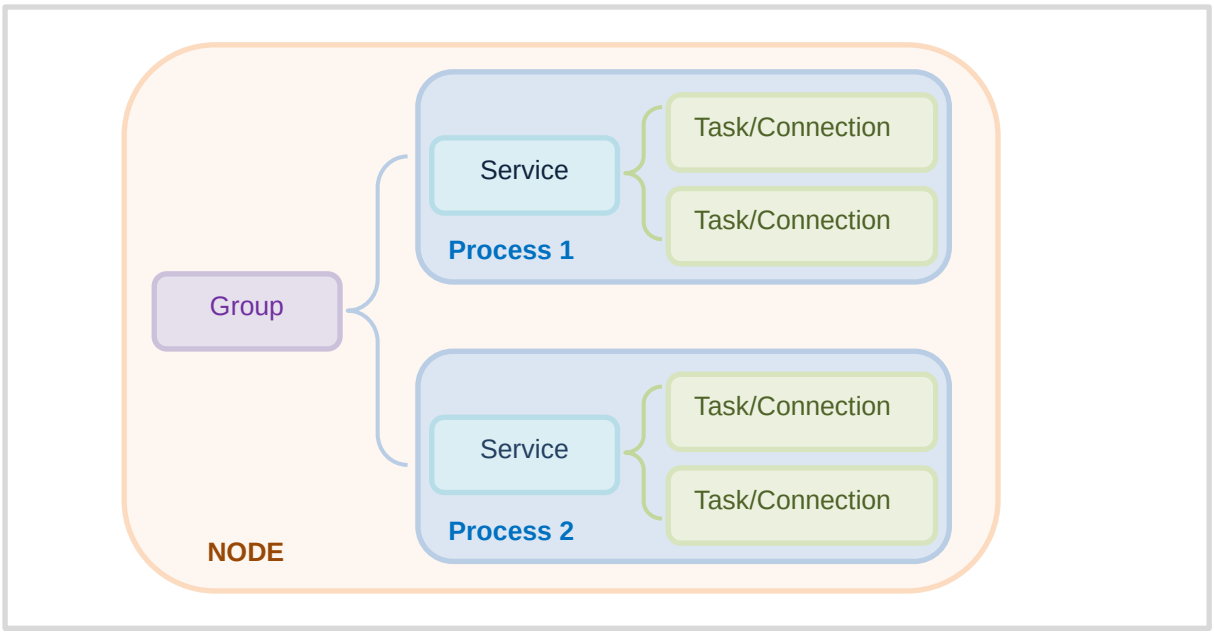
CubeMOM application development requires an understanding of CubeMOM as a prerequisite. To do this, this chapter explains the essentials that developers need to know.

Outline



CubeMOM is a message-oriented middleware that acts as a message carrier from an application perspective. Applications are divided into business programs and adaptor (communication) programs, and business programs are the development target. The adaptor (communication) program serves to connect the business program with other internal or external systems.

Component

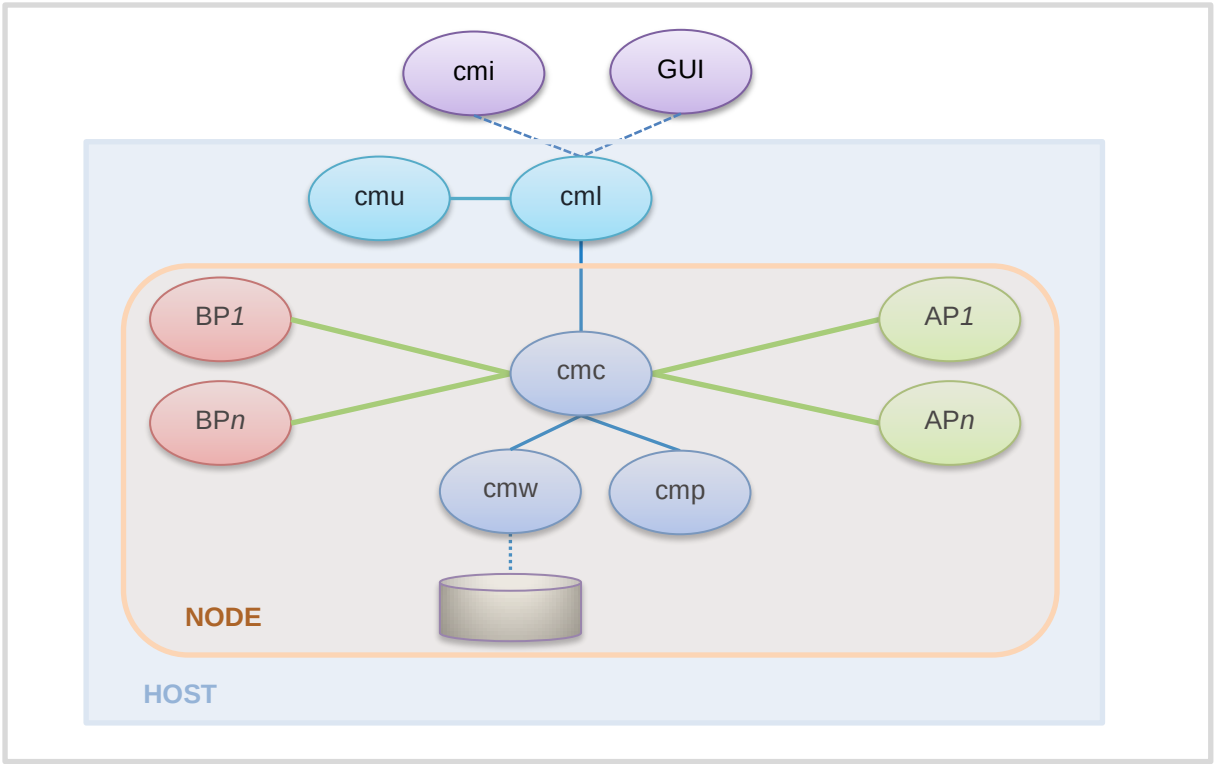


CubeMOM maintains components as logical abstractions of objects. An object contains one or more lower-level objects.

Level	Object	Description
1	Host	Physical computer system
2	Node	Logical computer system
3	Group	Business/Adaptor Process Group
4	Process	Business/Adaptor Process
5	Service	Specific function of business/adaptor process
6	Task	Business process, minimum unit of work
6	Connection	Adaptor process, communication access management

In business programs, services are implemented as functions and can send or receive as many messages simultaneously as the number of tasks the service has.

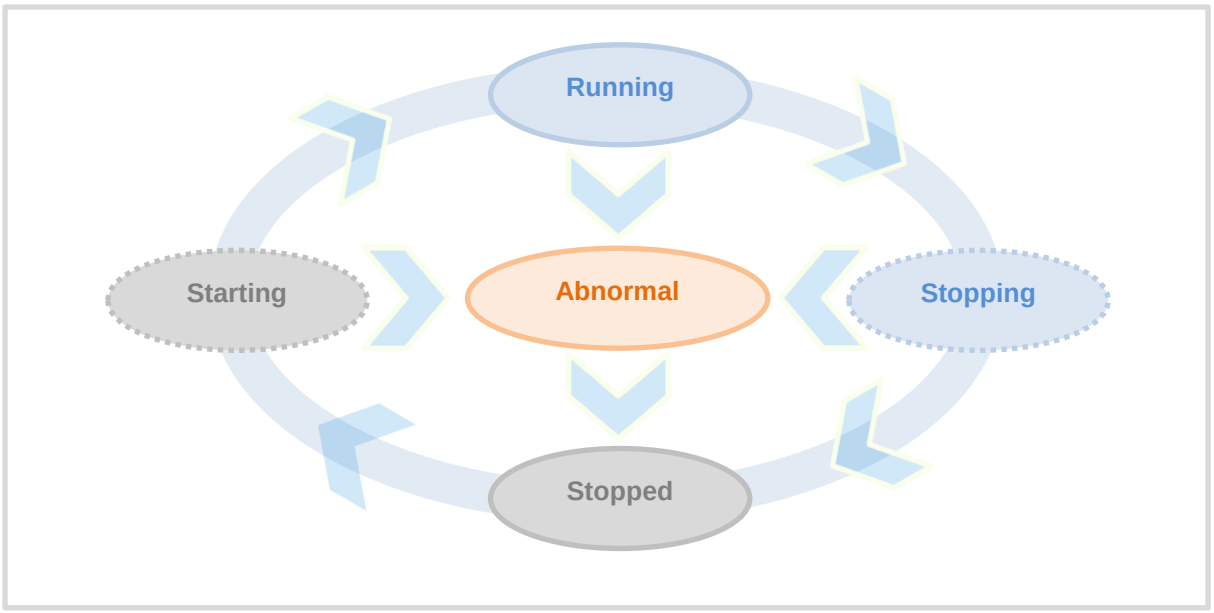
Process Relationship



The figure above is a relationship diagram of the CubeMOM process. The role of each process is shown in the following table.

Process Name	Description
cmi	Command Interpreter
GUI	Graphic(WEB) User Interface
cml	Host, User Interface Listener
cmu	Host, User Interface Server
cmc	Node, Main Process
cmw	Node, Log Process
cmp	Node, Link Process
AP*n*	Adaptor Process Group - Internal/external communication
BP*n*	Business Process Group - Developer implementation

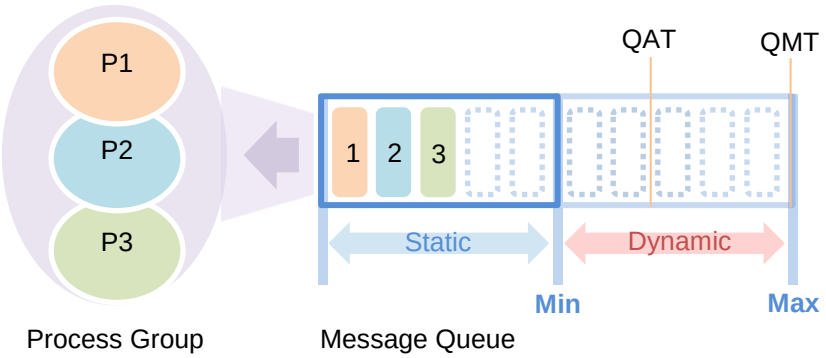
Process State Transition



CubeMOM increases or decreases the number of processes according to user commands or transaction volume and manages the process status. The user can know whether the process is normal or not with the process status.

Process state	Description
Stopped	This is the normal shutdown status, which means the status before startup or normally shutdown.
Starting	This is the state from the start of startup to the completion of connection and authentication with the core process. → Status from the start of the program to before the result of the cm_connect function.
Running	Startup is complete, meaning it has connected and authenticated with the core process.
Stopping	State in the process of shutdown, from the start of the shutdown command to the notification of shutdown completion (ACK).
Abnormal	An abnormal termination status means that it was terminated abnormally.

Message Queue



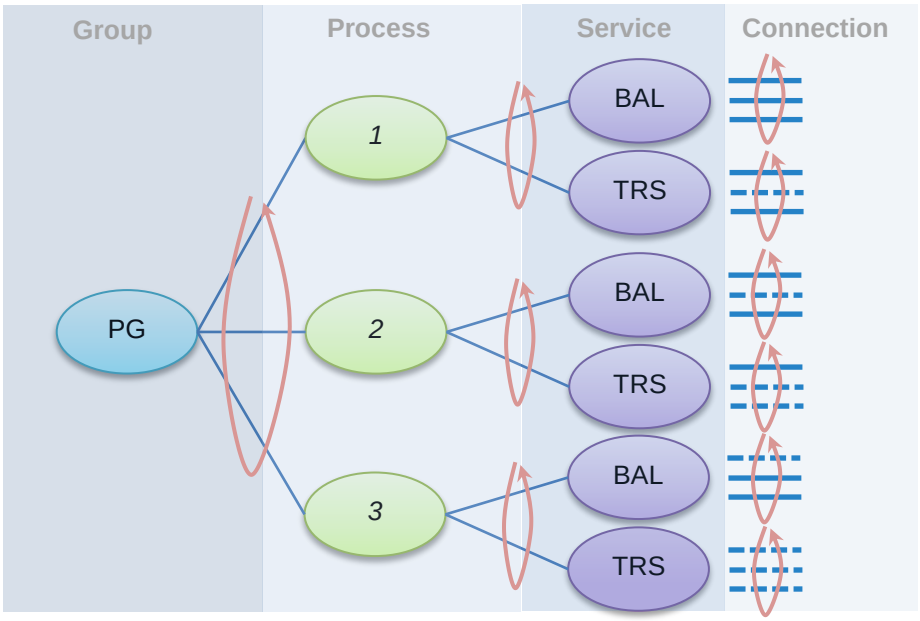
CubeMOM has a message queue per process group. Queue items are separated into statically reserved area and dynamically reserved area. The statically reserved area guarantees message queuing as much as the corresponding number, and the dynamically reserved area determines whether queuing is possible or not depending on the queue storage usage.

Queued messages are forwarded to a specific or random process, depending on the destination, and then removed after receiving an acknowledgment of completion (ACK). Depending on the situation, you may encounter failures as shown in the following table.

Queue-related failures	Description
Queue wait timeout	Occurs when a message has not been delivered to a process for a certain amount of time (set).
Acknowledgment of Completion (ACK) timeout	Occurs when a processing completion notification (ACK) is not received for a certain amount of time (set) after a message is delivered to a process. → Force process termination.
No queue space	Occurs when there are already queued up to the maximum number of messages that can be queued (set).

If a process terminates abnormally without notifying processing completion (ACK) after receiving a message, the message can be retransmitted to another process in the group. However, the elapsed time from the time the message is first queued to the time the process is abnormally terminated must be less than the queue wait time (set).

Message Routing



If the message source process does not specify the destination task or connection, the core process dynamically determines it. The decision method is first available (FA) or round-robin (RR) and which is a process group setting.

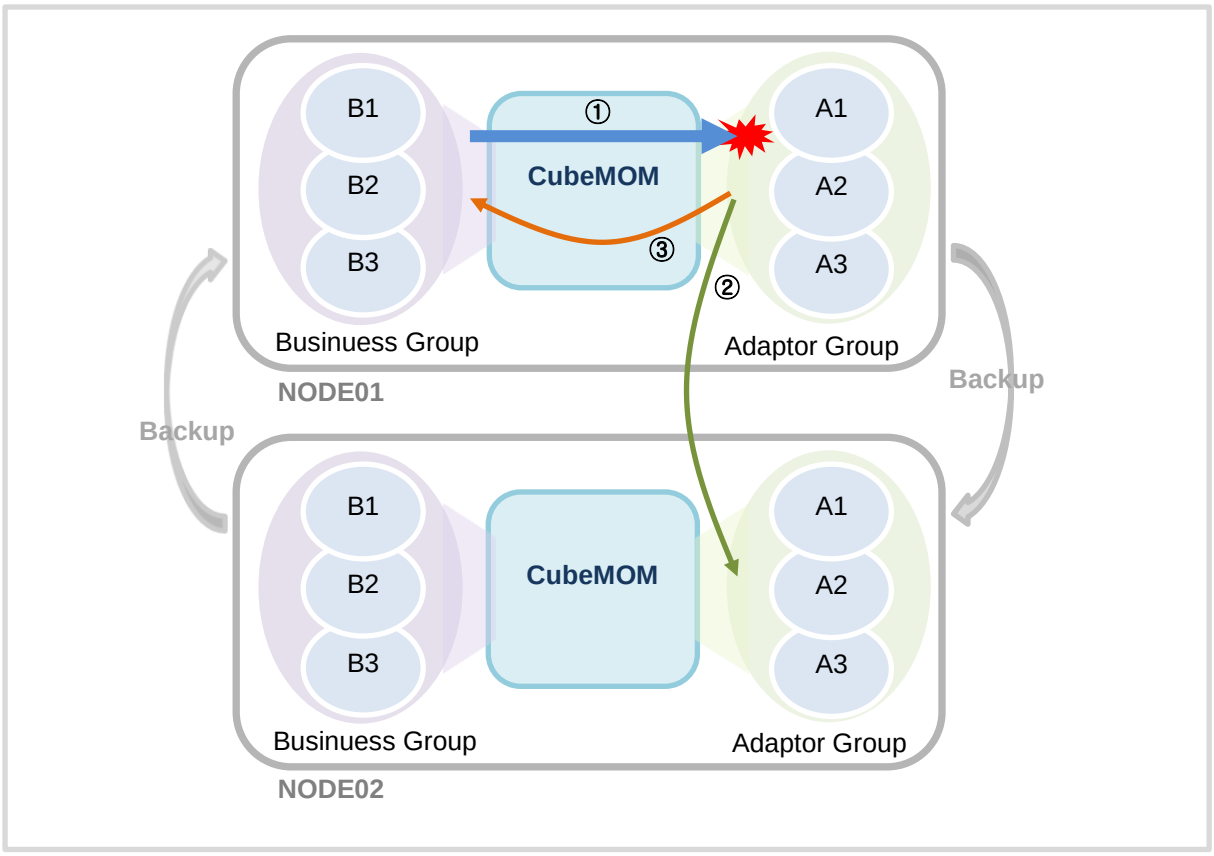
The figure above is a schematic of the example destinations in the table below. If the node of the destination is omitted, the node to which the source process belongs is assumed.

Destination Example	Routing
.PG	Any task/connection that can be sent within the group
.PG.BAL	Any task/connection that can be sent within the service
.PG.1	Any task/connection that can be sent within the process
.PG.1.BAL	Any task/connection that can be sent within the service of ".PG.1.BAL"
.PG.1.BAL.1	Task/Connection of ".PG.1.BAL.1"
.PG.1.BAL.1#243.3	Session "243.3" of ".PG.1.BAL.1"
.*.*.*.*	All groups, all processes, all services, all task/connection
.PG.1-2.BAL.-	PG group, 1~2 process, BAL service, All task/connection

※ **Destination General Form : [Node].PG[[.**PN**][.SV[.**CN**[**#pid.num**]]]]**

※ **Broadcast special characters : '*'(zero or more characters), '?'(any one character), '-'(number range)**

Message Control Flags



The message control flag is a value set by the message source and is a bit flag that determines whether the message is ‘forward to the backup node’ or ‘return to the source’ in case of delivery failure to the destination. That is, a value specifying the handling (‘forward to the backup node’/’return to the source’/’drop’) in case of message delivery failure.

The figure above is a schematic diagram of a message delivery failure situation. “NODE01” and “NODE02” are mutual backup relationship. When a delivery failure (①) occurs from the source to the destination of "NODE01", whether to 'forward to the backup node' (②) or 'return to the source' (③) is determined by the corresponding bit flag ON/OFF. If both bit flags are ON, ‘forward to the backup node’ takes precedence.

If a message that has been forwarded to a backup node fails again, it will again decide whether to 'forward to the backup node' or 'return to the source'. In the figure above, if the message forwarded to “NODE02” due to “NODE01” failure is failed again, “NODE01” has already passed through, so regardless of the bit flag, it is not a target for ‘forward to the backup node’. However, it can 'return to the source' according to bit flag.

The message control flags are a set of bit flags that specify the handling ('forward to the backup node'/'return to the source’/'drop') per failure type, as shown in the table below.

Failure Type	Description
DTNE	Destination not exist
DTNR	Destination not running
DTNS	Destination have not sendable session
DTQF	Destination queue full
QTMO	Queue wait timeout
CPNR	Core process not running
CPQF	Core process queue full
CPQT	Core process queue wait timeout

The bit numbers for each failure type of the message control flag are as follows. If both the ‘forward to the backup node’ (P) bit and the ‘return to the source’ (R) bit are OFF, the message is dropped in case of a failure.

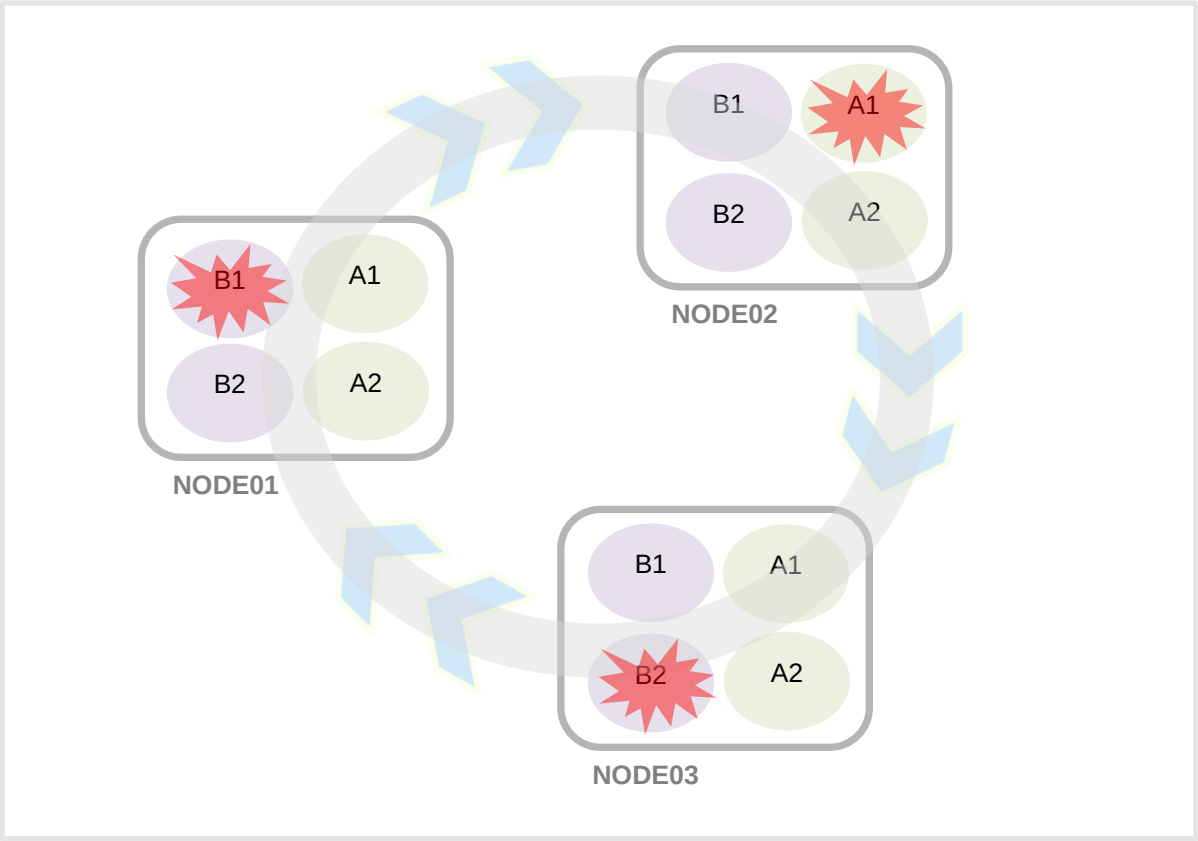
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P	R	P	R	P	R	P	R	P	R	-	R	-	R	-	R
DTNE		DTNR		DTNS		DTQF		QTMO		CPNR		CPQF		CPQT	

• ‘P’ : Whether to forward to the backup node, ‘R’ : Whether to return to the source, ‘-‘ : unused

For example, if the ‘Message Control Flags’ value is “FFD5”, the bit values for each error type are as follows.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1
DTNE		DTNR		DTNS		DTQF		QTMO		CPNR		CPQF		CPQT	

Multiplexing



CubeMOM can configure multiple process groups on multiple nodes to increase the availability of critical tasks. By configuring multiplexing, in case of any node failure (process, queue, communication, …), messages can be processed on the backup node without loss.

In the above figure, the backup of “NODE01” is “NODE02”, the backup of “NODE02” is “NODE03”, and the backup of “NODE03” is “NODE01”. If the 'message control flags' is set to 'forward to the backup node' and 'return to the source' for a certain failure, the message can pass through up to three nodes in the event of a failure, and if all three nodes fail, the message will be returned to the source node from the last node.

When configuring multiplexing, you can configure different node settings depending on your business. Considering this, the value of the ‘message control flags’ must be specified.

CubeMOM Applicaton

CubeMOM application is a program developed using CubeMOM as middleware. This means a program created using the functions (Internal API) provided by CubeMOM. Developers can use CubeMOM functions (Internal API) to increase development efficiency and focus on business logic.

Features

CubeMOM application has the following characteristics compared to other UNIX or Linux programs

- It works as a daemon process as a server program.
- Easy programming without deep understanding of data communication.
- Communication is performed asynchronously.
- Sent messages may be returned in case of a failure.

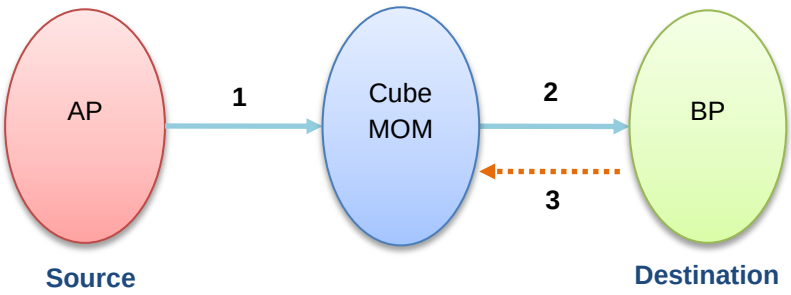
Flow

```
main () {
    1. Process resource creation and initialization.
    2. Connect to CubeMOM.
    while {
        3. Message receiving.
        4. Message processing.
        5. Notification of processing completion.
    }
    6. Disconnect CubeMOM.
    7. Free process resources.
}
```

1. Create and initialize process resources with the cm_initialize function.
2. Connect to CubeMOM with the cm_connect function.
3. Receive message from CubeMOM with the cm_recv function. Upon successful reception, the message type and length are known.
4. Processing of received message. Note that this may be the message returned. Whether a message is returned is determined by the macro function. Ex) CMA_BIT_SETTED(msg_type, CM_MSG_RETURN)
5. 'Notification of processing completion' for message received with the cm_send or cm_notify function. Processes that do not send 'notification of processing completion' within a certain period of time are considered abnormal and killed forcibly.
6. If you receive a process 'stop' command, disconnect from the CubeMOM with the cm_close function.

7. If you receive a process 'stop' command, remove the process resources with the cm_terminate function.

Sending and Receiving Message



The upper figure shows the process until the destination process (BP) receives the message sent by the source process (AP).

1. The sending process (AP) calls the cm_send function to send the message to the destination.
   • A message source must specify a task or connection.
   • Message destination can be omitted except for process group.
   • Specifies message control flags.

2. The receiving process (BP) calls the cm_recv function to receive message.
   • A message source is the task or connection of the sending process.
   • A message destination is the task or connection of the receiving process.
   • You can get the ID for the 'notification of processing completion' of the message.

3. The receiving process (BP) sends a 'notification of processing completion' of the received message. Received message 'notification of processing completion' can be done together with sending a new message or notification alone. Call the cm_send function if you want to 'notification of processing completion' together with sending a new message, and call the cm_notify function if you want to 'notify processing completion' alone.

   • Notify the CubeMOM with the ID geted when the message is received.

The general form of a message source or destination is as follows..

- General form **: [ND].PG[[.PN][.SV[.CN[#PID.SN]]]]**
  ND : Node Name
  PG : Process Group Name
  PN : Process Number
  SV : Svervice Name
  CN : Task or Connection Name
  PID : Process ID
  SN : Session Number

Message Control Flag Constants

The message control flags is an integer value that is a set of bit flags specifying handling (‘forward to the backup node’/’return to the source’/’drop’) for each failure type. The table below shows the constants that designate processing by failure type.

Failure Type	Description		
DTNE	Destination not exist		
	Constant	Hexadecimal	Meaning
	CM_M_BDTNE	0xC000	**P & R**
	CM_M_PDTNE	0x8000	**P**
	CM_M_RDTNE	0x4000	**R**
DTNR	Destination not running		
	Constant	Hexadecimal	Meaning
	CM_M_BDTNR	0x3000	**P & R**
	CM_M_PDTNR	0x2000	**P**
	CM_M_RDTNR	0x1000	**R**
DTNS	Destination have not sendable session		
	Constant	Hexadecimal	Meaning
	CM_M_BDTNS	0x0C00	**P & R**
	CM_M_PDTNS	0x0800	**P**
	CM_M_RDTNS	0x0400	**R**
DTQF	Destination queue full		
	Constant	Hexadecimal	Meaning
	CM_M_BDTQF	0x0300	**P & R**
	CM_M_PDTQF	0x0200	**P**
	CM_M_RDTQF	0x0100	**R**
QTMO	Queue wait timeout		
	Constant	Hexadecimal	Meaning
	CM_M_BQTMO	0x00C0	**P & R**
	CM_M_PQTMO	0x0080	**P**
	CM_M_RQTMO	0x0040	**R**
CPNR	Core process not running		
	Constant	Hexadecimal	Meaning
	CM_M_RCPNR	0x0010	**R**
CPQF	Core process queue full		
	Constant	Hexadecimal	Meaning
	CM_M_RCPQF	0x0004	**R**
CPQT	Core process queue wait timeout		
	Constant	Hexadecimal	Meaning
	CM_M_RCPQT	0x0001	**R**

※ ‘P’ - Forward to the backup node’, ‘R’ - Return to the source

The **CM_M_RALLE** constant means ‘return to the source’ in case of any failure, and the **CM_M_PALLE** constant means ‘forward to the backup node’ in case of any failure. The **CM_CFLAGS_CONFIG** constant means to use the (sender's service) setting value, which can reduce hard coding in the program.

When sending a message, use the bitwise OR operator to combine the constants defined in the table above, or designate the **CM_CFLAGS_CONFIG** constant for the ‘message control flags’ value.

Return Message

The message sent by the source process can be returned to the source according to the ‘message control flags’ in case of delivery failure to the destination. If a message is returned, the error code that occurred first is set as the reason for the return. For example, if a message is returned after passing through multiple nodes, the reason for the return is the error code that occurred first, even if the error occurred at each node is different.

The return message have the CM_MSG_RETURN bit flag in the ‘message type’. You can use the CMA_BIT_SETTED macro function to determine if it is a return message.

Ex) CMA_BIT_SETTED(msg_type, CM_MSG_RETURN)

Return message have higher priority than normal message. The core process preferentially delivers return messages regardless of the order in which the messages were queued.

The ‘message control flags’ value, which determines whether to return to the source due to message delivery failure, is set in consideration of business characteristics, and processing of return messages is a business decision.

CubeMOM API

CubeMOM API consists of two types : functions for CubeMOM application programs (Internal API) and functions for interoperating the legacy system and CubeMOM (Link API).

Summary

Type	Function	Description
Internal API	cm_initialize	Allocate and initialize process resources
	cm_terminate	Terminate process resources
	cm_connect	Connecto to CubeMOM
	cm_close	Disconnect from CubeMOM
	cm_send	Send message
	cm_recv	Receive message
	cm_notify_option	Notification Ack/Event options
	cm_notify	Notification Ack/Event
	cm_get_param	Get parameter value
	cm_split	Separation of message source/destination string
	cm_get_org	Get message orignator
	cm_get_dst	Get message destination
	cm_get_fd	Get file descriptor
	cm_get_strcode	Error code string
	cm_get_errtxt	Error message string
	cm_is_loglevel	Whether to enable log level
	cm_log	Log
	cm_save	Save message
	cm_set_logbuf_size	Set log buffer size
	cm_get_logbuf_size	Get log buffer size
	cm_get_log_sem_key	Get log semaphore key
	cm_get_log_path	Get log directory
	cm_fdump_open	Dump file, open
	cm_fdump_close	Dump file, close
	cm_fdump_printf	Dump file, print
	cm_fdump_lock	Dump file, lock
	cm_fdump_unlock	Dump file, unlock
	cm_get_shmps_addr	Shared memory, Get ‘process’ address
	cm_get_shmsv_addr	Shared memory, Get ‘service’ address
	cm_get_shmad_addr	Shared memory, Get ‘address’ address
	cm_get_xid	Get message tracking number string
	cm_get_session	Get unique session number
	cm_clr_session	Clear unique session number
	cm_get_myname	Get node, group, and process numbers
	cm_get_mypid	Get PID
Link API	cmi_ps_create	Create process object
	cmi_ps_destroy	Destory process object
	cmi_sv_create	Create service object
	cmi_sv_destroy	Destory service object
	cmi_set_sockopt	Set socket options
	cmi_load_certificate	Load certificate
	cmi_set_certificate_nid_commonName	Set certificate NID_commonName
	cmi_listen	Listen

	cmi_accept	Accept connection
	cmi_connect	Connect
	cmi_connect_result	Connect result
	cmi_tls_handshake	TLS handshake
	cmi_close	Close
	cmi_get_svfd	Get ‘service’ file descriptor
	cmi_get_cnfd	Get ‘connection’ file descriptor
	cmi_get_cnstat	Get ‘connection’ status
	cmi_send	Send message
	cmi_recv	Receive message
	cmi_free	Free resource

Error Code

All CubeMOM error codes use negative integer values to distinguish them from OS error codes.

Constant	Error code	Description
CM_NIL	-1	Object not exist
CM_EHEADER	-2	Invalid header
CM_EBUCKET	-3	Invalid hash bucket
CM_BITMAP	-4	Bitmap mismatch to data
CM_EARRAY	-5	Invalid array
CM_ELINK	-6	Invalid link
CM_EEXIST	-7	Object already exist
CM_ENOSPC	-8	No space in memory
CM_EINVAL	-9	Invalid argument
CM_ELIMIT	-10	Over limit
CM_EPERM	-11	Permission denied
CM_EACCES	-12	Not administrator
CM_EALLOW	-13	Not allowed
CM_ERECTYPE	-14	Invalid object type
CM_ERECSTAT	-15	Invalid object state
CM_ESTATRSN	-16	Invalid state reason code
CM_ERECLNUM	-17	Incorrect phyical number
CM_ERECPNUM	-18	Incorrect logical number
CM_ERECCCNT	-19	Incorrect child count
CM_ESDABCNT	-20	Incorrect sendable count
CM_EALLOCCNT	-21	Incorrect memory allocation count
CM_EALLOCOFS	-22	Incorrect memory allocation offset
CM_EALLOCSIZE	-23	Maximum size exceeded
CM_ESESSION	-24	Incorrect session number
CM_EMSGSIZE	-25	Incorrect message size
CM_ETIMEOUT	-26	Timeout
CM_EPROCTYPE	-27	Invalid process type
CM_EPROCNUM	-28	Invalid process number
CM_EOBJNAME	-29	Invalid object name
CM_ERESTARTCM	-31	cubemom needs to be restarted
CM_EHOSTINDEX	-32	Invalid host index
CM_ETHRDCOUNT	-33	Incorrect thread count
CM_EMSGTYPE	-34	Invalid message type
CM_EMSGLEN	-35	Invalid message length
CM_EMAXLEN	-36	Maximum length exceeded
CM_EMSGSRC	-37	Invalid message source
CM_EMSGDST	-38	Invalid message destination
CM_EENVVAR	-40	No environment variable
CM_ECMDNIL	-41	Command not exist
CM_ERANGE	-42	Out of range
CM_EINCORRECT	-43	Invalid input value
CM_ECMDINVAL	-44	Invalid command
CM_EMANNIL	-45	Mandatory attribute required
CM_ENOSYS	-46	Function not implemented
CM_EQFULL	-47	Queue full
CM_EACKTIMEOUT	-48	Acknowledgement timeout
CM_ENOLOGIN	-50	No login ID
CM_ELOGINIDPW	-51	Incorrect login or password
CM_EATTEMPTLIMIT	-52	Exceeded number of attempts
CM_ELOGINLOCKED	-53	Login locked

CM_ELOGINEXPIRED	-54	Login expired
CM_ECHPWSAME	-55	Password same as before
CM_EDTNE	-61	Destination not exist
CM_EDTNR	-62	Destination not running or not connected
CM_EDTNS	-63	Destination have not sendable session
CM_EDTQF	-64	Destination queue full
CM_EQTMO	-65	Queue timeout
CM_ECPNR	-66	Not running or not connected
CM_ECPQF	-67	Core process queue full
CM_ECPQT	-68	Core process queue timeout
CM_ESCNE	-71	Source not exist
CM_ESCNR	-72	Source not running or not connected
CM_ESCNS	-73	Source have not send attribute
CM_ESCQF	-74	Source queue full
CM_ERUNNING	-81	Object is running
CM_ENOTRUNNING	-82	Object is not running
CM_ENOTSTOPED	-83	Object is not stopped
CM_EDISABLED	-84	Object is disabled
CM_ENOHOST	-85	Node not included in host
CM_EPROCSTARTING	-86	Process starting
CM_EPROCSTOPPING	-87	Process stopping
CM_EOBJSTARTING	-88	Child object starting
CM_EOBJSTOPPING	-89	Child object stopping
CM_ENOADDR	-91	Address does not exist
CM_ELOAD	-92	Memory load incompleted
CM_RECONNECTED	-99	Process re-connected to core
CM_EDLOPEN	-101	Dynamic link library
CM_EDLSYM	-102	Can't find function
CM_ECERTCHAIN	-103	Certificate load
CM_EPRIKEY	-104	Private key load
CM_ETRUSTCA	-105	CA(s) load
CM_EVERITYPEER	-106	Peer authentication
CM_EHANDSHAKE	-107	TLS handshake
CM_ESHUTDOWN	-108	TLS shutdown
CM_EVERIFY	-109	Certificate check
CM_ECOMMNAME	-110	Certificate common name mismatch
CM_ETLSSEND	-111	TLS send
CM_ETLSRECV	-112	TLS receive

Macro

Macro functions for CubeMOM application.

CMA_SET_BIT

CMA_CLR_BIT

CMA_BIT_SETTED

Syntax

```
#define CMA_SET_BIT(nBitFlags, nFlag)         ( (nBitFlags) |=    (nFlag) )
#define CMA_CLR_BIT(nBitFlags, nFlag)         ( (nBitFlags) &= ~(nFlag) )
#define CMA_BIT_SETTED(nBitFlags, nFlag)    ( (nBitFlags) &      (nFlag) )
```

Parameters

[in/out] nBitFlags	Integer
[in] nFlag	ON/OFF Bit

Remarks

Turns a specific bit of an integer ON or OFF and determines if a specific bit is ON.

Example Code

```
// Is returned message?
if( CMA_BIT_SETTED(msg_type, CM_MSG_RETURN) ) {
      // do something …
}
```

Internal API

Internal APIs are functions for CubeMOM application.

cm_initialize

Syntax

```
void    *cm_initialize(int argc, char *argv[], int *error);
```

Parameters

[in] argc	Number of arrays of pointers
[in] argv	Array of pointers
[out] error	Integer pointer to store error code in case of error

Return value

If there are no errors, return the process object pointer. A process object pointer is required as an argument to most functions.

In case of an error, it returns a NULL pointer and sets the error code in the argument *error*.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Creates and initializes CubeMOM application, process resources. It also daemonizes processes and creates synchronization objects for shared memory.

The argument *argv* must contain its own identifier (“-cm=$NODE.$GROUP.$PROCESS”). Pass the execution arguments (argc, argv of the main function) received from the core process as they are.

The argument *error* is an integer pointer to store the error code, which will not change if there is no error.

Note The cm_initialize function should be called only once.

See also

cm_terminate

Example Code

```
#include <cubemom.h>

int main(int argc, char *argv[] ) {
	int		error = 0;
	void	*cmobj = NULL;

	cmobj = cm_initialize(argc, argv, &error );

	// do something ...

	if(cmobj) cm_terminate(cmobj);
	return 0;
}
```

cm_terminate

Syntax

```
int    cm_terminate (void *cmobj);
```

Parameters

[in] cmobj	Process object pointer

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Terminate the CubeMOM application, process resources.

See also

cm_initialize

Example Code

```
#include <cubemom.h>

int main(int argc, char *argv[] ) {
	int		error = 0;
	void		*cmobj = NULL;

	cmobj = cm_initialize(argc, argv, &error );

	// do something ...

	if(cmobj) cm_terminate(cmobj);
	return 0;
}
```

cm_connect

Syntax

```
int    cm_connect(void *cmobj);
```

Parameters

[in] cmobj	Process object pointer

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Connect to the core process. When the connection is completed internally, an authentication request message is sent, and if the response to the request is normal, it is considered successful.

The core process changes the status of a process that has completed authentication from ‘starting’ to ‘running’ within a certain (set) time after startup, and changes incomplete processes to 'abnormal'.

Note The cm_connect function should be called only once.

See also

cm_close, cm_initialize, cm_terminate

Example Code

```
#include <stdio.h>
#include <cubemom.h>

int main(int argc, char *argv[] ) {
      int           retval = 0 ;
      int           error    = 0 ;
      int           psnum = 0 ;
      void         *cmobj = NULL;
      char          group [CM_NAME_MAX_LEN+1]; // group name
      char          myname[CM_SDST_MAX_LEN+1]; // group.process
```

```
	cmobj = cm_initialize(argc, argv, &error );
	if( cmobj==NULL ) goto END;

	cm_get_myname(cmobj, NULL, group, &psnum);
	sprintf(myname, "%s.%d", group, psnum);

	if( retval = cm_connect(cmobj) ) {
		cm_log(cmobj, NULL, CM_CRITICAL, NULL, "%s cm_connect error(%d)", myname, retval);
		goto END;
	}

	// do something ...

	cm_close(cmobj);

END :
	if(cmobj) cm_terminate(cmobj);
	return 0;
}
```

cm_close

Syntax

```
int    cm_close(void *cmobj);
```

Parameters

[in] cmobj	Process object pointer

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Disconnect from the core process. Disconnecting is considered process shutdown. You must disconnect only when a process shutdown command is received from the core process. A process that disconnects without receiving a process shutdown command is regarded as an abnormal shutdown.

The core process changes normally shutdown processes to ‘stopped’ status and abnormally shutdown processes to ‘abnormal’ status.

Note The cm_close function should only be called when a process shutdown command is received.

See also

cm_connect, cm_initialize, cm_terminate

Example Code

```
#include <stdio.h>
#include <cubemom.h>

int main(int argc, char *argv[] ) {
    int         retval = 0 ;
    int         error  = 0 ;
    int         psnum = 0 ;
    void        *cmobj = NULL;
```

```
    char        group [CM_NAME_MAX_LEN+1]; // group name
    char        myname[CM_SDST_MAX_LEN+1]; // group.process

    cmobj = cm_initialize(argc, argv, &error );
    if( cmobj==NULL ) goto END;

    cm_get_myname(cmobj, NULL, group, &psnum);
    sprintf(myname, "%s.%d", group, psnum);

    if( retval = cm_connect(cmobj) ) {
        cm_log(cmobj, NULL, CM_CRITICAL, NULL, "%s cm_connect error(%d)", myname,
retval);
        goto END;
    }

    // do something ...

    cm_close(cmobj);

END :
    if(cmobj) cm_terminate(cmobj);
    return 0;
}
```

cm_send

Syntax

```
int    cm_send(void *cmobj, const char *xid, const char *src, const char *dst,
                    int mtype, int stype, const char *msg, int len,
                    int cflags, void **ackid, void *option);
```

Parameters

[in] cmobj	Process object pointer
[in] xid	Message tracking ID
[in] src	Message source
[in] dst	Message destination
[in] mtype	Message type

CM_MSG_DATA	Data

[in] stype	Message subtype

CM_MSGD_NORMAL	Normal

[in] msg	Message
[in] len	Message length
[in] cflags	Message control flags
[in] ackid	Processing completion notification ID
[in] option	Notification option object – adaptor only

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Send message to destination. The function does not return until the requested message has been sent. A return value of zero does not mean delivery to the destination is complete. It just means that the message was successfully sent to the forwarder (core process). You must connect to the core process before calling cm_send.

The argument *xid* is the message tracking ID, if you have a received message and want to keep it the same as the received message, specify the received *xid* as it is. If *xid* is NULL, generate a new value.

The argument *src* is the message source, which must be a task or connection. If transmission to destination fails, it may be returned to source depending on ‘message control flags’.

The argument *dst* is the message destination, and the general form is "[ND].PG[[.PN][.SV[.CN[#PID.SN]]]]". If dst is NULL, it is send to the destination of the service to which the task or connection of the argument src belongs. For the destination of the service, see the **msg_dst** attribute of the service object.

The argument *mtype* is the message type, which can only be the constant CM_MSG_DATA.

The argument *stype* is the message subtype, which can only be the constant CM_MSGD_NORMAL

The arguments *msg* and *len* are the message to send and its length. The message length cannot exceed the length minus 'CubeMOM internal header size (512 bytes or less)' from 'maximum message allocation size (**msg_max_alloc_size**)'.

The argument *cflags* is a message control flags, which is a set of bit flags that specify whether to 'forward to the backup node' or 'return to the source' for each type of failure when an error occurs during delivery to the destination. 'forward to the backup node' has higher priority than 'return to the source'. If the *cflags* value is CM_CFLAGS_CONFIG, the message control flags value of the service to which the task or connection of the argument *src* belongs is used. Refer service object **msg_ctrl_flgs** attribute.

The argument *ackid* is a unique identifier for message processing completion notification. If *ackid* is not NULL, it must be the same as the *ackid* of the received message.

All messages received from the core process must be notified of processing completion within a certain period of time. Processes that do not notify processing completion within a certain period of time are considered abnormal and terminated forcibly.

The argument *option* is a notification option object, which is used to notify the object status along with sending messages to the core process. The notification options object is destroyed when cm_send or cm_notify is called. For adapter processes only, business processes must specify NULL.

See also

cm_recv, cm_notify, cm_notify_option

Example Code

```
#include <stdio.h>
#include <cubemom.h>

int main(int argc, char *argv[] ) {
	int		retval = 0 ;
	int		error  = 0 ;
	int		mtype = 0 ;
	int		stype  = 0 ;
	int		option = 0 ;
	int		psnum  = 0 ;
	int		recv_len = 0 ;
	void		*cmobj	= NULL ;
	char		*recv_msg = NULL ;
	void		*ackid	= NULL ;
	char		group [CM_NAME_MAX_LEN+1]; // group name
	char		myname[CM_SDST_MAX_LEN+1]; // group.process
	char		xid	[CM_XID_MAX_LEN +1];
	char		src	[CM_SDST_MAX_LEN+1];
	char		dst	[CM_SDST_MAX_LEN+1];

	cmobj = cm_initialize(argc, argv, &error );
	if( cmobj==NULL ) goto END;

	cm_get_myname(cmobj, NULL, group, &psnum);
	sprintf(myname, "%s.%d", group, psnum);

	if( retval = cm_connect(cmobj) ) {
		cm_log(cmobj, NULL, CM_CRITICAL, NULL, "%s cm_connect error(%d)", myname,
retval);
		goto END;
	}

	retval = cm_recv(cmobj, xid, src, dst, &mtype, &stype, &recv_msg, &recv_len, &option,
&ackid);
	if( retval ) {
		cm_log(cmobj, dst, CM_ERROR, NULL, "%s cm_recv error(%d)", myname, retval) ;
		goto END;
	}

	// do something ...

	retval = cm_send(cmobj, xid, dst, src, mtype, stype, recv_msg, recv_len,
CM_CFLAGS_CONFIG, &ackid, NULL);
	if( retval ) {
		cm_log(cmobj, dst, CM_ERROR, xid, "%s cm_send error(%d)", dst, retval) ;
		goto END;
	}

END :
	if(cmobj) {
		cm_close(cmobj);
		cm_terminate(cmobj);
	}
	return retval;
}
```

cm_recv

Syntax

```
int    cm_recv(void *cmobj, char *xid, char *src, char *dst,
                int *mtype, int *stype, char **msg, int *len,
                int *reason, void **ackid);
```

Parameters

[in] cmobj	Process object pointer
[out] xid	Message tracking ID
[out] src	Message source
[out] dst	Message destination
[out] mtype	Message type

CM_MSG_RETURN	Return message
CM_MSG_DATA	Data message
CM_MSG_RCMD	Command message

[out] stype	Message subtype

• CM_MSG_DATA

CM_MSGD_NORMAL	Normal data

• CM_MSG_RCMD

CM_MSGC_START	‘start’ command
CM_MSGC_STOP	‘stop’ command
CM_MSGC_DELIVER	‘deliver’ command

[out] msg	Message
[out] len	Message length
[out] reason	Return message : reason for return, Command message : Reason for object state change
[out] ackid	Processing completion notification ID

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Receive messages from the core process. The function does not return until one message is received. If there are no messages to receive, it waits until a message is received. You must connect to the core process before calling cm_recv.

The argument *src* is the task or connection of the sending process as the message source.

The argument *dst* is the message destination, that is, the object of the receiving process (“[ND].PG.PN[.SV[.CN[#PID.SN]]]”). For a data message, even if the sending process does not specify a task or connection as the destination, the receiving process receives a destination with a task or connection specified. The destination specified by the sending process can be obtained with the cm_get_dst function.

The arguments *msg* and *len* are the message received and its length. Note that the value of the *msg* pointer (received message) cannot be changed. If you want to change the message, you must do so in a separate copy.

The argument *ackid* is a unique identifier for message processing completion notification. All messages received from the core process must be notified of processing completion within a certain period of time. Processes that do not notify processing completion within a certain period of time are considered abnormal and terminated forcibly.

See also

cm_send, cm_notify, cm_notify_option

Example Code

```
#include <stdio.h>
#include <cubemom.h>

int main(int argc, char *argv[] ) {
	int		retval = 0 ;
	int		error  = 0 ;
	int		mtype  = 0 ;
	int		stype  = 0 ;
	int		option = 0 ;
	int		psnum  = 0 ;
	int		recv_len = 0 ;
	void	*cmobj	= NULL ;
	char	*recv_msg = NULL ;
	void	*ackid	= NULL ;
	char	group [CM_NAME_MAX_LEN+1]; // group name
	char	myname[CM_SDST_MAX_LEN+1]; // group.process
	char	xid	[CM_XID_MAX_LEN +1];
	char	src	[CM_SDST_MAX_LEN+1];
	char	dst	[CM_SDST_MAX_LEN+1];

	cmobj = cm_initialize(argc, argv, &error );
	if( cmobj==NULL ) goto END;

	cm_get_myname(cmobj, NULL, group, &psnum);
	sprintf(myname, "%s.%d", group, psnum);

	if( retval = cm_connect(cmobj) ) {
```

```
        cm_log(cmobj, NULL, CM_CRITICAL, NULL, "%s cm_connect error(%d)", myname, retval);
        goto END;
    }

    retval = cm_recv(cmobj, xid, src, dst, &mtype, &stype, &recv_msg, &recv_len, &option, &ackid);
    if( retval ) {
        cm_log(cmobj, dst, CM_ERROR, NULL, "%s cm_recv error(%d)", myname, retval) ;
        goto END;
    }

    // do something ...

    retval = cm_send(cmobj, xid, dst, src, mtype, stype, recv_msg, recv_len, CM_CFLAGS_CONFIG, &ackid, NULL);
    if( retval ) {
        cm_log(cmobj, dst, CM_ERROR, xid, "%s cm_send error(%d)", dst, retval) ;
        goto END;
    }

END :
    if(cmobj) {
        cm_close(cmobj);
        cm_terminate(cmobj);
    }
    return retval;
}
```

cm_notify_option

Syntax

```
void    *cm_notify_option(void *ntobj, int option, int optval1, void *optval2, int *error);
```

Parameters

[in] ntobj	Notification options object pointer
[in] option	Notification option code

CM_NOPT_RESPONSE_TIME	Response time
CM_NOPT_EXTRA_EVENT	Additional event
CM_NOPT_SRSN_REASON	Reason for change of status
CM_NOPT_CONN_ADDR	Connection local/remote address

[in] optval1	Notification option value
[in] optval2	The local/remote address of the connection if the notification option code is CM_NOPT_CONN_ADDR
[out] error	Integer pointer to store error code in case of error

Return value

Returns a notification options object pointer if there are no errors. Returns a NULL pointer on error and sets the error code to argument *error*.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Create a notification option object or add 'notification option code' to a previously created notification option object. The created notification options object is destroyed when calling the cm_send or cm_notify function.

The argument *ntobj* is a notification option object pointer. Specify NULL when creating a new notification option object, and specify the previously created notification option object when adding 'notification option code' to a previously created notification option object.

Note The cm_notify_option function is for adapter process only. Business process should not call it.

See also

cm_notify, cm_send, cm_recv

Example Code

```
    // code omission

    notiopt = cm_notify_option(NULL, CM_NOPT_EXTRA_EVENT,
CM_MSGE_DISCONNECTING, NULL, &error);
    notiopt = cm_notify_option(notiopt, CM_NOPT_SRSN_REASON,
CMR_SRSN_RECV_LIMIT , NULL, &error);

    // Send message with notifications
    // retval = cm_send(cmobj, xid, dst, src, mtype, stype, send_msg, send_len,
CM_CFLAGS_CONFIG, &ackid, notiopt);
```

cm_notify

Syntax

```
int    cm_notify(void *cmobj, const char *src, int mtype, int stype, void **ackid,
                 int error, void *option);
```

Parameters

[in] cmobj	Process object pointer
[in] src	Message source
[in] mtype	Message type

CM_MSG_RCMD	Command
CM_MSG_EVENT	Event
CM_MSG_ACK	Acknowledgement

[in] stype	Message subtype

• CM_MSG_RCMD

CM_MSGC_START	‘start’ command
CM_MSGC_STOP	‘stop’ command
CM_MSGC_DELIVER	‘deliver’ command

• CM_MSG_EVENT

CM_MSGE_STOPPED	Shutdown complete
CM_MSGE_STARTED	Startup complete
CM_MSGE_DISCONNECTED	Disconnect complete
CM_MSGE_CONNECTED	Connection complete
CM_MSGE_DISCONNECTING	Disconnection in progress
CM_MSGE_CONNECTING	Connecting in progress
CM_MSGE_WAITING	Waiting for connection
CM_MSGE_ABNORMAL	Abnormal
CM_MSGE_STOP_FAIL	Shutdown failure
CM_MSGE_START_FAIL	Startup failure
CM_MSGE_CONNECT_FAIL	Connection failure
CM_MSGE_SENT	Send complete
CM_MSGE_RCVD	Receive complete
CM_MSGE_SEND_FAIL	Send failure
CM_MSGE_RECV_FAIL	Receive failure
CM_MSGE_DONT_PS_RESTART	Request to stop process restart

• CM_MSG_ACK

CM_MSGA_DONE	Done

[in] ackid	Processing completion notification ID
[in] error	Notification error code
[in] option	Notification options object

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Notifies the core process of message processing completion or events. A return value of zero does not mean that the notification has not been reflected. It just means that the notification was successfully sent to the core process.

The argument *src* is the message source and the notification object. The message processing completion notification must be the same as the received message source, and the event notification is the object on which the event occurred.

The argument *ackid* is a unique identifier for message processing completion notification. All messages received from the core process must be notified of processing completion within a certain period of time. Processes that do not notify processing completion within a certain period of time are considered abnormal and terminated forcibly.

The argument *error* is the error code to notify the core process. Specify an error code in case of failure such as start/stop, send/receive, etc. 0 means normal.

The argument *option* is a notification option object, which is used to notify the status of the object along with message processing completion or event notification. The notification options object is destroyed when cm_send or cm_notify is called. For adapter processes only, business processes must specify NULL.

See also

cm_notify_option, cm_send, cm_recv

Example Code

```
	// code omission

	retval = cm_recv(cmobj, xid, src, dst, &mtype, &stype, &recv_msg, &recv_len, &option, &ackid);

	if( mtype == CM_MSG_RCMD    ) {
		// do something ...

		retval = cm_notify(cmobj, dst, CM_MSG_ACK, CM_MSGA_DONE, &ackid, 0, NULL);
	}
```

cm_get_param

Syntax

```
int    cm_get_param(void *cmobj, const char *name, char *buf);
```

Parameters

[in] cmobj	Process object pointer
[in] name	Parameter name
[out] buf	Parameter value

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns the parameter value. Parameters are designed to minimize hardcoding in CubeMOM applications. A parameter is a pair of key and value, and the parameter name corresponds to the key.

The argument *buf* is a buffer to store the parameter value, and its size must be larger than the constant value CM_MAX_PARAM_LEN.

See also

Example Code

```
// code omission

char        param [CM_MAX_PARAM_LEN+1] = { '\0', };

retval = cm_get_param(cmobj, "PA00", param);
cm_log(cmobj, NULL, CM_INFO, NULL, "retval(%d), PA00 : [%s] ", retval, param);
```

cm_split

Syntax

```
int    cm_split(const char *str, char *node, char *group, int *proc, char *svc, int *task, int *end);
```

Parameters

[in] str	Source or destination string
[out] node	Node name
[out] group	Group name
[out] proc	Process number
[out] svc	Service name
[out] task	Task or Connection number
[out] end	Last object in string

CM_ENDP_NODE	Node
CM_ENDP_GROUP	Group
CM_ENDP_PROCESS	Process
CM_ENDP_SERVICE	Service
CM_ENDP_TACONN	Task or Connection
CM_ENDP_SESSION	Session

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Splits the message source or destination string. If NULL is specified for an output argument, the corresponding value is not returned.

See also

cm_get_org, cm_get_dst, cm_get_fd

Example Code

```
    // code omission

    retval = cm_recv(cmobj, xid, src, dst, &mtype, &stype, &recv_msg, &recv_len, &option, &ackid);

    retval = cm_split(dst, NULL, NULL, NULL, NULL, NULL, &endp);
```

```
// do something ...
```

cm_get_org

Syntax

```
int    cm_get_org(void *ackid, char *org);
```

Parameters

[in] ackid	Processing completion notification ID
[out] org	Message originator

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns the message originator string. The general form of message originator string is “[ND].PG[[.PN][.SV[.CN[#PID.SN]]]]”.

The argument *ackid* is a unique identifier for notification of message processing completion, and is the value when a message is received.

The argument *org* is a buffer to store the originagor of the message, and its size must be greater than the constant value CM_SDST_MAX_LEN.

See also

cm_split, cm_get_dst, cm_get_fd

Example Code

```
    // code omission
    char        org     [CM_SDST_MAX_LEN+1];

    retval = cm_recv(cmobj, xid, src, dst, &mtype, &stype, &recv_msg, &recv_len, &option,
&ackid);

    retval = cm_get_org(ackid, org);
```

```
    // do something ...
```

cm_get_dst

Syntax

```
int    cm_get_dst(void *ackid, char *dst);
```

Parameters

[in] ackid	Processing completion notification ID
[out] dst	Message destination

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns the destination string specified by the message source process. The general form of message destination string is “[ND].PG[[.PN][.SV[.CN[#PID.SN]]]]”.

The argument *ackid* is a unique identifier for notification of message processing completion, and is the value when a message is received.

The argument *dst* is a buffer to store the destination of the message, and its size must be greater than the constant value CM_SDST_MAX_LEN.

Note In the case of a data message, the *dst* argument of the cm_recv function is assigned a task or connection by the core process, even if the source process does not specify the task or connection as the destination.

See also

cm_split, cm_get_dst, cm_get_fd

Example Code

```
    // code omission
char        sdst    [CM_SDST_MAX_LEN+1];
```

```
    retval = cm_recv(cmobj, xid, src, dst, &mtype, &stype, &recv_msg, &recv_len, &option, &ackid);

    retval = cm_get_dst(ackid, sdst);

    // do something ...
```

cm_get_fd

Syntax

```
int    cm_get_fd(void *cmobj);
```

Parameters

[in] cmobj	Process object pointer

Return value

Returns the file descriptor number if there is no error, or -1 if an error occurs.

Remarks

Returns the file descriptor connected to CubeMOM. You must connect to the core process before calling cm_get_fd.

The cm_get_fd function is for input/output multiplexing, and communication with CubeMOM must use the CubeMOM application API.

Note Do not change the file descriptor to asynchronous mode. The CubeMOM application API is designed to operate in synchronous mode.

See also

cm_split, cm_get_org, cm_get_dst

cm_get_strcode

Syntax

```
const char    *cm_get_strcode(int type, int code);
```

Parameters

[in] type — Code Type

CMR_OREC_TYPE	Object type
CMR_STAT_CODE	Object state
CMR_SRSN_CODE	Reason for object state change
CMR_SOCK_STAT	Socket state

[in] code — Code

• CMR_OREC_TYPE

CMR_TYPE_NDROOT	Node Root
CMR_TYPE_NODE	Node
CMR_TYPE_GRROOT	Group Root
CMR_TYPE_GROUP	Group
CMR_TYPE_PROCESS	Process
CMR_TYPE_SERVICE	Service
CMR_TYPE_TACONN	Connection

• CMR_STAT_CODE

CMR_STAT_STOPPED	Stopped
CMR_STAT_RUNNING	Running
CMR_STAT_STOPPING	Stopping
CMR_STAT_STARTING	Starting
CMR_STAT_ABNORMAL	Abnormal
CMR_STAT_DISABLED	Disabled

• CMR_SRSN_CODE

CMR_SRSN_CMAUTO	Automatic
CMR_SRSN_COMMAND	command
CMR_SRSN_PROCESS	process
CMR_SRSN_ERROR	Error
CMR_SRSN_PEER	Peer
CMR_SRSN_START_TIMEOUT	Start Timeout
CMR_SRSN_STOP_TIMEOUT	Stop Timeout
CMR_SRSN_ACK_TIMEOUT	Ack Timeout
CMR_SRSN_SEND_TIMEOUT	Send Timeout
CMR_SRSN_RECV_TIMEOUT	Recv Timeout
CMR_SRSN_IDLE_TIMEOUT	Idle Timeout
CMR_SRSN_SEND_LIMIT	Send Limit
CMR_SRSN_RECV_LIMIT	Recv Limit

• CMR_SOCK_STAT

CMR_SOCK_DISCONNECTED	Disconnected
CMR_SOCK_CONNECTED	Connected
CMR_SOCK_DISCONNECTING	Disconnecting
CMR_SOCK_CONNECTING	Connecting

Return value

Code value (string) if there is no error, returns NULL in case of error.

Remarks

Returns a string constant corresponding to the code.

See also

cm_get_errtxt

cm_get_errtxt

Syntax

```
int    cm_get_errtxt(int error, char *buf, int buf_len);
```

Parameters

[in] error	Error code
[out] buf	Error message buffer
[in] buf_len	Error message buffer size

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns the error message corresponding to the error code. The buffer size must be greater than or equal to the constant value CM_MAX_ERR_TEXT_LEN.

See also

cm_get_strcode

cm_is_loglevel

Syntax

```
int    cm_is_loglevel(void *cmobj, const char *name, int level, int *error);
```

Parameters

[in] cmobj	Process object pointer
[in] name	Log object
[in] level	Log level

CM_CRITICAL	Critical
CM_ERROR	Error
CM_WARNING	Warning
CM_INFO	Infomation
CM_VERBOSE	Verbose
CM_DEBUG	Debug
CM_TRACE	Trace

[out] error	Integer pointer to store error code in case of error

Return value

Returns 1 (active) or 0 (inactive) if there is no error, or -1 if an error occurs and sets the error code in the argument *error*.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns whether the specified log level is active.

The argument *name* is the log object in the form "[ND].PG[[.PN][.SV[.CN[#PID.SN]]]]".

See also

cm_log, cm_set_logbuf_size, cm_get_logbuf_size, cm_get_log_sem_key

Example Code

```
        // code omission

    retval = cm_recv(cmobj, xid, src, dst, &mtype, &stype, &recv_msg, &recv_len, &option,
&ackid);
```

```
onoff = cm_is_loglevel(cmobj, dst, CM_INFO, &error);
```

cm_log

Syntax

```
int    cm_log(void *cmobj, const char *name, int level, const char *xid, const char *format, ...);
```

Parameters

[in] cmobj	Process object pointer
[in] name	Log object
[in] level	Log level

CM_CRITICAL	Critical
CM_ERROR	Error
CM_WARNING	Warning
CM_INFO	Infomation
CM_VERBOSE	Information details
CM_DEBUG	Debug

[in] xid	Message tracking ID
[in] format	Variable argument format
[in] …	Variable argument

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Logs to the specified log level. If the specified log level is inactive, do not log and return normal (0).

Logs one line when calling a function. The log line consists of "log level", "time", "xid", "random string" columns, and "random string" is the format of the argument *format*. A newline character is always appended.

Logs are written line by line and are limited by the buffer size. The default size of the log buffer is 512 bytes, including newline and end-of-string characters. To log a string longer than the log buffer size, you must first increase the log buffer size with the cm_set_logbuf_size function.

The CubeMOM application uses one log file per process group. This can cause the log to be overwritten if multiple processes in the group are logging at the same time. It is recommended to set the group object **prc_log_key** attribute as a synchronization key (semaphore) to prevent log overwriting.

The argument *name* is the object string of the process to be logged ("[ND].PG[[.PN][.SV[.CN[#PID.SN]]]]"). If the argument *name* contains the name of service, the value of the **svc_loglevel** attribute of that service is used as the log level. If the argument *name* is NULL, the value of the **grp_loglevel** attribute of the process group is used as the log level.

Note The argument *name* specifies the service name if it is relevant to the service, or NULL if it is not relevant to the service

The argument *xid* is the message tracking ID. If it is NULL, the "xid" column is logged as blank.

See also

cm_is_loglevel, cm_set_logbuf_size, cm_get_logbuf_size, cm_get_log_sem_key

Example Code

```
    // code omission

    if( retval = cm_connect(cmobj) ) {
        cm_log(cmobj, NULL, CM_CRITICAL, NULL, "%s cm_connect error(%d)", myname, retval);
        goto END;
    }
```

cm_save

Syntax

```
int    cm_save(void *cmobj, const char *xid, const char *src, const char *dst,
                    const char *msg, int len);
```

Parameters

[in] cmobj	Process object pointer
[in] xid	Message tracking ID
[in] src	Message source
[in] dst	Message destination
[in] msg	Message
[in] len	Message length

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Encrypts the message (AES256) and saves it to a file. It is saved with a file name in the format “$NODE_$GROUP_$DATE.msg” in the path (“plog”) of the global property **plog_path** value. The encrypted message can be decrypted and viewed with the “**msgview**” command.

The adapter process encrypts and stores messages if tracing is enabled in the **svc_loglevel** property value.

See also

cm_is_loglevel, cm_log

cm_set_logbuf_size

Syntax

```
int    cm_set_logbuf_size(void *cmobj, int len);
```

Parameters

[in] cmobj	Process object pointer
[in] len	Log buffer length

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Set the log buffer size. Must be at least 512 bytes.

The log buffer is a buffer used by the cm_log function to create log string, and its size is the maximum length of the log string, including newline and end-of-string characters.

See also

cm_is_loglevel, cm_log, cm_get_logbuf_size, cm_get_log_sem_key

cm_get_logbuf_size

Syntax

```
int    cm_get_logbuf_size(void *cmobj);
```

Parameters

[in] cmobj	Process object pointer

Return value

Returns the log buffer size.

Remarks

The log buffer is a buffer used by the cm_log function to create log string, and its size is the maximum length of the log string, including newline and end-of-string characters.

See also

cm_is_loglevel, cm_log, cm_set_logbuf_size, cm_get_log_sem_key

cm_get_log_sem_key

Syntax

```
int    cm_get_log_sem_key(void *cmobj, char *key);
```

Parameters

[in] cmobj	Process object pointer
[out] key	Log synchronization key (semaphore) buffer

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns the log synchronization key (semaphore). The value of the process group object **prc_log_key** attribute.

The argument *key* is a buffer to store the log synchronization key (semaphore), and its size must be larger than the constant value CM_KEY_MAX_LEN.

See also

cm_is_loglevel, cm_log, cm_set_logbuf_size, cm_get_logbuf_size

cm_get_log_path

Syntax

```
int    cm_get_log_path(void *cmobj, int logdiv, char *pathbuf, int bufsize);
```

Parameters

[in] cmobj	Process object pointer
[in] logdiv	Path Type

CM_PROC_LOG_PATH	Business/Adapter process log path
CM_CORE_LOG_PATH	Core process log path

[out] pathbuf	Path buffer
[in] bufsize	Path buffer size

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns the process log path. The value of the global attribute **plog_path** for business/adapter processes and the global attribute **clog_path** for core processes.

The argument *pathbuf* is a buffer to store the log path, and its size must be larger than the constant value CM_PATH_MAX_LEN.

cm_fdump_open

Syntax

```
FILE   *cm_fdump_open(void *cmobj, int *error);
```

Parameters

[in] cmobj	Process object pointer
[out] error	Integer pointer to store error code in case of error

Return value

Returns a file pointer if there are no errors. Returns a NULL pointer on error and sets the error code to argument *error*.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Open the dump file. Create the file if it doesn't exist and open it in append mode if it already exists.

Note Dump files can be logged in free format for debug purposes. It is one per process group and has the extension “dmp”.

See also

cm_fdump_close, cm_fdump_printf, cm_fdump_lock, cm_fdump_unlock

cm_fdump_close

Syntax

```
int    cm_fdump_close(void *cmobj, FILE *fp);
```

Parameters

[in] cmobj	Process object pointer
[in] fp	File pointer

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Close the dump file.

See also

cm_fdump_open, cm_fdump_printf, cm_fdump_lock, cm_fdump_unlock

cm_fdump_printf

Syntax

```
int    cm_fdump_printf(void *cmobj, FILE *fp, int time, const char *format, ...);
```

Parameters

[in] cmobj	Process object pointer
[in] fp	File pointer
[in] time	Time logging or not
[in] format	Variable argument format
[in] …	Variable argument

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Logs to a dump file in the specified format. Newline characters are not added automatically.

Dump file is one per process group. This can be overwritten if multiple processes in the group dump simultaneously. You must synchronize using the dump synchronization function (cm_fdump_lock, cm_fdump_unlock).

It is recommended to set the group object **prc_log_key** attribute as a synchronization key (semaphore) to prevent log overwriting.

The argument *time* is whether to log the time. If it is a value other than 0, the current time (“---mmddss.mmm.uuu---“) is logged and then logged in the specified format.

See also

cm_fdump_open, cm_fdump_close, cm_fdump_lock, cm_fdump_unlock

Example Code

```
	// code omission

	FILE *dump = cm_fdump_open(cmobj, &retval);
	if( dump==NULL ) {
		cm_log(cmobj, NULL, CM_ERROR, NULL, "%s dump file open error(%d)", myname, retval);
		goto END ;
	}

	cm_fdump_lock(cmobj);
	cm_fdump_printf(cmobj, dump, 1, "%s test\n", myname);
	cm_fdump_printf(cmobj, dump, 0, "..%d\n", 1);
	cm_fdump_printf(cmobj, dump, 0, "..%d\n", 2);
	cm_fdump_unlock(cmobj);

	// do something ...

END :
	if(dump) cm_fdump_close(cmobj, dump);
```

cm_fdump_lock

Syntax

```
int    cm_fdump_lock(void *cmobj);
```

Parameters

[in] cmobj	Process object pointer

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Lock the log synchronization object. If the value of the group object **prc_log_key** attribute is not set, log synchronization between processes does not work.

See also

cm_fdump_open, cm_fdump_printf, cm_fdump_unlock

cm_fdump_unlock

Syntax

```
int    cm_fdump_unlock(void *cmobj);
```

Parameters

[in] cmobj	Process object pointer

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Unlock the log synchronization object. If the value of the group object **prc_log_key** attribute is not set, log synchronization between processes does not work.

See also

cm_fdump_open, cm_fdump_printf, cm_fdump_lock

cm_get_shmps_addr

Syntax

```
int    cm_get_shmps_addr(void *cmobj, void **ndobj, void **pgaddr, void **psaddr);
```

Parameters

[in] cmobj	Process object pointer
[out] ndobj	Node shared memory object
[out] pgaddr	Group record
[out] psaddr	Process record

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns node shared memory objects and the shared memory addresses of group and process records.

Note The cm_get_shmps_addr function is for adapter process only.

See also

cm_get_shmsv_addr, cm_get_shmad_addr

cm_get_shmsv_addr

Syntax

```
int    cm_get_shmsv_addr(void *cmobj, void **svaddr, int *svcount);
```

Parameters

[in] cmobj	Process object pointer
[out] svaddr	Service record address
[out] svcount	Number of service records

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns the shared memory address and count of the current process, service record.

Note The cm_get_shmsv_addr function is for adapter process only.

See also

cm_get_shmps_addr, cm_get_shmad_addr

cm_get_shmad_addr

Syntax

```
int    cm_get_shmad_addr (void *cmobj, const char *name, void **adaddr);
```

Parameters

[in] cmobj	Process object pointer
[in] name	Service name
[out] adaddrr	Address record

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns the address of the ‘address’ record shared memory.

Note The cm_get_shmad_addr function is for adapter process only.

See also

cm_get_shmps_addr, cm_get_shmsv_addr

cm_get_xid

Syntax

```
int    cm_get_xid(void *cmobj, char *xid);
```

Parameters

[in] cmobj	Process object pointer
[out] xid	Message tracking ID

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns the new message tracking ID. When the cm_get_xid function is called, the serial number of the message tracking ID is incremented. The serial number is a 32-bit unsigned integer that restarts at 0 when the maximum value is reached.

The argument *xid* is the message tracking ID storage buffer, and its size must be larger than the constant value CM_XID_MAX_LEN.

cm_get_session

Syntax

```
int    cm_get_session(void *cmobj);
```

Parameters

[in] cmobj	Process object pointer

Return value

Returns the session number if there is no error, or -1 in case of an error.

Remarks

Returns the available session number. Session number must be cleared.

Note The cm_get_session function is for adapter process only.

See also

cm_clr_session

cm_clr_session

Syntax

```
int    cm_clr_session(void *cmobj, int session);
```

Parameters

[in] cmobj	Process object pointer
[in] session	Session number

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Clear the session number.

Note The cm_clr_session function is for adapter process only.

See also

cm_get_session

cm_get_myname

Syntax

```
int    cm_get_myname (void *cmobj, char *nd, char *pg, int *ps);
```

Parameters

[in] cmobj	Process object pointer
[out] nd	Node name buffer pointer
[out] pg	Group name buffer pointer
[out] ps	Process number pointer

Return value

Returns 0 if no error, error code in case of error.

Error code > 0	Refer OS Error Code
Error code < 0	Refer Error Code

Remarks

Returns the node name, group name, and process number of the current process. If NULL is specified for an output argument, the corresponding value is not returned.

The arguments *nd* and *pg* are name storage buffers, the size of which must be larger than the constant value CM_NAME_MAX_LEN.

See also

cm_get_mypid

cm_get_mypid

Syntax

```
int    cm_get_mypid(void *cmobj);
```

Parameters

[in] cmobj	Process object pointer

Return value

Returns the process ID if there is no error, or -1 on error.

Remarks

Returns the process ID.

See also

cm_get_myname

Link API

The Link API is a set of functions for interfacing legacy system with CubeMOM. The legacy system must communicate with the CubeMOM's 'link adapter' using the Link API.

cmi_ps_create

Syntax

```
void   *cmi_ps_create(int flags);
```

Parameters

[in] flags	Object creation option flags	
	CMI_TLS	Transport Layer Security

Return value

Returns the process object pointer if there are no errors. If an error occurs, return a NULL pointer and set the error code in errno.

Remarks

Creates and initializes a process object. A process object is the parent object of a service object. If the process object includes the 'Transport Layer Security' service object, the constant value CMI_TLS must be specified in the argument *flags*.

Note Only one process object should be created.

See also

cmi_ps_destroy

cmi_ps_destroy

Syntax

```
int    cmi_ps_destroy(void *psobj);
```

Parameters

[in]	psobj	Process object pointer

Return value

Returns 0 if there is no error, or an error code in case of an error.

Remarks

Destroy the process object.

See also

cmi_ps_create

cmi_sv_create

Syntax

```
void    *cmi_sv_create(void *psobj, int csmode, const char *addr, int port, int flags);
```

Parameters

[in] psobj	Process object pointer
[in] csmode	TCP client or server

CMA_CS_CLIENT	Client
CMA_CS_SERVER	Server

[in] addr	Address (IP/Host Name/Domain Name) string
[in] port	Port
[in] flags	Object creation options

CMI_ASYNC	Asynchronous communication
CMI_TLS	Transport Layer Security
CMI_TLS_VERIFY_PEER	Transport Layer Security, Peer authentication

Return value

Returns the service object pointer if there are no errors. Returns a NULL pointer on error and sets the error code in errno.

Remarks

Create and initialize service object. The service object is the parent object of the connection object.

See also

cmi_sv_destroy

cmi_sv_destroy

Syntax

```
int    cmi_sv_destroy(void *svobj);
```

Parameters

[in] svobj	Service object pointer

Return value

Returns 0 if there is no error, or an error code in case of an error.

Remarks

Destroy the service object.

See also

cmi_sv_create

cmi_set_sockopt

Syntax

```
int    cmi_set_sockopt(void *svobj, int sockopt, int val1, int val2);
```

Parameters

[in] svobj	Service object pointer
[in] nsockopt	Socket option

CMA_SO_KEEPALIVE	SO_KEEPALIVE
CMA_SO_RCVBUF	SO_RCVBUF
CMA_SO_SNDBUF	SO_SNDBUF
CMA_SO_REUSEADDR	SO_REUSEADDR
CMA_SO_REUSEPORT	SO_REUSEPORT
CMA_SO_LINGER	SO_LINGER
CMA_TCP_NODELAY	TCP_NODELAY

[in] val1	Option value 1
[in] val2	Option value 2

Return value

Returns 0 if there are no errors. Returns -1 on error and sets the error code in errno.

Remarks

Set service object socket options. Service object socket options apply to newly created connection objects.

For the CMA_SO_KEEPALIVE, CMA_SO_REUSEADDR, CMA_SO_REUSEPORT, and CMA_TCP_NODELAY options, if the value of the argument *val1* is not 0, it is considered true, and if it is 0, it is considered false, and the value of the argument *val2* is ignored.

The CMA_SO_RCVBUF and CMA_SO_SNDBUF options use the value of the argument *val1* and ignore the value of *val2*.

The CMA_SO_LINGER option considers true if the value of the argument *val1* is not 0, false if the value is 0, and uses the value of *val2* as the SO_LINGER value if the value of the argument *val1* is true.

cmi_load_certificate

Syntax

```
int    cmi_load_certificate(void *svobj, const char *certf,
                            const char *prikeyf,    const char *prikeypw, const char *trustcaf);
```

Parameters

[in] svobj	Service object pointer
[in] certf	Certificate (chain) file name
[in] prikeyf	Private key file name
[in] prikeypw	Private key password
[in] trustcaf	Trust CA file name

Return value

Returns 0 if there are no errors. Returns -1 on error and sets the error code in errno.

Remarks

Load the certificate. Service objects must be objects created with the transport layer security option (CMI_TLS).

The arguments *prikeyf*, *prikeypw*, *trustcaf* specify NULL if they do not exist.

See also

cmi_set_certificate_nid_commonName

cmi_set_certificate_nid_commonName

Syntax

```
int cmi_set_certificate_nid_commonName(void *svobj, const char *name);
```

Parameters

[in] svobj	Service object pointer
[in] name	'NID common name' of peer certificate

Return value

Returns 0 if there are no errors. Returns -1 on error and sets the error code in errno.

Remarks

Set 'NID common name' of peer certificate. The set value is used to check whether it is the same when authenticating the other party. Service objects must be objects created with the transport layer security option (CMI_TLS).

See also

cmi_load_certificate

cmi_listen

Syntax

```
int    cmi_listen(void *svobj, int backlog);
```

Parameters

[in] svobj	Service object pointer
[in] backlog	Connection queue length

Return value

Returns 0 if there are no errors. Returns -1 on error and sets the error code in errno.

Remarks

It is a function that wraps the BSD socket API listen function and has the same role.

See also

cmi_accept, cmi_connect, cmi_connect_result, cmi_tls_handshake, cmi_close

cmi_accept

Syntax

```
void    *cmi_accept(void *svobj);
```

Parameters

[in] svobj	Service object pointer

Return value

Returns a connection object pointer if there are no errors. Returns NULL on error and sets the error code in errno.

Remarks

It is a function that wraps the BSD socket API accept function and has the same role.

See also

cmi_listen, cmi_connect, cmi_connect_result, cmi_tls_handshake, cmi_close

cmi_connect

Syntax

```
void    *cmi_connect(void *svobj, int *rlt);
```

Parameters

[in] svobj	Service object pointer
[out] rlt	Connection result

CMI_PROCEEDING	Proceeding
CMI_COMPLETION	Connected

Return value

Returns a connection object pointer if there are no errors. Returns NULL on error and sets the error code in errno.

Remarks

It is a function that wraps the BSD socket API connect function and has the same role.

The argument *rlt* gives the status of an asynchronous socket connection as a result of the connection attempt.

See also

cmi_listen, cmi_accept, cmi_connect_result, cmi_tls_handshake, cmi_close

cmi_connect_result

Syntax

```
int    cmi_connect_result(void *cnobj, int *rlt);
```

Parameters

[in] cnobj	Connection object pointer
[out] rlt	Connection result

CMI_PROCEEDING	Proceeding
CMI_COMPLETION	Connected

Return value

Returns 0 if there are no errors. Returns -1 on error and sets the error code in errno.

Remarks

Returns the result of an asynchronous socket connection attempt.

The argument *rlt* gives the status of an asynchronous socket connection as a result of the connection attempt.

See also

cmi_listen, cmi_accept, cmi_connect, cmi_tls_handshake, cmi_close

cmi_tls_handshake

Syntax

```
int    cmi_tls_handshake(void *cnobj, int *rlt, int *req);
```

Parameters

[in] cnobj	Connection object pointer
[out] rlt	TLS handshake result

CMI_PROCEEDING	Proceeding
CMI_COMPLETION	Complete

[out] req	TLS library's request action

CMI_TLS_WANT_READ	Read
CMI_TLS_WANT_WRITE	Write

Return value

Returns 0 if there are no errors. Returns -1 on error and sets the error code in errno.

Remarks

Proceed with the TLS(Transport Layer Security) handshake.

The argument *rlt* is the asynchronous socket TLS handshake result.

The argument *req* is the TLS library's request action.

See also

cmi_listen, cmi_accept, cmi_connect, cmi_connect_result, cmi_close

cmi_close

Syntax

```
int    cmi_close(void *cnobj);
```

Parameters

[in] cnobj	Connection object pointer

Return value

Returns 0 if there is no error, or an error code in case of an error.

Remarks

Closes the connection's socket and destroys the connection object.

See also

cmi_listen, cmi_accept, cmi_connect, cmi_connect_result, cmi_tls_handshake

cmi_get_svfd

Syntax

```
int    cmi_get_svfd(void *svobj);
```

Parameters

[in] svobj	Service object pointer

Return value

Returns the file descriptor if there are no errors, or a negative integer in case of errors.

Remarks

Return the file descriptor of the service object.

See also

cmi_get_cnfd, cmi_get_cnstat

cmi_get_cnfd

Syntax

```
int    cmi_get_cnfd(void *cnobj);
```

Parameters

[in] cnobj	Connection object pointer

Return value

Returns the file descriptor if there are no errors, or a negative integer in case of errors.

Remarks

Returns the file descriptor of the connection object.

See also

cmi_get_svfd, cmi_get_cnstat

cmi_get_cnstat

Syntax

```
int    cmi_get_cnstat(void *cnobj);
```

Parameters

[in] cnobj	Connection object pointer

Return value

Returns the connection status value.

Status value	Connection status
0	Not connected
1	Connected, TLS handshake not complete
2	Connected, TLS handshake complete

Remarks

Returns the connection status value.

See also

cmi_get_svfd, cmi_get_cnfd

cmi_send

Syntax

```
int    cmi_send(void *cnobj, const char *xid, const char *dst, const char *msg, int len);
```

Parameters

[in] cnobj	Connection object pointer
[in] xid	Message tracking ID
[in] dst	Message destination
[in] msg	Message
[in] len	Message length

Return value

Returns the bytes sent if there are no errors. Returns -1 on error and sets the error code in errno.

Remarks

Sends a message to the connection object's socket. Success as a function execution result does not mean completion of delivery to peer. It just means that the message was sent successfully.

The argument *dst* is the message destination, and the general form is “[ND].PG[[.PN][.SV[.CN[#PID.SN]]]]”.

See also

cmi_recv, cmi_free

cmi_recv

Syntax

```
int    cmi_recv(void *cnobj, char *xid, char *src, char **msg, int *pending);
```

Parameters

[in] cnobj	Connection object pointer
[out] xid	Message tracking ID
[out] src	Message source
[out] msg	Message
[out] pending	In case of TLS, whether additional data to be read exists

Return value

Returns the bytes received if there are no errors. Returns -1 on error and sets the error code in errno. Returns 0 if the peer terminated the connection normally.

Remarks

Receive messages from the connection object's socket.

The argument *src* is the task or connection of the sending process as the message source.

The argument *msg* is the message received. It is dynamically allocated memory that must be freed (cmi_free).

See also

cmi_send, cmi_free

cmi_free

Syntax

```
void    cmi_free(void *msg);
```

Parameters

[in] msg	Dynamically allocated memory pointer

Return value

No returns.

Remarks

Free dynamically allocated memory.

See also

cmi_send, cmi_recv