Dept. of Smart ICT Convergence, Konkuk University

Communication Framework (CM) User Guide

CM v1.9.8

Collaborative Computing Systems Lab.

2020-1-20

Contents

Preface

This document is a user guide of the communication framework (CM) developed with Java. CM is developed to help an application developer easily build a networked application by providing various communication-related functionalities. For more detailed information of CM, please refer to a recent technical report [1].

This document guides how to develop a client-server system with example codes using CM. Therefore, we need at least two applications, a desktop server and a client. For the clarity of the guide, CM relevant codes have been tested in the following implementation environment.

- JDK version: Oracle JDK 8 and 10 or Azul Zulu OpenJDK 1.8.0_181 (probably compatible with higher version of JDK too)
- Eclipse IDE for Java Developers: Photon Release (4.8.0) (probably compatible with higher version of Eclipse IDE too)
- CM library: CM.jar and configuration files (*.conf files)
- External library: MySQL (mysql-connector-java-5.1.31-bin.jar)

CM project files (library, source codes, sample client and server applications) and documentation (quick start guide, user guide and API documents) are available in the following URL:

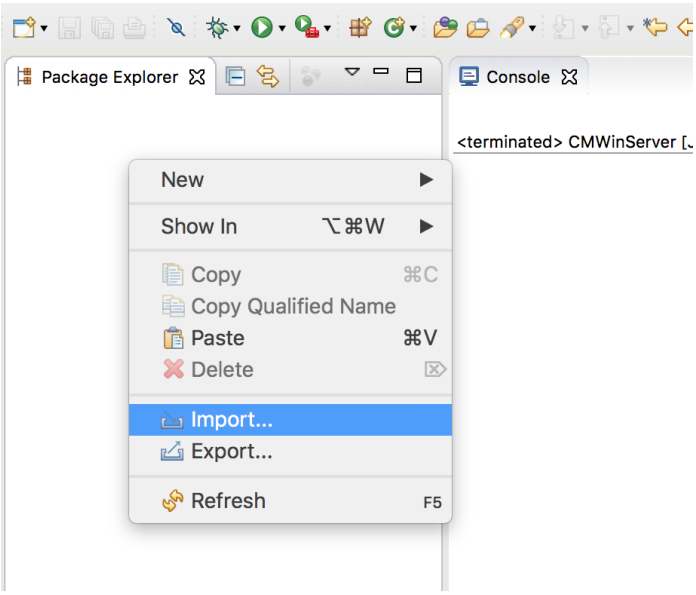
https://sites.google.com/site/kuccslab/research/cm

1. Download of the CM library and sample client/server applications

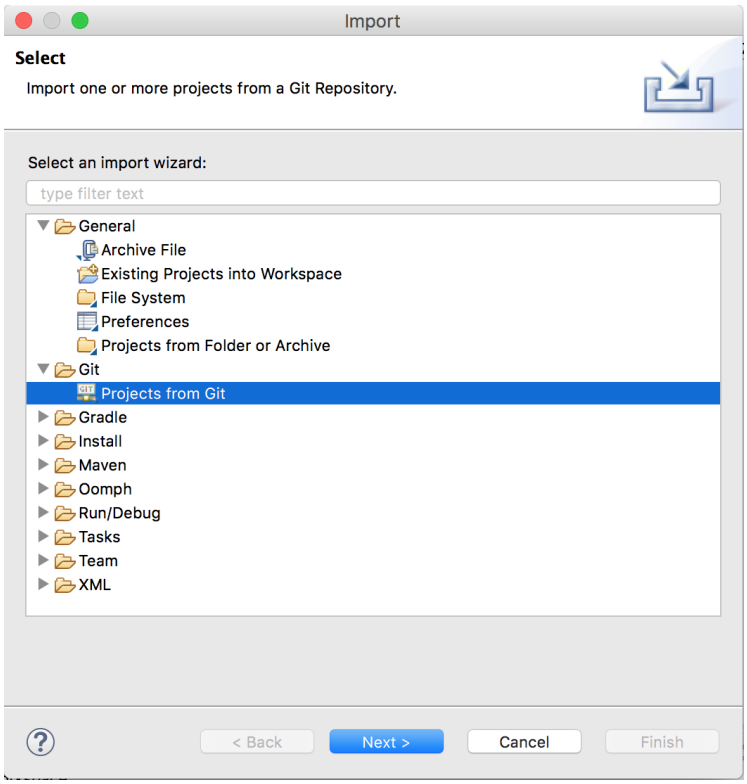
For testing and using CM functionalities, we provide required libraries (CM.jar and others), CM configuration files, and sample client and server applications as a single project through GitHub. Users can download the sample project from the URL (https://github.com/ccslab/CMTest.git). The CM source codes also can be downloaded from the other URL (https://github.com/ccslab/CM.git). Users can use a Git program to download the CM project from the remote Git repository and import it in Eclipse, or can directly use the EGit plugin in Eclipse to download and import the CM project from the remote Git repository. For the latter case, follow the steps as described below:

1. Run Eclipse and select a workspace directory.

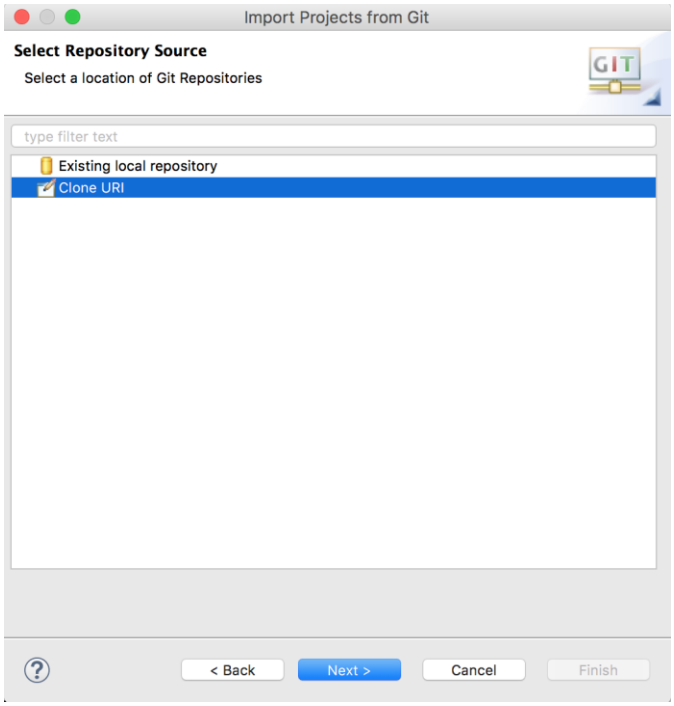
2. Mouse right click in the package explorer view and select the “import” menu.



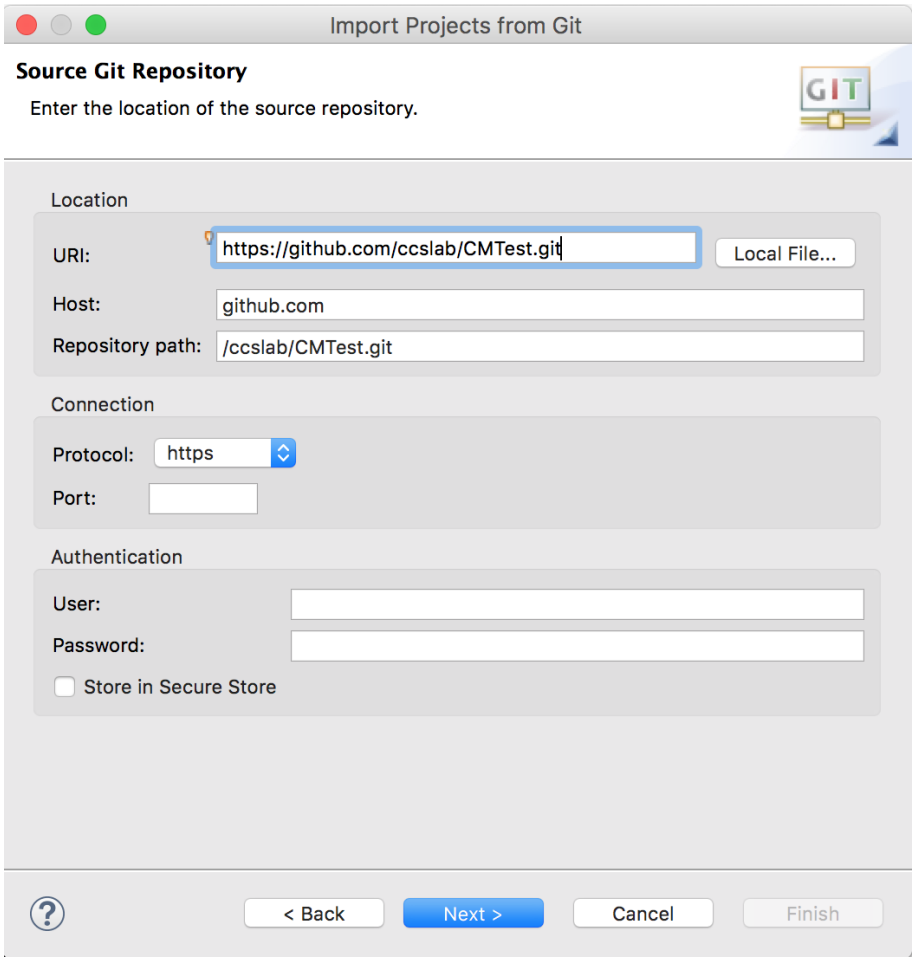
3. Select the “Projects from Git” menu in the import wizard window and click the “Next” button.



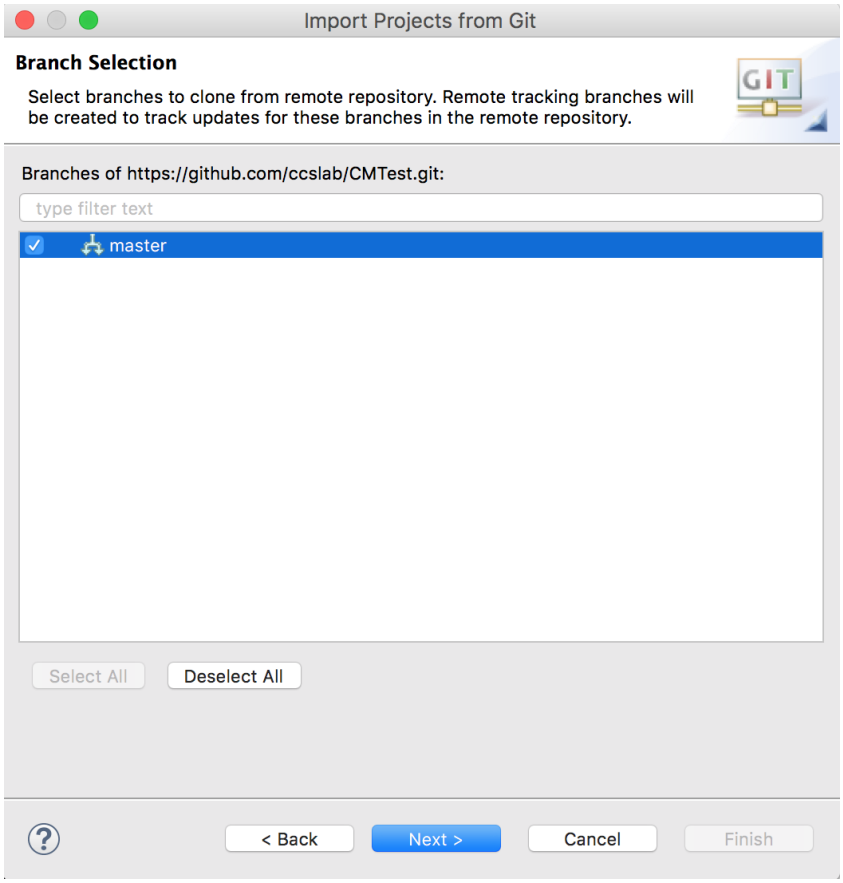
4. Select the “Clone URI” menu and click the “Next” button.



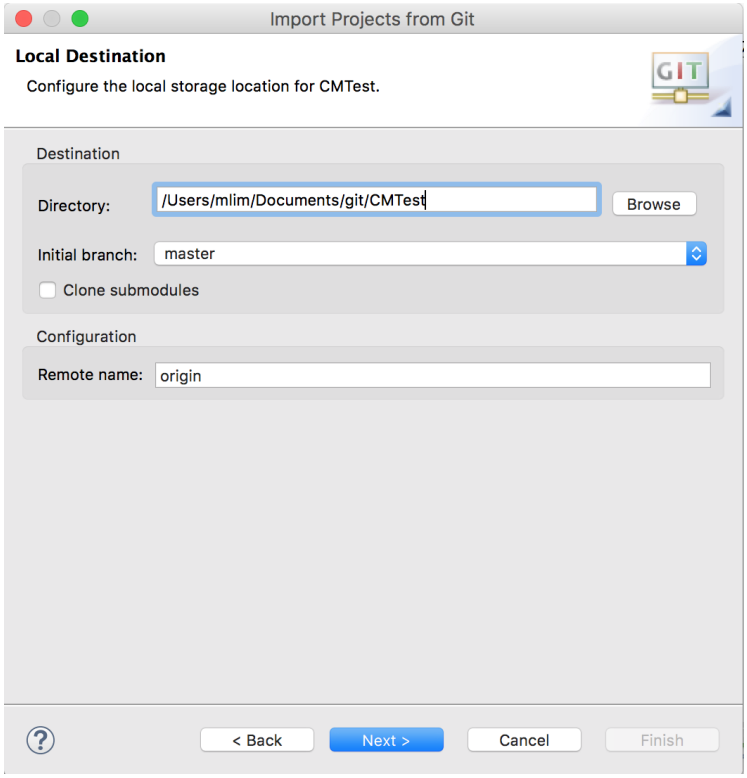
5. In the URI field, fill in the project URL (https://github.com/ccslab/CMTest.git) and click the "Next" button.



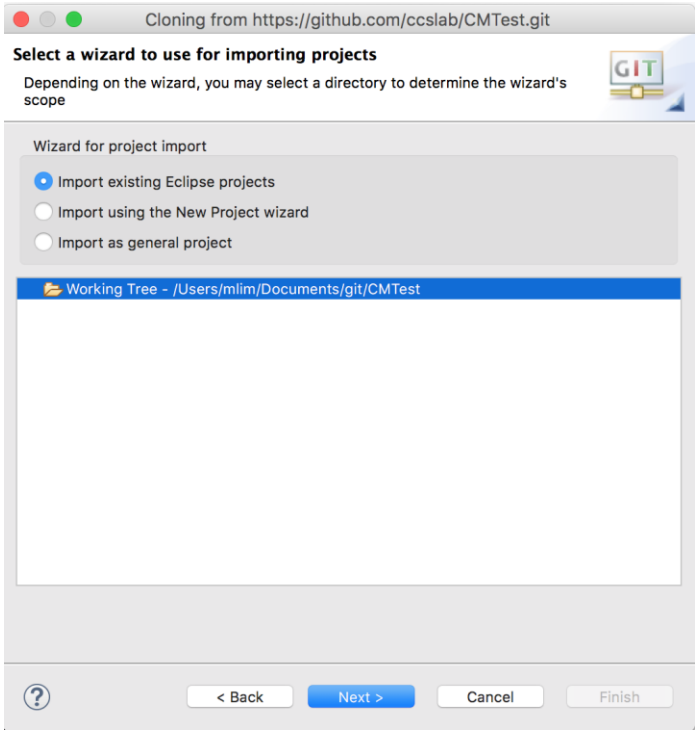
6. Select the master branch and click the "Next" button.



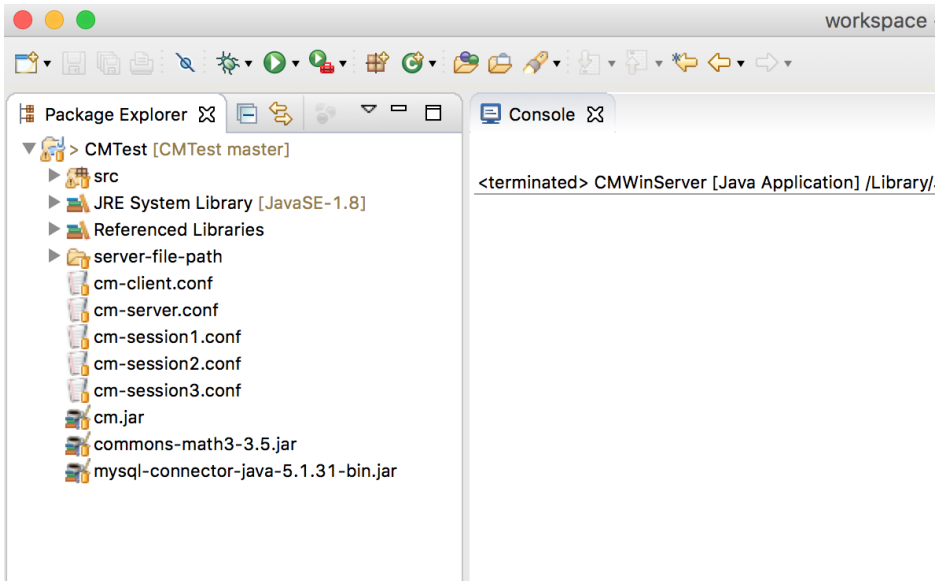
7. Fill in the destination directory where the project will be downloaded.



8. After the completion of downloading the project, select the "Import existing Eclipse projects", and click the "Next" button.



9. Select the CMTest project, and click the "Finish" button. After the successful import process, the package explorer view of Eclipse looks like below.



In the CMTest project directory, there are three jar files that are required for developing and running

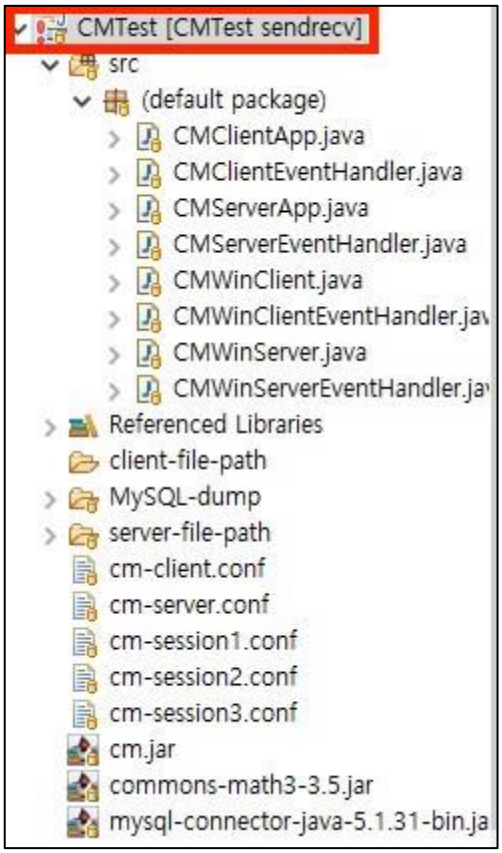
CM applications. "cm.jar" is the CM library file, and the other two jar files are external libraries for the MySQL Connector (mysql-connector-java-5.1.31-bin.jar) and the mathematics and statistics library (commons-math3-3.5.jar).

There are also several "xxx.conf" files in the project directory. These are CM configuration files that need to be set before running the sample client and server applications according to the user's preferences. The details of the configuration files are described in the following section.

The CMTest project also includes sample client and server application codes using the CM libraries. Sources codes are in the "src" directory. The sample client and server applications are divided into window-based (CMWinClient.java and CMWinServer.java) and text-based applications (CMClientApp.java and CMServerApp.java). For the simplicity of the codes, this document describes the CM functionalities and example codes based on the text-based version.

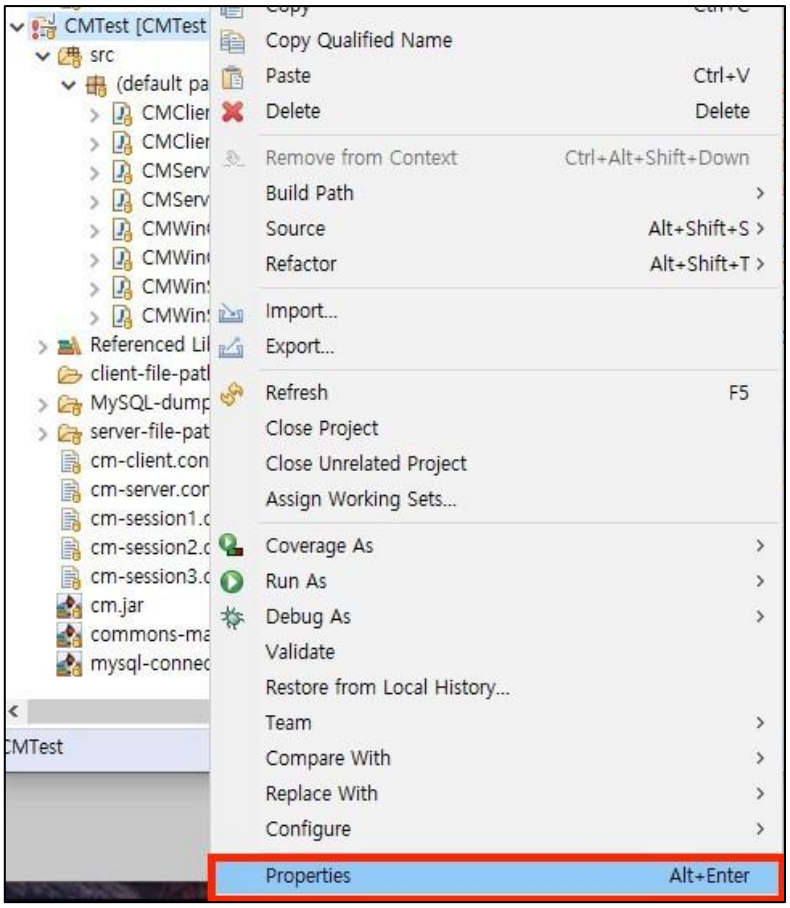
1.1 In case of build error due to JRE problem

If your JRE version or platform is not compatible with that of the downloaded project, Eclipse fails to build the project. For example, Eclipse shows an error in the package explorer as the following figure.

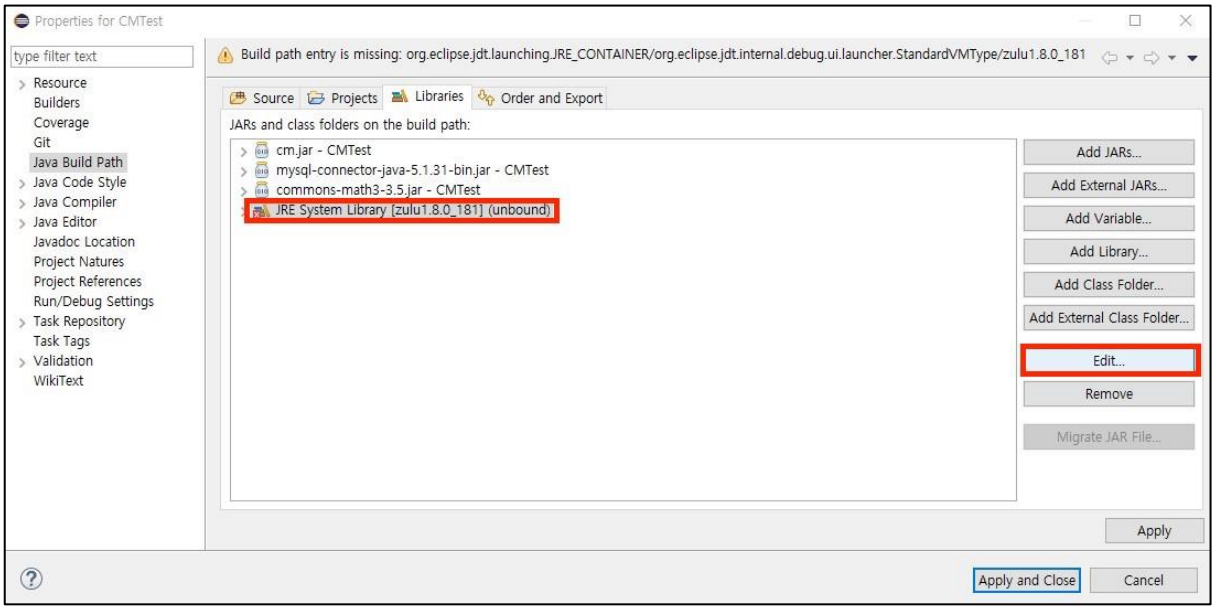


To solve this error, you should set your JRE to the project as following steps.

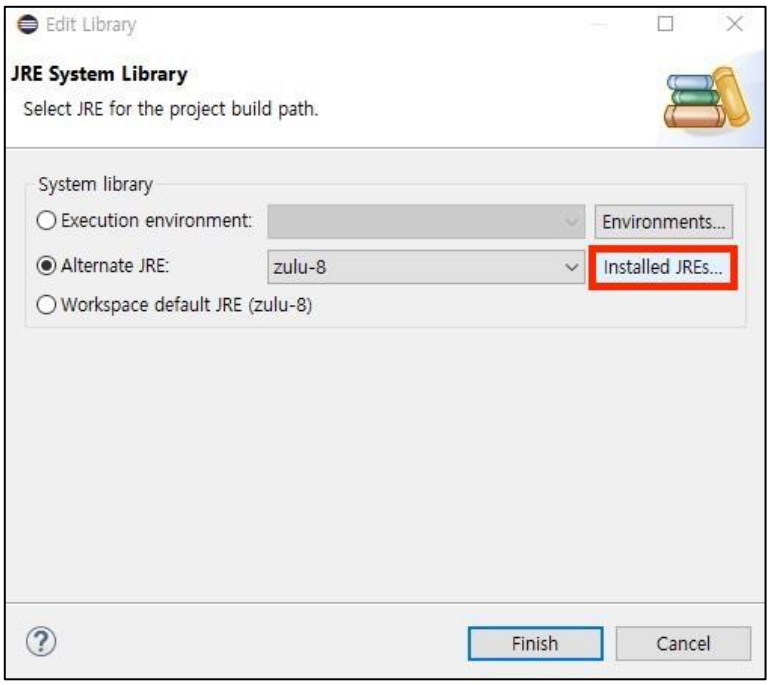
1. Click right mouse button on the project name in the package explorer to open the context menu, and select the "Properties" menu.



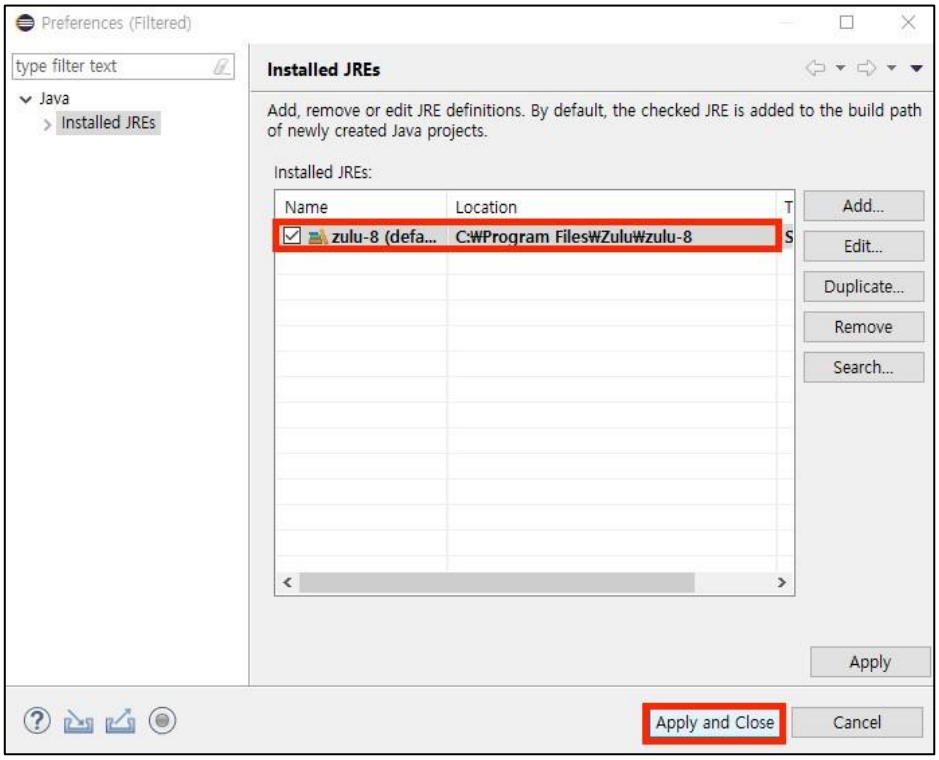
2. In the properties page, select the "Java Build Path" menu. In the "Libraries" tab, select the problematic JRE library, and press the "Edit..." button.



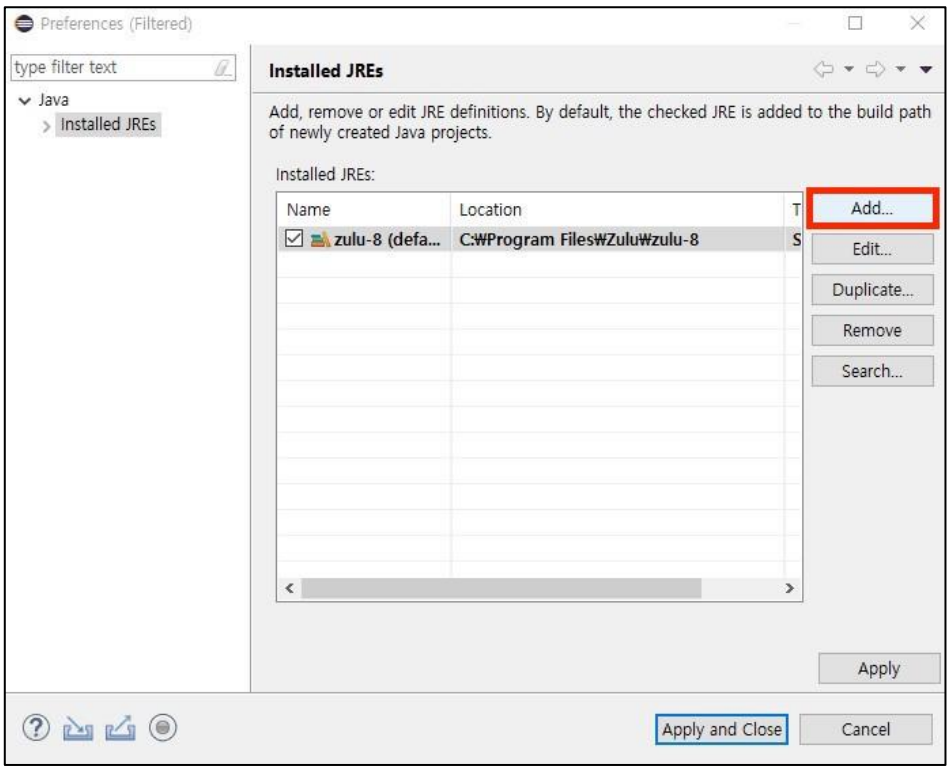
3. In the JRE System Library page, press the "Installed JREs..." button.



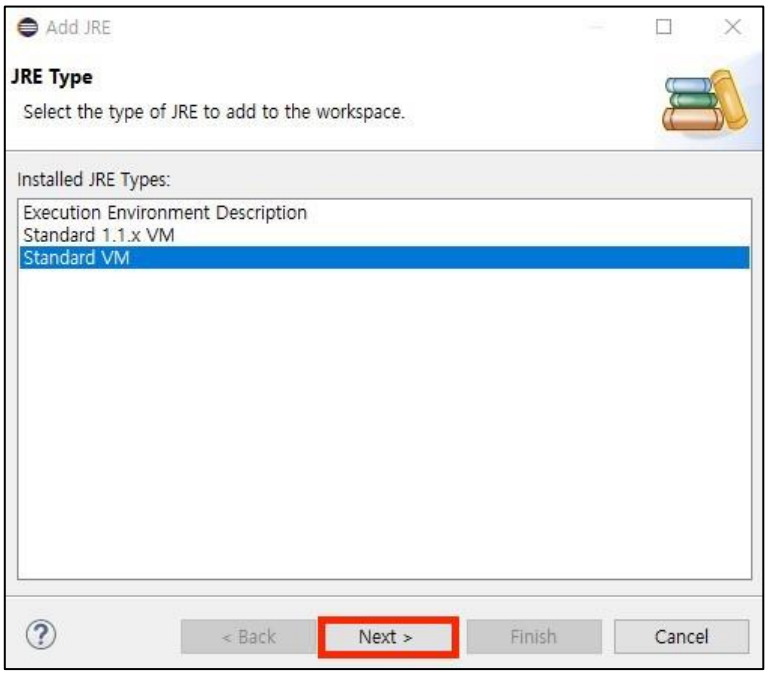
4. In the Installed JREs page, select your version of JRE, and press the "Apply and Close" button.



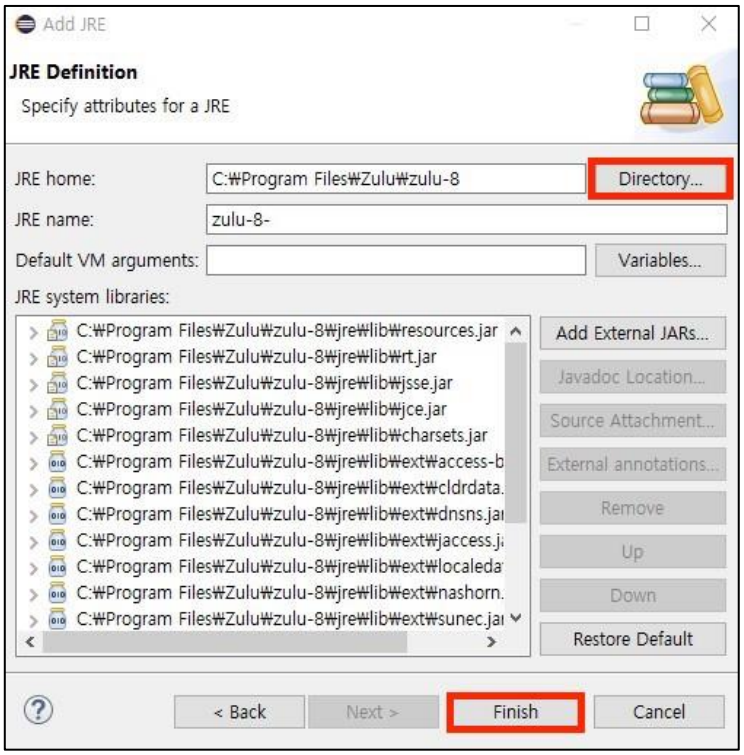
5. If your JRE is not in the list of installed JREs, you should add it manually. To this end, press the "Add..." button.



6. In the JRE type page, select the "Standard VM" and press the "Next" button.



7. In the JRE definition page, press the "Directory" button and choose the root directory where the JRE has been installed. Press the "Finish" button and go back to step 4 to select the added JRE.



2. CM configuration files

After locating the configuration files in an appropriate directory, we have to configure default values of the files.

2.1 CM server configuration file

A server application must set the CM server configuration files. A main file is *cm-server.conf* file. This is a simple text file and can be open by any text editor. The following figure is an example of this file.

```
########## CM fundamental configuration

# system type (SERVER/CLIENT)
SYS_TYPE        SERVER

# communication architecture (CM_CS/CM_PS)
COMM_ARCH       CM_CS

# default server configuration
SERVER_ADDR     localhost
#SERVER_ADDR    192.168.0.17
SERVER_PORT     7777

# my configuration (this server)
MY_PORT 7777

# default udp configuration
UDP_PORT 8888

# default multicast configuration
MULTICAST_ADDR  224.1.1.1
MULTICAST_PORT  7000

########## Options related to the file transfer

# default directory where a server or a client searches for a requested file
# default directory where a client maintains a downloaded file from a server
FILE_PATH       ./server-file-path

# file transfer scheme (true / false)
# true: file transfer with the additional channel and thread
# false: file transfer with the default channel and with no additional thread
FILE_TRANSFER_SCHEME    1

# appended file reception scheme (true / false)
# true: file reception of only the remaining file blocks (append mode)
# false: file reception from the first file block (overwrite mode)
FILE_APPEND_SCHEME      0

########## CM login/session mechanism parameters

# login scheme (true / false)
# true: application is responsible for user authentication
# false: no authentication
LOGIN_SCHEME    0

# session scheme (true / false)
# true: use multiple sessions and regions
# false: no session and region (use a default session and region)
SESSION_SCHEME  0
```

```
########## CM DB configuration

# DB usage (true/ false)
# true: use DB (MySQL) internally for user registration and content management
# false: do not use DB
DB_USE          1
DB_HOST         localhost
DB_USER         test
DB_PASS         test
DB_PORT         3306
DB_NAME         cmdb

########## CM SNS parameters

# SNS content download scheme (true/ false)
# true: use an adaptable download scheme
# false: default download with static number of contents
DOWNLOAD_SCHEME         0
DOWNLOAD_NUM            5

# Download scheme for image attachments of SNS content
# 0: full mode, 1: partial mode, 2: prefetch mode, 3: none
ATTACH_DOWNLOAD_SCHEME  1

# Number of days used by the prefetching mode
# Access history since (current date - this days) is used for analyzing access score
ATTACH_ACCESS_INTERVAL  7

# Prefetching threshold
# Prefetching is triggered if the estimated interest rate >= this threshold
ATTACH_PREFETCH_THRESHOLD       0.3
#ATTACH_PREFETCH_THRESHOLD      0.0

# Thumbnail image size for SNS content with attachment
# If VER_SIZE is 0, it means the scale is proportional to horizontal scale factor
THUMBNAIL_HOR_SIZE      200
THUMBNAIL_VER_SIZE      0

########## CM simulation parameters

# added artificial delay (ms) per event transmission by the event manager
# 0: default value (no additional delay)
# > 0: before sending an event, the thread sleeps for this value
SIM_TRANS_DELAY         0

########## CM session configuration

# session information
SESSION_NUM     3

SESSION_FILE1   cm-session1.conf
SESSION_NAME1   session1

SESSION_FILE2   cm-session2.conf
SESSION_NAME2   session2

SESSION_FILE3   cm-session3.conf
SESSION_NAME3   session3
```

Figure. CM server configuration file

2.1.1 Configuration for fundamental communication services

Because most configuration fields have been already set to default values, the only thing a developer has to do is to set address information of a default server. CM can configure multiple servers, and they consist of the default server and additional servers. The default server is one to which all clients must always connect. Thus, a server application using CM can be the default server or an additional server. In the case of a single-server system, the server application is the default server. In this example, we use a single server. Following configuration fields are used to set the address and port

of the default server and this server application.

- *SERVER_ADDR* : IP address of the default server.
- *SERVER_PORT* : Port number of the default server.
- *MY_PORT* : Port number of this server application. If it is the default server, this value must be the same as that of the *SERVER_PORT* field.

```
# default server configuration
# If SERVER_ADDR == my address && SERVER_PORT == MY_PORT,
# this server is the default CM server.
#SERVER_ADDR    localhost
SERVER_ADDR     192.168.1.114
SERVER_PORT     7777

# my configuration (this server)
MY_PORT 7777
```

Figure. Configuration example of server address and port numbers

2.1.2 Configuration for extended communication services

In the server configuration file, a developer can set other communication-related policies supported by the CM as well. The details are described below.

- *SYS_TYPE* : application type. A server application must set this field as *SERVER*, and a client as *CLIENT* in the client-server model.

```
# system type (SERVER/CLIENT)
# SERVER: CM server
# CLIENT: CM client
SYS_TYPE        SERVER
```

Figure. Configuration example of the SYS_TYPE field

- *COMM_ARCH* : communication architecture. This field designates the communication architecture of an application using CM. Possible values are *CM_CS* for the client-server model, and *CM_PS* for the hybrid model which uses multicast communication in addition to the client-server model.

```
# communication architecture (CM_CS/CM_PS)
# CM_CS: client-server model
# CM_PS: peer-server model
COMM_ARCH       CM_CS
```

Figure. Configuration example of the COMM_ARCH field

- *UDP_PORT* : default port number of the server application. Using CM, applications can send a message with a UDP connection. This field sets a port number of server application for the default UDP connection, which is open when the server CM starts.

- *MULTICAST_ADDR* : default multicast address. If the CM server sets the hybrid communication architecture, the server application joins this multicast address.

- *MULTICAST_PORT* : default multicast port number. If the CM server sets the hybrid communication architecture, the server application binds to this port number to receive the multicast event through the default multicast group.

```
# default udp configuration
UDP_PORT 8888

# default multicast configuration
MULTICAST_ADDR  224.1.1.1
MULTICAST_PORT  7000
```

Figure. Configuration example of the default UDP port number and multicast address

- *SIM_TRANS_DELAY* : added artificial delay in millisecond per event transmission by CM. If this value is greater than 0, the event sending thread of CM sleeps for this value before it sends a CM event. This field is only used for the simulation of different network condition effect. This value should be 0 for sending an event without any simulated delay.

```
# added artificial delay (ms) per event transmission by the event manager
# 0: default value (no additional delay)
# > 0: before sending an event, the thread sleeps for this value
SIM_TRANS_DELAY         0
```

Figure. Configuration example of the SIM_TRANS_DELAY field

- *LOG_LEVEL* : level of CM log messages printed on the console. CM provides three levels of log messages that are printed out on the console while a CM application is running. If the field value is 0, the log level is minimum and mostly only error messages are printed. If the field value is 1, the log level is normal and common log messages are printed. If the field value is 2, the log level is maximum and log messages per CM event are printed.

```
# 0: CMInfo._DEBUG = false; CMInfo._DEBUG_2 = false
# 1: CMInfo._DEBUG = true; CMInfo._DEBUG_2 = false
# 2: CMInfo._DEBUG = true; CMInfo._DEBUG_2 = true
LOG_LEVEL       1
```

Figure. Configuration example of the LOG_LEVEL field

2.1.3 Configuration for file-transfer service

CM supports different file-transfer functionalities according to the values of following configuration fields.

- *FILE_PATH* : default path for file transfer. CM refers to this path for the file transfer. If a server or client application is requested for a file, the application searches for the requested file in this path. If the client application receives a file, the received file is stored in this directory. If the server application receives a file, the received file is stored in a different sub-directory of this path. The sub-directory is the name of a client that sends the file. If the default path does not exist, CM creates it and its sub-directory when needed.

- *FILE_TRANSFER_SCHEME* : file transfer scheme (0 or 1). CM provides two kinds of the file-transfer services. If the value of this field is 0, CM uses the default TCP socket channel to send and receive a file. If the value of this field is 1, CM uses an advanced file-transfer service where separate blocking TCP socket channels and threads are used to transfer files.

- *FILE_APPEND_SCHEME :* file reception mode (0 or 1). If the value is 0 (overwrite mode), a file receiver always starts to receive the entire file even if the receiver already has parts of the file. If the value of this field is 1 (append mode), the receiver starts to receive only the remaining bytes of the file if the receiver already has parts of the file.

- *PERMIT_FILE_TRANSFER :* auto-permission mode (0 or 1) of file-transfer request. If the value is 0, an application should manually respond the push or pull request of a file by calling CMStub.replyEvent() method in the stub package. If this field is 1, CM automatically accepts every push or pull request.

```
# default directory where a server or a client searches for a requested file
# default directory where a client maintains a downloaded file from a server
FILE_PATH       ./server-file-path

# file transfer scheme (1 or 0)
# 1: file transfer with the additional channel and thread
# 0: file transfer with the default channel and with no additional thread
FILE_TRANSFER_SCHEME    1

# appended file reception scheme (1 or 0)
# 1: file reception of only the remaining file blocks (append mode)
# 0: file reception from the first file block (overwrite mode)
FILE_APPEND_SCHEME      1

# permission mode of file-transfer request (1 or 0)
# 1: automatic permission of push/pull request
# 0: no automatic permission of push/pull request (an application should manually
# respond the request)
PERMIT_FILE_TRANSFER    1
```

Figure. Configuration example of the file-transfer service

2.1.4 Configuration for CM user authentication and session policies

- *LOGIN_SCHEME* : user authentication policy. This field specifies whether a server process user authentication or not. In CM, a client (or a user) must log in to the default server before interacting with other clients. If the value is 0 (false value), it means that the server does not authenticate a user when it receives a login request, and the server CM always accepts the request. If the value is 1 (true value), the server conducts the user authentication process. A server application can use its own authentication method or the authentication of the CM. If the CM is set to use its database (, which also can be set in the server configuration file), the server CM authenticates a user with the registered user information in the DB. Otherwise, a server application must have a separate authentication method, and must notify a requesting client of the authentication result. The detailed login process of the CM is described in another section.

- *MAX_LOGIN_FAILURE* : maximum number of allowable login failures. If the number of login failures of a CM client is greater than the MAX_LOGIN_FAILURE value, the CM server disconnects the default communication channel from the client.

- *KEEP_ALIVE_TIME* : keep-alive time of the server CM. If this field value is greater than 0 in second, the CM server activates connection keep-alive management. In the keep-alive management, if a client does not send any CM event through the default channel for the

keep-alive time that is assigned to the client, the server assumes that the client is not alive, and disconnects it. When a client logs in to the server, it sends its own keep-alive time to the server. If the client's keep-alive time is 0 second, the server assigns its KEEP_ALIVE_TIME value to the client. If the client's keep-alive time is greater than 0, the server assigns that keep-alive time to the client.

If the KEEP_ALIVE_TIME value is set to 0, the server deactivates the keep-alive management, and it does not check whether its clients are alive or not.

- *SESSION_SCHEME* : multi-session policy. With this field, a developer can determine the application will use one session or multiple sessions. In CM, users can be grouped hierarchically with the concept of a session and a group. A server application can set multiple sessions and a session can contain multiple groups. A user always must belong to a session and a group and he/she can interact with other users in his/her session or group. If the field value is 0, CM does not use multiple sessions but only one default session. In this case, when a user logs in to a server, he/she automatically joins this session and its default group. If the value is 1, a server application can configure multiple sessions so that a user can select one of them to join. The details of session management are described later in another section.

```
# login scheme (true / false)
# true: application is responsible for user authentication
# false: no authentication
LOGIN_SCHEME    1

# maximum login failure that is allowed
MAX_LOGIN_FAILURE       5

# keep-alive time (second)
# 0: deactivate keep-alive management
# > 0: activate keep-alive management
KEEP_ALIVE_TIME         60

# session scheme (true / false)
# true: use multiple sessions and groups
# false: no session and group (use a default session and group)
SESSION_SCHEME  0
```

Figure. Configuration example of the login and session policies

2.1.5 Configuration for SNS content download/upload policies

- *DOWNLOAD_SCHEME* : transmission policy of SNS content (0 or 1). This field specifies how

many SNS content a server transmits to a client. If the value is 0, CM adopts a fixed amount of SNS content, which is determined by *DOWNLOAD_NUM* field. If the value is 1, CM uses a dynamic downloading scheme [2].

- *DOWNLOAD_NUM* : default number of SNS content item to be downloaded when a client requests to download SNS content.

- *ATTACH_DOWNLOAD_SCHEME* : download scheme for image attachments of SNS content (0: full mode, 1: partial mode, 2: prefetch mode, 3: none mode). In the full mode, the CM server sends images with the original quality to the client. In the partial mode, the server sends thumbnail images instead of the original images. In the none mode, the server sends only text links to images. [3] In the prefetch mode, the server sends thumbnail images to the client, and sends also original images that the client is interested in. [4]

- *ATTACH_ACCESS_INTERVAL :* number of investigation days of user access history used by the prefetching mode. This field is used only when the *ATTACH_DOWNLOAD_SCHEME* field is set to the prefetch mode (2).

- *ATTACH_PREFETCH_THRESHOLD* : prefetching threshold value (0 ~ 1). Prefetching is triggered if the estimated interest rate >= this threshold. This field is used only when the ATTACH_DOWNLOAD_SCHEME field is set to the prefetch mode (2).

- *THUMBNAIL_HOR_SIZE* : width of the created thumbnail image that is attached to SNS content. Whenever the CM client uploads an image, the CM server creates and stores the corresponding thumbnail image.

- *THUMBNAIL_VER_SIZE* : height of the created thumbnail image that is attached to SNS content. If this value is 0, it means the vertical scale is proportional to the horizontal scale factor comparing to the original image size.

```
# SNS content download scheme (true/ false)
# true: use an adaptable download scheme
# false: default download with static number of contents
DOWNLOAD_SCHEME         0
DOWNLOAD_NUM            5

# Download scheme for image attachments of SNS content
# 0: full mode, 1: partial mode, 2: prefetch mode, 3: none
ATTACH_DOWNLOAD_SCHEME  1

# Number of days used by the prefetching mode
# Access history since (current date - this days) is used for analyzing access score
ATTACH_ACCESS_INTERVAL  7

# Prefetching threshold
# Prefetching is triggered if the estimated interest rate >= this threshold
ATTACH_PREFETCH_THRESHOLD       0.3
#ATTACH_PREFETCH_THRESHOLD      0.0

# Thumbnail image size for SNS content with attachment
# If VER_SIZE is 0, it means the scale is proportional to horizontal scale factor
THUMBNAIL_HOR_SIZE      200
THUMBNAIL_VER_SIZE      0
```

Figure. Configuration example of the SNS content download/upload policies

2.1.6 Configuration for CM DB usage

- *DB_USE* : DB usage flag. This field sets whether a server application uses the internal DB of CM or not. The server CM internally uses MySQL DB especially for its own user management and SNS content management. If the value is 0, the application does not use the CM DB. If the value is 1, the application uses the CM DB and following additional DB information must be set. The details of how to configure CM DB with MySQL is described in the "MySQL server configuration for the CM DB management" section of this document.

  - *DB_HOST* : hostname or IP address of DB. If the DB is installed in the same machine as the server application, this value is set to 'localhost'.
  - *DB_USER* : user name of DB.
  - *DB_PASS* : password of DB.
  - *DB_PORT* : port number of DB.
  - *DB_NAME* : DB name

```
# DB usage (true/ false)
# true: use DB (MySQL) internally for user registration and content management
# false: do not use DB
DB_USE          1
DB_HOST         localhost
DB_USER         test
DB_PASS         test
DB_PORT         3306
DB_NAME         cmdb
```

Figure. Configuration example of the DB usage

2.1.7 Configuration of multiple sessions

- *SESSION_NUM* : number of sessions. The value must be equal to or greater than 1 because CM uses at least one or more sessions. When the server CM starts, it creates sessions as configured here. In the provided server-cm.conf file, three sessions are already configured. According to the number of sessions, a developer also has to configure each session as following fields.

  - SESSION_FILE# : name of session configuration file. '#' is an integer number (starting with 1) in order to differentiate among sessions. In a session configuration file, a developer sets group information of this session, which is described in the next section.

  - SESSION_NAME# : session name. '#' is an integer number (starting with 1) in order to differentiate among sessions. Because a session name in CM is an identifier, a unique name must be assigned.

```
# session information
SESSION_NUM     3

SESSION_FILE1   cm-session1.conf
SESSION_NAME1   session1

SESSION_FILE2   cm-session2.conf
SESSION_NAME2   session2

SESSION_FILE3   cm-session3.conf
SESSION_NAME3   session3
```

Figure. Configuration example of the session

2.2 CM session configuration files

According to the number of sessions, configuration file names and session names in cm-server.conf file, a developer must configure session configuration files. In each file, a developer sets group information of each session such as the number of groups, group names, and their multicast addresses. The following figure is an example of a session configuration file.

```
#group configuration
SESSION_NAME        session1
GROUP_NUM           4

GROUP_NAME1         g1
GROUP_ADDR1         224.1.1.2
GROUP_PORT1         7001

GROUP_NAME2         g2
GROUP_ADDR2         224.1.2.2
GROUP_PORT2         7011

GROUP_NAME3         g3
GROUP_ADDR3         224.1.3.2
GROUP_PORT3         7021

GROUP_NAME4         g4
GROUP_ADDR4         224.1.4.2
GROUP_PORT4         7031
```

Figure. CM session configuration file

Required fields in a session configuration file are described as below.

- *SESSION_NAME* : session name. This value must be the same as that of server-cm.conf file.
- *GROUP_NUM* : number of groups. The value must be equal to or greater than 1 because CM uses at least one or more groups. When the server CM starts, it creates groups as configured here. For each group, a developer must set a group name, group address, and group port number.
  - *GROUP_NAME#* : group name. '#' is an integer number (starting with 1) in order to differentiate among groups. Because a group name in CM is an identifier, a unique name must be assigned.
  - *GROUP_ADDR#* : group multicast address. '#' is an integer number (starting with 1) in order to differentiate among groups. If the communication architecture of CM is configured as *CM_PS* model, a user in a group can communicate with other users in the same group with a multicast channel. Therefore, each group is assigned a multicast address for this purpose.

- *GROUP_PORT#* : group port number. '#' is an integer number (starting with 1) in order to differentiate among groups.

2.3 CM client configuration file

A client application must set the CM client configuration file. A main file is *cm-client.conf* file. The following figure is an example of this file.

```
########## CM fundamental configuration

# system type
# CLIENT: CM client
SYS_TYPE        CLIENT

# default server configuration
#SERVER_ADDR    localhost
SERVER_ADDR     203.252.148.71
SERVER_PORT     7777

# default udp configuration
UDP_PORT 9000

# multicast configuration
MULTICAST_ADDR  224.1.1.1
MULTICAST_PORT  7000

########## Options related to the file transfer

# default directory where a client searches for a requested file
# default directory where a client maintains a downloaded file from a server
FILE_PATH       ./client-file-path

# appended file reception scheme (1 or 0)
# 1: file reception of only the remaining file blocks (append mode)
# 0: file reception from the first file block (overwrite mode)
FILE_APPEND_SCHEME      1

# permission mode of file-transfer request (1 or 0)
# 1: automatic permission of push/pull request
# 0: no automatic permission of push/pull request (an application should manually
# respond the request)
PERMIT_FILE_TRANSFER    0

########## keep-alive strategy

# keep-alive time (second)
# 0: deactivate keep-alive management
# > 0: activate keep-alive management
KEEP_ALIVE_TIME         60

########## CM simulation parameters

# added artificial delay (ms) per event transmission by the event manager
# 0: default value (no additional delay)
# > 0: before sending an event, the thread sleeps for this value
SIM_TRANS_DELAY         0

########## CM Log levels

# 0: minimum level (CMInfo._DEBUG = false; CMInfo._DEBUG_2 = false)
# 1: normal level (CMInfo._DEBUG = true; CMInfo._DEBUG_2 = false)
# 2: maximum level (CMInfo._DEBUG = true; CMInfo._DEBUG_2 = true)
LOG_LEVEL       1
```

Figure. CM client configuration file

Unlike the CM server configuration file, the CM client configuration file does not have many fields. What a developer should set is system type, default server information, UDP port number, and file path information as described below.

- *SYS_TYPE* : application type. A client application must set this field as *CLIENT* in the client-server model.
- *SERVER_ADDR* : IP address of the default server.
- *SERVER_PORT* : Port number of the default server
- *UDP_PORT* : default port number of the server application. Using CM, applications can send a message with a UDP connection. This field sets a port number of server application for the default UDP connection, which is open when the server CM starts.
- *FILE_PATH* : default path for file transfer. CM refers to this path for the file transfer. If a client application is requested for a file, it searches for the file in this path. If the client application receives a file, the received file is stored in this path. If the default path does not exist, CM creates it when needed.

In addition to the above fundamental configuration, the CM client also can set up the following configuration fields.

- *MULTICAST_ADDR* : default multicast address. If the CM server sets the hybrid communication architecture, the server application joins this multicast address.
- *MULTICAST_PORT* : default multicast port number. If the CM server sets the hybrid communication architecture, the server application binds to this port number to receive the multicast event through the default multicast group.
- *FILE_APPEND_SCHEME* : file reception mode (0 or 1). If the value is 0 (overwrite mode), a file receiver always starts to receive the entire file even if the receiver already has parts of the file. If the value of this field is 1 (append mode), the receiver starts to receive only the remaining bytes of the file if the receiver already has parts of the file.
- *PERMIT_FILE_TRANSFER :* auto-permission mode (0 or 1) of file-transfer request. If the value is 0, an application should manually respond the push or pull request of a file by calling

CMStub.replyEvent() method in the stub package. If this field is 1, CM automatically accepts every push or pull request.

- *KEEP_ALIVE_TIME* : keep-alive time of the client CM (in second). If the field value is 0, the client CM deactivates its keep-alive management. That is, it does not send a heartbeat event to the server CM. If the field value is greater than 0, the client CM activates its keep-alive management after it logs in to the server CM by periodically sending the heartbeat event to the server. The period of heartbeat event is the client's KEEP_ALIVE_TIME value. The client CM contains its KEEP_ALIVE_TIME value in the login request event so that the server CM can maintain the heartbeat period of the client.
- *SIM_TRANS_DELAY* : added artificial delay in millisecond per event transmission by CM. If this value is greater than 0, the event sending thread of CM sleeps for this value before it sends a CM event. This field is only used for the simulation of different network condition effect. This value should be 0 for sending an event without any simulated delay.
- *LOG_LEVEL* : level of CM log messages printed on the console. This is the same as the LOG_LEVEL field in the server configuration file (cm-server.conf).

3. Participating in CM network

3.1 Initialization of CM in an application

When a developer completes to set CM configuration files, he/she is now ready for implementing a client-server application using CM.

A developer can create classes for his/her applications. In this guide, we assume that the name of such classes for the server and client applications are *CMServerApp* and *CMClientApp*, respectively.

To initialize and start CM, both the server and the client applications need to import CM package classes, declare an appropriate instance of the CM stub class, set a CM event handler object, and call the start method of the stub class. Once the CM is initialized and starts to run, an application can call CM APIs provided through the stub class.

3.1.1 CM initialization of a desktop server application

The following figure shows a server application class (*CMServerApp*) which initializes CM.

```
import kr.ac.konkuk.ccslab.cm.*;

public class CMServerApp {
	private CMServerStub m_serverStub;
	private CMServerEventHandler m_eventHandler;

	public CMServerApp()
	{
		m_serverStub = new CMServerStub();
		m_eventHandler = new CMServerEventHandler(m_serverStub);
	}

	public CMServerStub getServerStub()
	{
		return m_serverStub;
	}

	public CMServerEventHandler getServerEventHandler()
	{
		return m_eventHandler;
	}

	public static void main(String[] args) {
		CMServerApp server = new CMServerApp();
		CMServerStub cmStub = server.getServerStub();
		cmStub.setAppEventHandler(server.getServerEventHandler());
		cmStub.startCM();
	}

}
```

Figure. Initialization of the server CM

In the *CMServerApp* class, a *CMServerStub* variable is declared because the server application uses CM server stub module. After the application object is created in the *main* method, the next thing to do is to set a CM event handler. After a developer sets the event handler, the application can catch every received CM event from a remote CM object. The details of how to define an event handler is described in the next sub-section. By calling the *startCM* method of the stub object, CM initializes as configured by the CM server configuration file, and starts to run.

3.1.2 CM event handler

In the previous example, an event handler object (m_eventHandler) is created and set to the CM stub module by the *setAppEventHandler* method. An event handler has a role of receiving a CM event whenever it is received. As shown in the following figure, a developer can define an event handler class which includes application codes so that the server application can deal with any task when a specific CM event is received.

```
import kr.ac.konkuk.ccslab.cm.*;
```

```
public class CMServerEventHandler implements CMAppEventHandler {
	private CMServerStub m_serverStub;

	public CMServerEventHandler(CMServerStub serverStub)
	{
		m_serverStub = serverStub;
	}

	@Override
	public void processEvent(CMEvent cme) {
		switch(cme.getType())
		{
		case CMInfo.CM_SESSION_EVENT:
			processSessionEvent(cme);
			break;
		default:
			return;
		}
	}

	private void processSessionEvent(CMEvent cme)
	{
		CMSessionEvent se = (CMSessionEvent) cme;
		switch(se.getID())
		{
		case CMSessionEvent.LOGIN:
			System.out.println("["+se.getUserName()+"] requests
login.");
			break;
		default:
			return;
		}
	}
}
```

Figure. Event handler class

The event handler class, *CMServerEventHandler*, must implement the *CMAppEventHandler* interface which defines an event processing method, *processEvent*. In the above example, the class constructor has one parameter which is a reference to the CM server stub, because the event handler sometimes has to access the stub module in order to call CM APIs as the response to a received CM event. The *processEvent* method defines a developer's event processing functions with the received event of the *CMEvent* class. The above example shows that when the event handler receives a CM event of which type is the session event and its ID is the *CMSessionEvent.LOGIN*, it prints out a message of the event. The *LOGIN* event is sent by a CM client when the client requests to log in to the CM server. The detailed usage of CM event is described in other sections with example codes. Once a developer implements an event processing function and set the event handler to the CM stub, the *processEvent* method will be called by CM whenever it receives an event from a remote CM node.

3.1.3 CM initialization of a desktop client application

The following figure shows a client application class (*CMClientApp*) which initializes CM.

```
import kr.ac.konkuk.ccslab.cm.*;

public class CMClientApp {
	private CMClientStub m_clientStub;
	private CMClientEventHandler m_eventHandler;

	public CMClientApp()
	{
		m_clientStub = new CMClientStub();
		m_eventHandler = new CMClientEventHandler(m_clientStub);
	}

	public CMClientStub getClientStub()
	{
		return m_clientStub;
	}

	public CMClientEventHandler getClientEventHandler()
	{
		return m_eventHandler;
	}

	public static void main(String[] args) {
		CMClientApp client = new CMClientApp();
		CMClientStub cmStub = client.getClientStub();
		cmStub.setAppEventHandler(client.getClientEventHandler());
		cmStub.startCM();
	}
}
```

Figure. Initialization of the server CM

In the *CMClientApp* class, a *CMClientStub* variable is declared because the client application uses CM client stub module. After the application object is created in the *main* method, the next thing to do is to set a CM application event handler. The details of how to define an event handler is the same as the case of the server event handler. By calling the *startCM* method of the stub object, CM initializes as configured by the CM client configuration file and starts to run. This stage is called the *CM_INIT* state of the client.

If a server and client applications successfully initialize CM, the client in the *CM_INIT* state also becomes connected to the server and is ready to log in to the server. We call this state of the client the *CM_CONNECT* state.

3.2 Login to default server

A CM client must log in to the default server in order to interact with other CM nodes. For the login process, CM client stub provides the *loginCM* method.

```
public boolean loginCM(String strUserName, String strPassword)
```

Figure. login API in the CM client stub.

This method takes two parameters: a user name and password. The password parameter is a plain password string, and it is encrypted in CM using the *CMUtil.getSHAHash()* method. When a client calls this method, the client CM requests login to the default server. The following simple example shows that a client application receives a user name and password, and requests login to the default server.

```
// CM initialization has completed.

String strUserName = null;
String strPassword = null;
boolean bRequestResult = false;
Console console = System.console();

System.out.print("user name: ");
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
try {
	strUserName = br.readLine();
	if(console == null)
	{
		System.out.print("password: ");
		strPassword = br.readLine();
	}
	else
		strPassword = new String(console.readPassword("password: "));
} catch (IOException e) {
	e.printStackTrace();
}

bRequestResult = m_clientStub.loginCM(strUserName, strPassword);
if(bRequestResult)
	System.out.println("successfully sent the login request.");
else
	System.err.println("failed the login request!");
```

Figure. Login request

When the server CM receives the login request, it authenticates a requesting user according to the CM login scheme, which is set in the CM server configuration file (cm-server.conf).

3.2.1 No login scheme

If the server CM sets the *LOGIN_SCHEME* field as 0, it does not authenticate a user and accepts every login request, and a requesting client always succeeds to log in to the default server. Thus, the server and client applications do not need to do anything else after the login request.

3.2.2 Using login scheme

If the server CM sets the *LOGIN_SCHEME* field as 1, a server application is responsible for authenticating the requesting user with its own authentication policy. To this end, the server application has to catch the login request event (*CMSessionEvent.LOGIN*) in the event handler, authenticate the user, and notify the user of the authentication result. The following example shows the authentication routine which uses CM DB management module for the authentication. The authentication codes are added to the *processSessionEvent* method of the server event handler in the previous example.

```
…
private void processSessionEvent(CMEvent cme)
{
	CMConfigurationInfo confInfo =
m_serverStub.getCMInfo().getConfigurationInfo();
	CMSessionEvent se = (CMSessionEvent) cme;
	switch(se.getID())
	{
	case CMSessionEvent.LOGIN:
		System.out.println("["+se.getUserName()+"] requests login.");
		if(confInfo.isLoginScheme())
		{
			boolean ret =
CMDBManager.authenticateUser(se.getUserName(), se.getPassword(),
                                                m_serverStub.getCMInfo());
			if(!ret)
			{
				System.out.println("["+se.getUserName()+"]
authentication fails!");
				m_serverStub.replyEvent(se, 0);
			}
			else
			{
				System.out.println("["+se.getUserName()+"]
authentication succeeded.");
				m_serverStub.replyEvent(se, 1);
			}
		}
		break;
	default:
		return;
	}
}
```

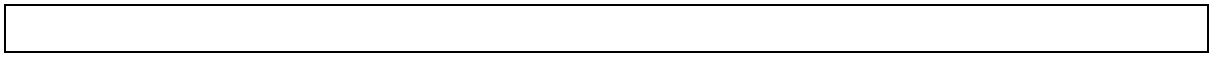


Figure. Process of login request

For the user authentication with the CM DB manager, the server application calls the *authenticateUser* method, which checks whether the given user name is registered in the CM DB or not, and whether the given password is correct or not. The return value of this method is the Boolean type. The details of the usage of CM DB manager is described in the other section. Of course, the server application may use different authentication method instead of the CM DB manager. No matter of which authentication method is used by the server application, the authentication result is given to CM by calling the *CMStub.replyEvent()* method in the stub package. The *replyEvent()* method takes two parameters. The first one is the received login request event. The second parameter is a return code that says whether the requesting user is a valid user or not. If the value is 1, the user is a valid user. If the value is 0, the user is not a valid user. If the server application calls the *replyEvent()* method, the server CM makes a response event (*CMSessionEvent.LOGIN_ACK*) to the login request, and sends it to the requesting client.

```
public boolean replyEvent(CMEvent event, int nReturnCode)
```

Figure. CMStub.replyEvent() method

To check whether the login request is successful or not, the client event handler needs to catch the *LOGIN_ACK* event. The following example shows how the client event handler figures out the result of the login request. In the *LOGIN_ACK* event, a result field of the Integer type is set, and the value can be retrieved by the *isValidUser* method. If the value is 1, the login request successfully completes and the requesting client is in the *CM_LOGIN* state. Otherwise, the login process fails.

```
import kr.ac.konkuk.ccslab.cm.*;

public class CMClientEventHandler implements CMEventHandler {
	private CMClientStub m_clientStub;

	public CMClientEventHandler(CMClientStub stub)
	{
		m_clientStub = stub;
	}

	@Override
	public void processEvent(CMEvent cme) {
		switch(cme.getType())
		{
		case CMInfo.CM_SESSION_EVENT:
			processSessionEvent(cme);
			break;
		default:
```

```
                        return;
                }
        }

        private void processSessionEvent(CMEvent cme)
        {
                CMSessionEvent se = (CMSessionEvent)cme;
                switch(se.getID())
                {
                case CMSessionEvent.LOGIN_ACK:
                        if(se.isValidUser() == 0)
                        {
                                System.err.println("This client fails
authentication by the default server!");
                        }
                        else if(se.isValidUser() == -1)
                        {
                                System.err.println("This client is already in the
login-user list!");
                        }
                        else
                        {
                                System.out.println("This client successfully logs
in to the default server.");
                        }
                        break;
                default:
                        return;
                }
        }
}
```

Figure. Checking the login request

The detailed information of *LOGIN_ACK* event is described below.

Event Type		CMInfo.CM_SESSION_EVENT	
Event ID		CMSessionEvent.LOGIN_ACK	
Event field	Field Data type	Field definition	Get method
User validity	Int	1: valid user 0: user authentication failed -1: the same user already logged in	isValidUser()
Communication architecture	String	Specified communication architecture CM_CS: client-server model CM_PS: client-server with multicast	getCommArch()
Login scheme	Int	1: user authentication used 0: no user authentication	isLoginScheme()

Session scheme	Int	1: multiple sessions used 0: single session used	isSessionScheme()

Table. LOGIN_ACK event

3.2.3 Synchronous login

The aforementioned login process is conducted in an asynchronous manner. CM also provides a synchronous login API as follows. [8]

```
public CMSessionEvent syncLoginCM(String strUserName, String strPassword)
```

Figure. synchronous login API in the CM client stub.

The parameters of the *syncLoginCM()* method are the same as those of the asynchronous login method (*loginCM(String, String)*). The *strUserName* parameter is the login user name and the *strPassword* parameter is the plain password text that is encrypted in CM. Unlike the asynchronous login method, this method makes the main thread of the client block its execution until it receives and returns the reply event (CMSessionEvent.LOGIN_ACK) from the default server. Therefore, the client can synchronously get the login result from the default server as the return value of this method. The following code snippet shows an example of using the synchronous login service of CM.

```
// CM initialization has completed.

String strUserName = null;
String strPassword = null;
CMSessionEvent loginAckEvent = null;
Console console = System.console();

System.out.print("user name: ");
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
try {
        strUserName = br.readLine();
        if(console == null)
        {
                System.out.print("password: ");
                strPassword = br.readLine();
        }
        else
                strPassword = new String(console.readPassword("password: "));
```

```
} catch (IOException e) {
        e.printStackTrace();
}

loginAckEvent = m_clientStub.syncLoginCM(strUserName, strPassword);
if(loginAckEvent != null)
{
        // print login result
        if(loginAckEvent.isValidUser() == 0)
                System.err.println("This client fails authentication by the default server!");
        else if(loginAckEvent.isValidUser() == -1)
                System.err.println("This client is already in the login-user list!");
        else
                System.out.println("This client successfully logs in to the default server.");
}
else
        System.err.println("failed the login request!");
```

Figure. Synchronous login example.

3.2.4 Notification of login of other users

When the server CM accepts the login request from a client, the server CM also notifies other participating clients of the information of the login user with the *SESSION_ADD_USER* event. A client application can catch this event in the event handler routine if it wants to use such information. The detailed information of the *SESSION_ADD_USER* event is shown below.

Event Type		CMInfo.CM_SESSION_EVENT	
Event ID		CMSessionEvent.SESSION_ADD_USER	
Event field	Field Data type	Field definition	Get method
User name	String	Name of the login user	getUserName()
Host address	String	Host address of the login user	getHostAddress()

Table. SESSION_ADD_USER event

3.3 Joining a session

After the login process has completed, a client application must join a session and a group of CM

to finish entering the CM network. The session join process is different according to whether the server CM adopts single session or multiple sessions in the CM server configuration file. If a client joins a session, the client CM automatically proceeds to join a default group of the session.

3.3.1 Single session

If the *SESSION_SCHEME* field is 0 in the CM server configuration file, the server application uses single session. The single session information and at least one group in it must be configured as well. In this case, the client CM automatically requests to join the session as soon as the login request finishes. Thus, the client application has nothing to do explicitly for joining a session.

3.3.2 Multiple sessions

If the *SESSION_SCHEME* field is 1 in the CM server configuration file, the server application supports multiple sessions so that a client can choose one of them to join. A developer must configure multiple sessions in the server configuration file, and must configure at least one group in each session in separate session configuration file as well. After the login process, the client application does not know how many sessions are provided by the server. Therefore, the client needs to request session information, choose one session, and request to join a session.

3.3.3 Requesting session information

To request session information to the default server, the client application simply can call the *requestSessionInfo()* method of the CM client stub.

```
public boolean requestSessionInfo()
```

Figure. Synopsis of requestSessionInfo().

When the server CM receives this request, it makes and sends a response event which contains available session information such as session name, server address, server port number, and current number of session members. The client application can receive session information in the client event handler as shown in the following example. The example shows only the relevant part of the entire event handler class.

```
…
public void processEvent(CMEvent cme) {
        switch(cme.getType())
        {
        case CMInfo.CM_SESSION_EVENT:
                processSessionEvent(cme);
                break;
```

```
        default:
                return;
        }
}

private void processSessionEvent(CMEvent cme)
{
        CMSessionEvent se = (CMSessionEvent)cme;
        switch(se.getID())
        {
        case CMSessionEvent.RESPONSE_SESSION_INFO:
                processRESPONSE_SESSION_INFO(se);
                break;
        default:
                return;
        }
}

private void processRESPONSE_SESSION_INFO(CMSessionEvent se)
{
        Iterator<CMSessionInfo> iter = se.getSessionInfoList().iterator();
        System.out.format("%-60s%n", "------------------------------------------
------------------");
        System.out.format("%-20s%-20s%-10s%-10s%n", "name", "address", "port",
"user num");
        System.out.format("%-60s%n", "------------------------------------------
------------------");

        while(iter.hasNext())
        {
                CMSessionInfo tInfo = iter.next();
                System.out.format("%-20s%-20s%-10d%-10d%n",
tInfo.getSessionName(), tInfo.getAddress(), tInfo.getPort(),
tInfo.getUserNum());
        }
}
```

Figure. Receiving session information

The response event of the *requestSessionInfo* method is the *RESPONSE_SESSION_INFO* of the *CMSessionEvent* class. This event contains a session information list as a Java Vector object. The reference to this Vector can be retrieved by the *getSessionInfoList* method of the response event. Each Vector element is a reference to the *CMSessionInfo* class which contains session information such as the session name, the server address, the server port number, and the current number of session members. The result of the above example is in the following figure.

```
> ------------------------------------------------------------
name                address             port      user num
------------------------------------------------------------
session1            192.168.2.9         7777      0
session2            192.168.2.9         7777      0
session3            192.168.2.9         7777      0
```

Figure. Received session information

The detailed information of the *RESPONSE_SESSION_INFO* event is described below.

Event Type		CMInfo.CM_SESSION_EVENT	
Event ID		CMSessionEvent.RESPONSE_SESSION_INFO	
Event field	Field Data type	Field definition	Get method
Number of sessions	Int	Number of sessions	getSessionNum()
Vector of sessions	Vector<CMSessionInfo>	List of session information	getSessionInfoList()

Table. RESPONSE_SESSION_INFO event

3.3.4 Synchronously requesting session information

The aforementioned *requestSessionInfo()* method asynchronously gets the current session information from the default server. CM also provides the synchronous version that is called *syncRequestSessionInfo()* as follows.

```
public CMSessionEvent syncRequestSessionInfo()
```

Figure. Synopsis of syncRequestSessionInfo().

Unlike the asynchronous method (*requestSessionInfo()*), this method makes the main thread of the client block its execution until it receives and returns the reply event (*CMSessionEvent.RESPONSE_SESSION_INFO*) from the default server. The following code snippet shows an example of using the *syncRequestSessionInfo()* method.

```
// We assume that the client has logged in to the default server.

CMSessionEvent se = null;
se = m_clientStub.syncRequestSessionInfo();
if(se == null)
{
        System.err.println("failed the session-info request!");
        return;
}
// print the request result
Iterator<CMSessionInfo> iter = se.getSessionInfoList().iterator();
System.out.format("%-60s%n", "------------------------------------------------------------");
System.out.format("%-20s%-20s%-10s%-10s%n", "name", "address", "port", "user num");
```

```
System.out.format("%-60s%n", "------------------------------------------------------------");
while(iter.hasNext())
{
	CMSessionInfo tInfo = iter.next();
	System.out.format("%-20s%-20s%-10d%-10d%n", tInfo.getSessionName(),
tInfo.getAddress(), tInfo.getPort(), tInfo.getUserNum());
}
```

Figure. syncRequestSessionInfo() usage example.

3.3.5 Requesting session join

If the CM server provides multiple sessions, a client application can join a session only after it requests and receives session information from the default server. To join a session, the client application calls the *joinSession* method of the client stub.

```
public boolean joinSession(String sname)
```

Figure. Synopsis of joinSession().

The *joinSession* method takes one parameter which is a session name to which the client joins. The following line of code is an example where the client joins a session of which name is given by the user.

```
// We assume that the client has logged in to the default server and received session information.

String strSessionName = null;
boolean bRequestResult = false;
System.out.print("session name: ");
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
try {
	strSessionName = br.readLine();
} catch (IOException e) {
	e.printStackTrace();
}
bRequestResult = m_clientStub.joinSession(strSessionName);
if(bRequestResult)
	System.out.println("successfully sent the session-join request.");
else
	System.err.println("failed the session-join request!");
```

Figure. Join request of a session

When the server CM receives the join-session request, it sends back the client a reply event (CMSessionEvent.JOIN_SESSION_ACK) that contains the group information in the session. If the reply event does not have any group information, it implies that the join-session task fails because every session in CM must has at least one group. The detailed information of the reply event is as follows.

Event Type		CMInfo.CM_SESSION_EVENT	
Event ID		CMSessionEvent.JOIN_SESSION_ACK	
Event field	Field Data type	Field definition	Get method
Number of groups	int	Number of groups in this session	getGroupNum()
Group list	Vector<CMGroupInfo>	Vector of group information. Each group information contains a group name, group multicast address, and group multicast port number.	getGroupInfoList()

Table. JOIN_SESSION_ACK event.

If the client CM finishes joining a session, it becomes in the *CM_SESSION_JOIN* state and it automatically proceeds to enter the first group of the session. For example, if the client joins "session1" of the aforementioned server configuration file, it also enters the group, "g1", which is the first group of the session, "session1".

Only if a client enters a group, it finally becomes ready to interact with other CM nodes.

3.3.6 Synchronously requesting session join

The aforementioned *joinSession()* method works asynchronously such that the client should receive the reply event asynchronously in its event handler. CM also provides the synchronous version of this method, which is called *syncJoinSession()* as follows.

```
public CMSessionEvent syncJoinSession(String sname)
```

Figure. Synopsis of syncJoinSession().

Unlike the asynchronous method (*joinSession(String)*), this method makes the main thread of the client block its execution until it receives and returns the reply event (*CMSessionEvent.JOIN_SESSION_ACK*) from the default server. The following code snippet shows an example of using the *syncJoinSession()* method.

```
// We assume that the client has logged in to the default server and received session information.

CMSessionEvent se = null;
String strSessionName = null;
System.out.print("session name: ");
```

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
try {
        strSessionName = br.readLine();
} catch (IOException e) {
        e.printStackTrace();
}

se = m_clientStub.syncJoinSession(strSessionName);
if(se != null)
        System.out.println("successfully joined a session that has ("+se.getGroupNum()+")
groups.");
else
        System.err.println("failed the session-join request!");
```

Figure. Usage example of syncJoinSession().

3.3.7 Notification of joining a session of other users

When the server CM completes the session joining request from a client, the server CM also notifies other participating clients of the information of the new session user with the *CHANGE_SESSION* event. A client application can catch this event in the event handler routine if it wants to use such information. The detailed information of the *CHANGE_SESSION* event is shown below.

Event Type		CMInfo.CM_SESSION_EVENT	
Event ID		CMSessionEvent.CHANGE_SESSION	
Event field	Field Data type	Field definition	Get method
User name	String	Name of a user who joins a session	getUserName()
Session name	String	Name of a session which the user joins	getSessionName()

Table. CHANGE_SESSION event

3.3.8 Notification of joining a group of other users

When the server CM completes the group joining request from a client, the server CM also notifies other participating clients of the information of the new group user with the *NEW_USER* event. When the client CM receives this event, it stores the information of a new group user so that it can figure out current group members later. A client application also can catch this event in the event handler routine if it wants to use such information. The detailed information of the *NEW_USER* event

is shown below.

Event Type		CMInfo.CM_DATA_EVENT	
Event ID		CMDataEvent.NEW_USER	
Event field	Field Data type	Field definition	Get method
Current session	String	Current session name of the user	getHandlerSession()
Current group	String	Current group name of the user	getHandlerGroup()
User name	String	Name of the new group user	getUserName()
Host address	String	Host address of the new group user	getHostAddress()
UDP port number	int	UDP port number of the new group user	getUDPPort()

Table. NEW_USER event

3.3.9 Notification of existing group members

When the server CM completes the group joining request from a client, the server CM also notifies the new user of the information of other existing group users with the series of *INHABITANT* events. When the client CM receives this event, it stores the information of an existing group user so that it can figure out current group members later. A client application also can catch this event in the event handler routine if it wants to use such information. The detailed information of the *INHABITANT* event is shown below.

Event Type		CMInfo.CM_DATA_EVENT	
Event ID		CMDataEvent.INHABITANT	
Event field	Field Data type	Field definition	Get method
Current session	String	Current session name of the user	getHandlerSession()
Current group	String	Current group name of the user	getHandlerGroup()
User name	String	Name of the existing group user	getUserName()
Host address	String	Host address of the existing group user	getHostAddress()
UDP port number	Int	UDP port number of the existing group user	getUDPPort()

Table. INHABITANT event

4. Leaving CM network

A developer should notice that a client application requires the initialization, the login, the session join, and the group join processes in the order named to interact with other CM nodes. These processes incur the following transition of client states.

CMInfo.CM_INIT -> CMInfo.CM_CONNECT -> CMInfo.CM_LOGIN -> CMInfo.CM_SESSION_JOIN

Note that the server CM does not have such states. From the *CMInfo.CM_SESSION_JOIN* state where a client joins a group and a session, the client can transit to any previous state as described below.

4.1 Leaving current session

To leave current session, a client can call the *leaveSession* method of the CM client stub as shown in the following example. This method first makes the client leave the current group and session. When a client leaves a session, it changes to the *CMInfo.CM_LOGIN* state and cannot interact with other CM nodes any more. To join a session again, the client must call *joinSession* method with a session name parameter.

```
// m_clientStub is a reference to the CM client stub object.
m_clientStub.leaveSession();
```

Figure. Leaving current session

4.1.1 Notification of leaving a session of other users

When the server CM completes the session leaving request from a client, the server CM also notifies other participating clients of the information of the leaving user with the *CHANGE_SESSION* event. A client application can catch this event in the event handler routine if it wants to use such information. If the session name field of this event is an empty space, a client can know that the user leaves his/her current session.

4.1.2 Notification of leaving a group of other users

When the server CM completes the group leaving request from a client, the server CM also notifies other participating clients of the information of the leaving group user with the *REMOVE_USER* event. When the client CM receives this event, it deletes the information of the leaving group user. A client application also can catch this event in the event handler routine if it wants to use such information. The detailed information of the *REMOVE_USER* event is shown below.

Event Type		CMInfo.CM_DATA_EVENT	
Event ID		CMDataEvent.REMOVE_USER	
Event field	Field Data type	Field definition	Get method
Current session	String	Current session name of the user	getHandlerSession()
Current group	String	Current group name of the user	getHandlerGroup()
User name	String	Name of the leaving group user	getUserName()

Table. REMOVE_USER event

4.2 Logout from default server

To log out from the default server, a client can call the *logoutCM* method of the CM client stub as shown in the following example. If a client is in the *CMInfo.CM_SESSION_JOIN* state, this method first makes the client leave the session. When a client logs out from the default server, it changes to the *CMInfo.CM_CONNECT* state and still keeps the connection to the default server. If the client wants to enter the CM network again, it must follow the login, the session join, and the group join processes.

```
// m_clientStub is a reference to the CM client stub object.
m_clientStub.logoutCM();
```

Figure. Logout from default server

4.2.1 Notification of logout of other users

When the server CM completes the logout request from a client, the server CM also notifies other participating clients of the information of the logout user with the *SESSION_REMOVE_USER* event. A client application can catch this event in the event handler routine if it wants to use such information. The detailed information of the *SESSION_REMOVE_USER* event is shown below.

Event Type		CMInfo.CM_SESSION_EVENT	
Event ID		CMSessionEvent.SESSION_REMOVE_USER	
Event field	Field Data type	Field definition	Get method
User name	String	Name of the logout user	getUserName()

Table. SESSION_REMOVE_USER event

4.3 Disconnection from default server

To disconnect from the default server, a client can call the *disconnectFromServer("SERVER")* method of the CM client stub as shown in the following example. Frist, this method makes the client leave the current group and session and log out from the default server if the client is in a corresponding

state. Finally, the client disconnects from the default server and changes to the *CMInfo.CM_INIT* state. If the client wants to enter the CM network again, it must connect to the default server by calling the *connectToServer("SERVER")* method of the CM client stub, and must follow the login, session join, and group join processes as before.

```
// m_clientStub is a reference to the CM client stub object.
m_clientStub.disconnectFromServer();
```

Figure. Disconnection from default server

4.4 Terminating CM

If a client or a server application does not use CM and wants to terminate it, it can call the *terminateCM* method of the CM stub as shown in the following example. This method is the counterpart of the *startCM* method and finishes the execution of CM. In the case of a client application, before the termination, the client leaves the current group and session, logs out, and disconnect from the default server in order to keep the remaining CM network in a consistent state. Furthermore, when a server or client application is unexpectedly closed, other remaining CM nodes are notified of the unexpected disconnection and delete the information of the closed CM node.

```
// m_clientStub is a reference to the CM client stub object.
m_clientStub.terminateCM();
```

Figure. Termination of CM client

5. Changing current group

If a session has multiple groups, a client joins the first group after it joins the session. The client can change a current group by calling the *changeGroup* method of the CM client stub. The parameter of the method is the name of target group to which the client wants to join. The target group name surely must exist in the configuration file of a session in a server application. Then, the client leaves the current group and joins the new target group. The following codes are an example of changing a group of which name is given by the user.

```
// A client already has joined a group.
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String strGroupName = null;
System.out.println("====== change group");
try {
        strGroupName = br.readLine();
} catch (IOException e) {
        e.printStackTrace();
```

```
}
m_clientStub.changeGroup(strGroupName);
```

Figure. Changing a current group

6. A chatting event

When a client interacts with other clients or a server, the most frequent way is a chatting event. A client simply can send a chatting event by calling the *chat* method of the CM client stub. This method takes two String parameters: a target and a text message. The target parameter must start with '/' character and it specifies the range of recipients of a chatting message as described below.

Target	Definition
/b	A chatting message is sent to the all login users.
/s	A chatting message is sent to the all session members of the sending user.
/g	A chatting message is sent to the all group members of the sending user.
/'name'	A chatting message is sent to a specific CM node of which name is 'name'. The name can be another user name or a server name. If 'name' is *SERVER*, the message is sent to the default server.

Table. Definition of the target parameter of the *chat* method

The following codes are an example of using the *chat* method.

```
// A client already has joined a group.
String strTarget = null;
String strMessage = null;
System.out.println("====== chat");
System.out.print("target(/b, /s, /g, or /username): ");
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
try {
        strTarget = br.readLine();
        strTarget = strTarget.trim();
} catch (IOException e) {
        e.printStackTrace();
}
System.out.print("message: ");
try {
        strMessage = br.readLine();
} catch (IOException e) {
        e.printStackTrace();
}

m_clientStub.chat(strTarget, strMessage);
```

Figure. Example of using the *chat* method

6.1 Receiving a chatting event

A CM application can receive a chatting event by catching a pre-defined CM event in the event handler like other events. There are two types of CM chat events. One is the *SESSION_TALK* event of the *CMSessionEvent* class. A client can receive this event if it at least logs in to the default server. The other event is the *USER_TALK* event of the *CMInterestEvent* class. A client can receive this event only if it joins a group. The following tables describe the detailed field information of these events.

Event Type		CMInfo.CM_SESSION_EVENT	
Event ID		CMSessionEvent.SESSION_TALK	
Event field	Field Data type	Field definition	Get method
User name	String	Name of the sending user	getUserName()
Text message	String	A chatting message	getTalk()
Session name	String	A current session of the sending user	getHandlerSession()

Table. *SESSION_TALK* event

Event Type		CMInfo.CM_INTEREST_EVENT	
Event ID		CMInterestEvent.USER_TALK	
Event field	Field Data type	Field definition	Get method
User name	String	Name of the sending user	getUserName()
Text message	String	A chatting message	getTalk()
Session name	String	A current session of the sending user	getHandlerSession()
Group name	String	A current group of the sending user	getHandlerGroup()

Table. *USER_TALK* event

The following codes are an example of catching the chatting events and printing out the received information. The example shows only the relevant part of the entire event handler class.

```
…
public void processEvent(CMEvent cme) {
        switch(cme.getType())
        {
        case CMInfo.CM_SESSION_EVENT:
                processSessionEvent(cme);
                break;
        case CMInfo.CM_INTEREST_EVENT:
                processInterestEvent(cme);
                break;
        default:
                return;
```

```
        }
}

private void processSessionEvent(CMEvent cme)
{
        CMSessionEvent se = (CMSessionEvent)cme;
        switch(se.getID())
        {
        case CMSessionEvent.SESSION_TALK:
                System.out.println("("+se.getHandlerSession()+")");
                System.out.println("<"+se.getUserName()+">: "+se.getTalk());
                break;
        default:
                return;
        }
}

private void processInterestEvent(CMEvent cme)
{
        CMInterestEvent ie = (CMInterestEvent) cme;
        switch(ie.getID())
        {
        case CMInterestEvent.USER_TALK:
                System.out.println("("+ie.getHandlerSession()+",
"+ie.getHandlerGroup()+")");
                System.out.println("<"+ie.getUserName()+">: "+ie.getTalk());
                break;
        default:
                return;
        }
}
```

Figure. Receiving chatting events

7. Sending/receiving a CM event

In this section, we describe how a developer can send and receive a CM event in an application.

7.1 Creating an event (*CMDummyEvent*)

Because CM handles every outgoing and incoming message as the form of a CM event, a developer first needs to create an appropriate CM event in order to send an event. For simplicity, we assume that a developer wants to send the *CMDummyEvent*, which is one of supported event types of CM and has only one String field. This event is useful when a developer designs a simple event which contains the semantic in a String variable. To create an event of the *CMDummyEvent* type, the required fields are described as below.

Event Type	CMInfo.CM_DUMMY_EVENT

Event field	Field Data type	Field definition	Set method
Session name	String	A current session of the sending user	setHandlerSession()
Group name	String	A current group of the sending user	setHandlerGroup()
String message	String	A string value	setDummyInfo()

Table. CMDummyEvent fields and set methods

The following codes are an example of creating and sending an event of the *CMDummyEvent* type to all the group members of a sending client.

```
// A client already has joined a group.
CMInteractionInfo interInfo = m_clientStub.getCMInfo().getInteractionInfo();
CMUser myself = interInfo.getMyself();

System.out.println("====== test CMDummyEvent in current group");
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.print("input message: ");
String strInput = null;
try {
	strInput = br.readLine();
} catch (IOException e) {
	e.printStackTrace();
}

CMDummyEvent due = new CMDummyEvent();
due.setHandlerSession(myself.getCurrentSession());
due.setHandlerGroup(myself.getCurrentGroup());
due.setDummyInfo(strInput);
m_clientStub.cast(due, myself.getCurrentSession(), myself.getCurrentGroup());
due = null;
```

Figure. Example of sending a CMDummyEvent event

In the above example, the client application first get a reference to a user object (myself), which contains the detailed information of the user such as the user name, current session name, current group name, and so on. In CM, entire internal information is stored in the *CMInfo* class and the instance of this class is retrieved by the *getCMInfo* method of the CM stub object. The *CMInfo* class contains different information classes and one of them is the *CMInteractionInfo* class. In the client CM, the *CMInteractionInfo* class contains global information such as user information, default server information, available session information received by the default server, and other information on additional servers. The reference of the *CMInteractionInfo* object can be retrieved by the *getInteractionInfo* method of the *CMInfo* class. Finally, the user information is accessed by calling the *getMyself* method of the *CMInteractionInfo* class. The getMyself method is frequently used in order for a client to retrieve the current user information, the session name, and the group name.

In the next part of the example, the client is provided a string message with the keyboard input by the user.

To create an event of the *CMDummyEvent* type, the client declares an object of the event, and set the required fields that are the current session, current group, and string message. When a developer declares an instance of a CM event class, the event type is automatically assigned to the instance because the selected event class already specifies the corresponding event type. The current session and group to which the user joins are retrieved by calling the *getCurrentSession* and *getCurrentGroup* methods of the user object.

As the last step, the client sends the event by calling the *cast* method of the CM stub. The cast method sends an event to all users in the specified group of the specified session. CM supports various transmission modes of an event with different sending methods and options. The details of the transmission modes are described in the next sub-section.

7.2 Sending a CM event

A CM server or client application can send an event with one of three transmission modes: one-to-one, one-to-many, and one-to-all transmission. In the one-to-one transmission, there is only one receiver. In the one-to-many and one-to-all modes, there can be multiple receivers. Such transmission modes are realized by three methods of the CM stub module: *send*, *cast (or multicast)*, and *broadcast* methods.

7.2.1 *send* method

The *send* method sends a CM event to one receiver. The synopsis of this method is described below.

```
boolean send(CMEvent cme, String strTarget)
```

Figure. send method

The *send* method requires at least two parameters. The first parameter is a CM event. Because the *CMEvent* class is the inherited class of every child event class, any type of CM event can be sent. The second parameter is the name of a receiver. The name can be a server name or a user name. If the target is the default server, CM defines its name as *SERVER*. With the two parameters, the *send* method sends an event with the default TCP connection.

7.2.2 *cast* method

The *cast* method sends a CM event to a receivers group using multiple one-to-one transmission.

The range of receivers can be set with a session and a group. That is, this method sends an event to a specific session members or group members. The synopsis of this method is described below.

```
boolean cast(CMEvent cme, String sessionName, String groupName)
```

Figure. cast method

The *cast* method requires at least three parameters. Like the *send* method, the first parameter is a CM event. The second and third parameters specify the limit of receivers. With the different combination of the values of these parameters, the range of receivers is also different as described below. With the two parameters, the *cast* method sends an event with the default TCP connection.g

sessionName	groupName	Target receivers
Not null	Not null	All member users of a group (*groupName*) in a session (*sessionName*)
Not null	Null	All member users of a session (*sessionName*)
null	Null	All users who log in to the default server

Table. Range of receivers

7.2.3 *multicast* method

If a CM server application sets the communication architecture as the *CM_PS* in the server configuration file, it creates a multicast channel assigned to each group. In this case, the server and client applications can send an event using a multicast channel by calling the *multicast* method. The synopsis of this method is described below.

```
boolean multicast(CMEvent cme, String sessionName, String groupName)
```

Figure. multicast method

The *multicast* method requires at least three parameters. The first parameter is a CM event and the other two parameters specify a target session and group. Unlike the *cast* method, both the session and group name values must not be null, because the multicast is enabled only for a specific group. Particularly, a client application can multicast an event only to its current group in its current session.

7.2.4 *broadcast* method

The *broadcast* method sends a CM event to all users who currently log in to the default server via multiple one-to-one transmissions. The synopsis of this method is described below.

```
boolean broadcast(CMEvent cme)
```

Figure. broadcast method

The *broadcast* method does not have a target parameter and requires only one parameter, a CM event to be sent. In this case, this method sends an event with the default TCP connection.

7.2.5 Sending options

- Transmission reliability

Except the *multicast* method, the other three CM transmission methods (*send*, *cast*, and *broadcast*) send an event through the default TCP connection which has already been established between the default server and a client. Furthermore, they can also send an event with the default UDP connection as well if a sending option is specified in an additional parameter. The extended synopsis of the three transmission methods is described below.

```
boolean send(CMEvent cme, String strTarget, int opt)
boolean cast(CMEvent cme, String sessionName, String groupName, int opt)
boolean broadcast(CMEvent cme, int opt)
```

Figure. Transmission methods with a sending option

The three methods have an additional parameter (*opt*), a sending option. As an *int* type value, the option parameter specifies one of two transport layer protocols, TCP or UDP. If the *opt* parameter value is *CMInfo.CM_STREAM* or 0, a transmission method uses the TCP connection to send an event. If the parameter is *CMInfo.CM_DATAGRAM* or 1, a transmission method uses the UDP connection.

The base transmission methods can be represented with those with the sending option. For example, the *cast* method in the previous example (Figure 21) can be represented differently as below. All the following method calls have the same effect.

```
M_clientStub.cast(due, myself.getCurrentSession(), myself.getCurrentGroup());
m_clientStub.cast(due, myself.getCurrentSession(), myself.getCurrentGroup(), 0);
m_clientStub.cast(due, myself.getCurrentSession(), myself.getCurrentGroup(),
CMInfo.CM_STREAM);
```

Figure. Different form of the cast method

- Multiple communication channels

According to application's requirement, two CM nodes can establish multiple TCP or UDP channels between them using APIs of the CM stub module. The details of how to add and remove additional channel is explained in another section. In this case, a sending node needs to select a channel if it sends an event to the other node. To support this functionality, all the CM transmission methods provide additional option parameter, a channel index. The synopsis of the methods is described below.

```
boolean send(CMEvent cme, String strTarget, int opt, int nChNum)
boolean cast(CMEvent cme, String sessionName, String groupName, int opt, int
nChNum)
boolean multicast(CMEvent cme, String sessionName, String groupName, int nChNum)
boolean broadcast(CMEvent cme, int opt, int nChNum)
```

Figure. Transmission methods with a sending option and a channel index

The last parameter (*nChNum*) is the *int* value of a channel index. CM internally stores every established channel, and each channel is identified with an integer index. The index of the default socket channel is 0, and the index of additional channel must be greater than the default index. The index of the default datagram channel is a default UDP port number that is set in the CM configuration file of the sending node. The index (port number) of the target (receiving) node is implicitly set to the default UDP port number of that node. If a sending node wants to send a CM event to an additional datagram channel of a receiving node, the sender should call the other *send*() method that has a separate target port number parameter.

The base transmission methods can be represented with those with the sending option and the channel index parameters. For example, the *cast* method in the previous example (Figure 21) can be represented differently as below. All the following method calls have the same effect.

```
m_clientStub.cast(due, myself.getCurrentSession(), myself.getCurrentGroup());
m_clientStub.cast(due, myself.getCurrentSession(), myself.getCurrentGroup(), 0,
0);
m_clientStub.cast(due, myself.getCurrentSession(), myself.getCurrentGroup(),
CMInfo.CM_STREAM, 0);
```

Figure. Different form of the cast method

- Blocking/Nonblocking communication channels

When sending an event, CM can use a default non-blocking communication channel or a blocking communication channel. A blocking or non-blocking channel can be added or removed by the CM additional channel service. If the CM application has both the blocking and non-blocking channels, it can choose one of them when it sends an event by using the send method of which synopsis is described below.

```
boolean send(CMEvent cme, String strTarget, int opt, int nChNum, boolean isBlock)
boolean send(CMEvent cme, String strTarget, int opt, int nSendPort, int nRecvPort, boolean isBlock)
```

Figure. send method with the blocking/non-blocking channel parameter.

In the first *send()* method, the last parameter (*isBlock*) is a Boolean value indicating that this method uses a blocking channel or a non-blocking channel with the channel key (*nChNum*). If the parameter is true, the blocking channel is used. Otherwise, the non-blocking channel is used. The other send

methods without the blocking/non-blocking parameter use the non-blocking communication channel that is the default communication in CM.

Comparing the first method, the second *send()* method has one additional parameter (nRecvPort) that is the receiver port number. This method can be called only in the following conditions. First, this method uses only the datagram (udp) channel. Therefore, the opt parameter must be always set to CMInfo.CM_DATAGRAM (or 1). Second, this method can send an event to a CM node of which a target port number (that is not the default port number) is known by the sender. Unlike the other *send()* methods, this method cannot use the internal forwarding scheme of CM.

7.3 Receiving a CM event

7.3.1 Asynchronous reception with the default non-blocking channel

In the initialization phase of a CM application, we have already set an event handler object to the CM stub. The event handler has the role of receiving every incoming CM events. If the event handler catches a pre-defined CM event, it identifies the event with its event type and ID values by calling the *getType* and *getID* methods of the received event. However, for a user-defined event such as the *CMDummyEvent*, it is just identified only with the event type because the further identification of the *CMDummyEvent* event is responsible for the developer. Probably, an event ID can be included in the string field. At the event handler, each field of the *CMDummyEvent* can be retrieved by the get methods as described below.

Event Type		CMInfo.CM_DUMMY_EVENT	
Event field	Field Data type	Field definition	Get method
Session name	String	A current session of the sending user	getHandlerSession()
Group name	String	A current group of the sending user	getHandlerGroup()
String message	String	A string value	getDummyInfo()

Table. CMDummyEvent fields and get methods

The following codes are the part of the event handler which catches an event of the *CMDummyEvent* class and prints out the received fields of the event.

```
…
public void processEvent(CMEvent cme) {
```

```
        switch(cme.getType())
        {
        case CMInfo.CM_DUMMY_EVENT:
                processDummyEvent(cme);
                break;
        default:
                return;
        }
}

private void processDummyEvent(CMEvent cme)
{
        CMDummyEvent due = (CMDummyEvent) cme;
        System.out.println("session("+due.getHandlerSession()+"),
group("+due.getHandlerGroup()+")");
        System.out.println("dummy msg: "+due.getDummyInfo());
        return;
}
```

Figure. Receiving a CMDummyEvent event

7.3.2 Synchronous reception with the blocking channel

To receive a CM event synchronously, the *CMStub* class provides following methods.

```
CMEvent receive(SocketChannel sc)
CMEvent receive(DatagramChannel dc)
```

Figure. receive methods.

The *receive* methods receive a CM event using a blocking socket/datagram channel. When an application calls the receive method, the application blocks its execution until it receives a CM event through the channel parameter. When an event is received, the method returns the event and the application resumes its execution.

The application (server or client) can add a blocking socket or datagram channel by the *CMClientStub.addBlockSocketChannel()* or *CMStub.addBlockDatagramChannel()* method. Furthermore, an application can retrieve its blocking socket or datagram channels with *CMClientSutb.getBlockSocketChannel()* or *CMStub.getBlockDatagramChannel()* method.

7.4 Synchronously Sending and Receiving CM events

Communication between a client and a server in the client-server model mostly follows a request-reply pattern. In the request-reply pattern, the client requests a service from the server, and the

server replies to the request. The communication type between the client and server is divided into a synchronous and an asynchronous communication as shown in the following Figure.

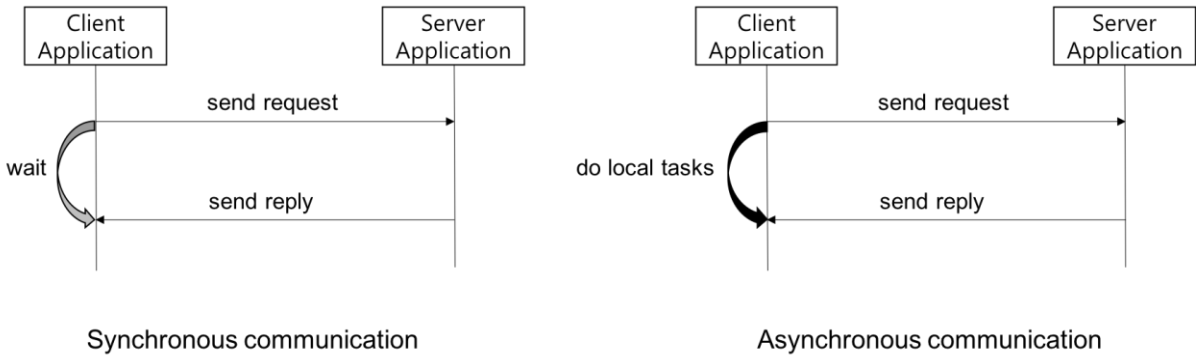


Figure. Synchronous vs. Asynchronous Communication.

In the synchronous communication, after the client sends a service request to the server, it suspends the execution and waits until the server replies to the request. When it receives the reply from the server, the client resumes the execution. In the asynchronous communication, the client does not wait the server reply after it sends the request. Instead, the client immediately continues its execution. For example, the client can continue to receive user's local input while the server deals with the request. To this end, the asynchronous communication requires a mechanism where the client separates the reception of the server reply and the execution of other local tasks.

An application may predetermine a communication method between the synchronous and the asynchronous communication according to the application requirements. However, it would be better if the application can selectively choose a communication method according to the service type and its usage context. The main criterion to determine a communication method is whether a client has other tasks to do while it is waiting for the response to the service request or not. The synchronous communication is a simple method if the client can perform the next task only after it receives the reply of the request from the server. On the other hand, if the client has other tasks to perform while it is waiting for the reply, it is more efficient to use the asynchronous communication. For example, a client can request a file-transfer service from a server in the synchronous or asynchronous manner. If the client has only the next task and the task is associated with the requested file, it is appropriate to request a file synchronously. On the contrary, if the client has another task that is independent from the file-transfer request, it is suitable to request the file asynchronously.

7.4.1 *sendrecv* method

```
public CMEvent sendrecv(CMEvent cme, String strReceiver, int nWaitEventType, int nWaitEventID, int nTimeout)
```

Figure. Synopsis of sendrecv().

The *sendrecv()* method sends a CM event and receive a reply event. This method is used when a sender node needs to synchronously communicates with a receiver through the default non-blocking communication channel.

The *cme* parameter is the event to be sent. The *strReceiver* parameter is the target name. The target node can be a server or a client. If the target is a client, the event and its reply event are delivered through the default server. The nWaitEventType and nWaitEventID parameters are the waited event type and ID of the reply event from *strReceiver*. The *nTimeout* parameter is the maximum time to wait in milliseconds. If *nTimeout* is greater than 0, the main thread is suspended until the timeout time elapses. If *nTimeout* is 0, the main thread is suspended until the reply event arrives without the timeout.

The return value of this method is a reply event if it is successfully received, or null otherwise.

Two CM nodes conduct the synchronous communication as follows:

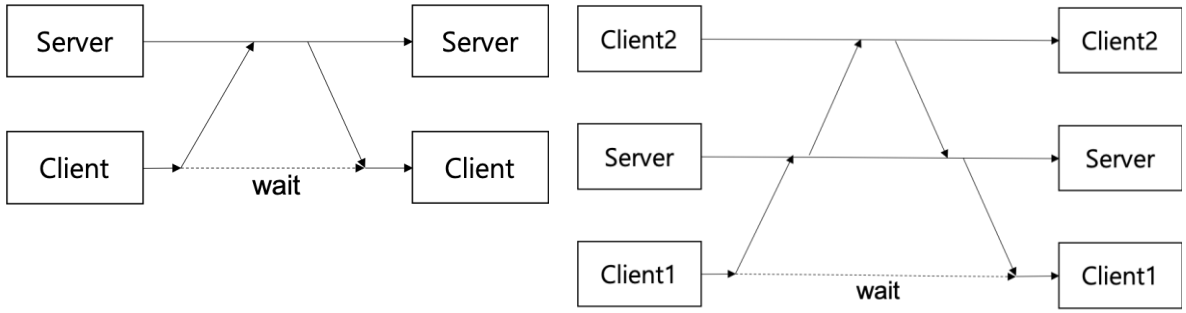
1. A sender node sends an event to a receiver.

2. The sender waits until the receiver sends a reply event.

3. The sender receives the reply event.

In the step 2, the application main thread suspends its execution after it calls this method. However, the sender still can receive events because CM consists of multiple threads, and the other threads can deal with the reception and process of events.

The communication supported by the *sendrecv()* method is an one-to-one synchronous communication where a sender and a receiver synchronously communicate with each other. The synchronous communication can be divided into two types: directly and indirectly synchronous communications. As shown in the following figure, if the communication is conducted between a client and a server, it is called directly synchronous communication. If the communication is conducted between a client and another client, it is called indirectly synchronous communication [9].



(a) Directly synchronous communication

(b) Indirectly synchronous communication

Figure. Directly vs. Indirectly Synchronous Communications.

The following code snippet show an example of using the *sendrecv()* method where the sender client sends a CM user event and it receives another user event as the reply event. The details of the CM user event are described in another section of this document.

```
CMUserEvent ue = new CMUserEvent();
CMUserEvent rue = null;
String strTargetName = null;

// a user event: (id, 111) (string id, "testSendRecv")
// a reply user event: (id, 222) (string id, "testReplySendRecv")
// create a user event
ue.setID(111);
ue.setStringID("testSendRecv");
// get target name
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("user event to be sent: (id, 111), (string id, ₩"testSendRecv₩")");
System.out.println("reply event to be received: (id, 222), (string id, ₩"testReplySendRecv₩")");
System.out.print("Target name(empty for ₩"SERVER₩"): ");
try {
        strTargetName = br.readLine().trim();
} catch (IOException e) {
        e.printStackTrace();
        return;
}

if(strTargetName.isEmpty())
        strTargetName = "SERVER";
rue = (CMUserEvent) m_clientStub.sendrecv(ue, strTargetName, CMInfo.CM_USER_EVENT,
```

```
222, 10000);

if(rue == null)
	System.err.println("The reply event is null!");
else
	System.out.println("Received reply event from ["+rue.getSender()+"]: (type,
"+rue.getType()+"), (id, "+rue.getID()+"), (string id, "+rue.getStringID()+")");
```

Figure. Usage example of sendrecv().

The following code snippet shows the receiver of the *sendrecv()* method. If the receiver receives the pre-defined event from the sender, it sends back the pre-defined reply event to the sender.

```
// The event handler of a CM node receives the CM user event sent by the sendrecv() method

public void processEvent(CMEvent cme) {
	switch(cme.getType())
	{
	…
	case CMInfo.CM_USER_EVENT:
		processUserEvent(cme);
		break;
	…
	}
}
private void processUserEvent(CMEvent cme)
{
	CMUserEvent ue = (CMUserEvent) cme;
	if(ue.getStringID().equals("testSendRecv"))
	{
		if(!m_clientStub.getMyself().getName().equals(ue.getReceiver()))
			return;
		CMUserEvent rue = new CMUserEvent();
		rue.setID(222);
		rue.setStringID("testReplySendRecv");
		boolean ret = m_clientStub.send(rue, ue.getSender());
		if(ret)
			System.out.println("Sent reply event: (id, "+rue.getID()+"), (string id,
"+rue.getStringID()+")");
		else
```

```
                System.err.println("Failed to send the reply event!");
            }
}
```

Figure. Example of sendrecv() receiver.

7.4.2 *castrecv* method

```
public CMEvent[] castrecv(CMEvent event, String strSessionName, String strGroupName, int
nWaitedEventType, int nWaitedEventID, int nMinNumWaitedEvents, int nTimeout)
```

Figure. Synopsis of castrecv().

The castrecv() method sends a CM event and receive multiple reply events. This method is used when a sender node needs to synchronously communicates with multiple receivers (CM group members, session members, or all login users) through the default non-blocking communication channel.

The event parameter is the event to be sent. The strSessionName parameter is the target session name. If this parameter is null, it implies all sessions. If strGroupName is not null, this parameter must not be null, either. The strGroupName is the target group name. If this parameter is null, it implies all groups. The nWaitedEventType and the nWaitedEventID parameters are the waited event type and ID of the reply events. The nMinNumWaitedEvents parameter is the minimum number of waited events. The sender waits until it receives at least nMinNumWaitedEvents events. The nTimeout parameter is the maximum time to wait in milliseconds. If nTimeout is greater than 0, the main thread is suspended until the timeout time elapses. If nTimeout is 0, the main thread is suspended until the all reply events arrives without the timeout.

The return value of this method is an array of reply CM events if the minimum number (nMinNumWaitedEvents) of reply events are successfully received, or null otherwise. The size of the array can be greater than nMinNumWaitedEvents.

A sender node and multiple receiver nodes conduct the synchronous communication as follows:

1. A sender node sends an event to receivers that are specified by a session name and a group name.

2. The sender waits until at least the given minimum number of receivers send reply events.

3. The sender receives an array of received reply events.

In the step 2, the application main thread suspends its execution after it calls this method. However, the sender still can receive events because CM consists of multiple threads, and the other threads can deal with the reception and process of events.

The communication supported by the castrecv() method is an one-to-many synchronous communication where a sender and multiple receivers synchronously communicate with each other. In this method, the sender can be a client or a server, but the receivers should be clients. If there are n-1 receivers (and one sender), we call this kind of communication as n-clients (indirect) synchronous communication as shown in the following figure.

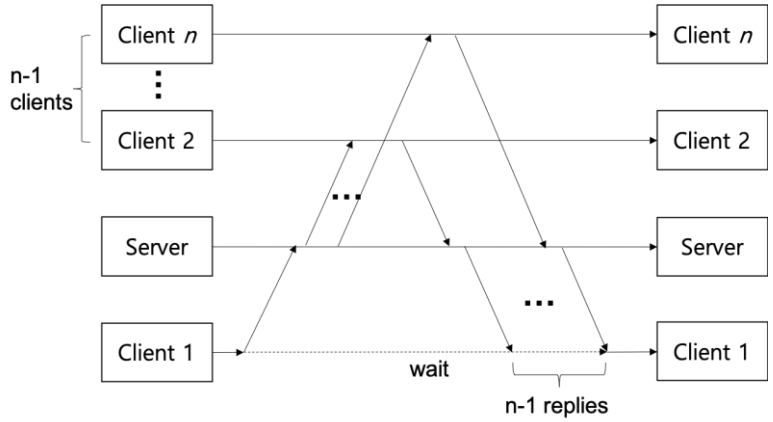


Figure. n-clients (indirect) synchronous communication.

The following code snippet show an example of using the *castrecv()* method where the sender client sends a CM user event to multiple clients and it receives another user event as the reply event.

```
CMUserEvent ue = new CMUserEvent();
CMEvent[] rueArray = null;
String strTargetSession = null;
String strTargetGroup = null;
String strMinNumReplyEvents = null;
int nMinNumReplyEvents = 0;
int nTimeout = 10000;

// a user event: (id, 112) (string id, "testCastRecv")
// a reply user event: (id, 223) (string id, "testReplyCastRecv")

// set a user event
ue.setID(112);
ue.setStringID("testCastRecv");
```

```
// set event target session and group
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
try {
        System.out.print("Target session(empty for null): ");
        strTargetSession = br.readLine().trim();
        System.out.print("Target group(empty for null): ");
        strTargetGroup = br.readLine().trim();
        System.out.print("Minimum number of reply events(empty for 0): ");
        strMinNumReplyEvents = br.readLine().trim();
} catch (IOException e) {
        e.printStackTrace();
        return;
}

if(strTargetSession.isEmpty())
        strTargetSession = null;
if(strTargetGroup.isEmpty())
        strTargetGroup = null;
if(strMinNumReplyEvents.isEmpty())
        strMinNumReplyEvents = "0";
try {
        nMinNumReplyEvents = Integer.parseInt(strMinNumReplyEvents);
}catch(NumberFormatException e) {
        e.printStackTrace();
        System.err.println("Wrong number format!");
        return;
}

rueArray = m_clientStub.castrecv(ue, strTargetSession, strTargetGroup,
CMInfo.CM_USER_EVENT, 223, nMinNumReplyEvents, nTimeout);

if(rueArray == null)
{
        System.err.println("Error in castrecv()!");
        return;
}

System.out.println("Number of received reply events: "+rueArray.length);
```

```
System.out.print("Reply from: ");
for(int i = 0; i < rueArray.length; i++)
        System.out.print(rueArray[i].getSender()+" ");
System.out.println();
```

Figure. Usage example of castrecv().

The following code snippet shows the receiver of the *castrecv()* method. If the receiver receives the pre-defined event from the sender, it sends back the pre-defined reply event to the sender.

```
// The event handler of a CM node receives the CM user event sent by the castrecv() method

public void processEvent(CMEvent cme) {
        switch(cme.getType())
        {
        …
        case CMInfo.CM_USER_EVENT:
                processUserEvent(cme);
                break;
        …
        }
}
private void processUserEvent(CMEvent cme)
{
        CMUserEvent ue = (CMUserEvent) cme;
        if(ue.getStringID().equals("testCastRecv"))
        {
                CMUserEvent rue = new CMUserEvent();
                rue.setID(223);
                rue.setStringID("testReplyCastRecv");
                boolean ret = m_clientStub.send(rue, ue.getSender());
                if(ret)
                        System.out.println("Sent reply event: (id, "+rue.getID()+"), (sting id,
"+rue.getStringID()+")");
                else
                        System.err.println("Failed to send the reply event!");
        }
}
```

Figure. Example of castrecv() receiver.

8. CM user event

In a CM application, a developer can define the simple format of a user-defined event using the *CMDummyEvent* class. However, this event type has the limitation that the developer should define the required event format in the single string field of the *CMDummyEvent*. To support the more flexible format of a user-defined event, CM provides another CM event class, *CMUserEvent*. Using this event class, a developer can define an event field with a field data type, a field name and its value. A created event field can be added to the instance of the *CMUserEvent* class. A *CMUserEvent* event which has its own event fields is identified by a String identifier. The name of an identifier is also defined by a developer. The required methods to set and get the identifier and event fields are described below.

Event type	CMInfo.CM_USER_EVENT
Methods	Usage
void setStringID(String id)	Set a String ID of this event
void setEventField(int type, String fName, String fValue)	add an event field
void setEventBytesField(String fName, int byteNum, byte[] bytes)	add an event field which is a byte array
String getStringID()	Get a String ID of this event
String getEventField(int type, String fName)	Get the value of an event field
byte[] getEventBytesField(String fName)	Get the byte array of an event field

Table. CMUserEvent class

The *setStringID* method is used to define an ID of a user-defined event. A developer should note that the data type of an ID is String rather than *int*. Although the pre-defined CM events use an ID of the *int* type, the *CMUserEvent* class uses the String type because it gives more readability for a user-defined event. The ID of a user event can be retrieved by calling the *getStringID* method.

To set an event field in a user event, the *setEventField* method is used. This method requires three parameters: a data type, a field name, and a field value. For the data type of a field, CM provides six primitive types which are named as constant values as shown below.

Field data type	Matching Java data type
CMInfo.CM_INT	int
CMInfo.CM_LONG	long
CMInfo.CM_FLOAT	float

CMInfo.CM_DOUBLE	double
CMInfo.CM_CHAR	char
CMInfo.CM_STR	String

Table. Field data type of CMUserEvent event

The field data type and the field name parameters are required for identifying different event fields in a user event. The last parameter is the field value. This value must always be given as the String type no matter which data type is used for this event field. In order to retrieve a field value, the *getEventField* method can be called. This method needs the field data type and the field name as parameters and returns the value of the corresponding event field. Because the return value is the String type, a developer should transform it to an original data type if required.

The *setEventBytesField* method is used to set an event field which is an arbitrary bytes stream. For example, the read bytes from a file can be an event field. This method requires three parameters. The first parameter is the field name which identifies the event field of bytes stream. The second parameter is the number of bytes of the bytes stream. The last parameter is a byte array which contains the bytes value. The byte array can be retrieved by the *getEventBytesField* method. With only the file name parameter, this method returns the bytes value as a byte array.

The following example shows how a CM client creates a sample user event. The ID of the event is "testID" and six event fields with different data types are set to the event. The client sends the created event to the default server.

```
…
CMUserEvent ue = new CMUserEvent();
int nField = 1;
long lField = 2;
float fField = 3.0f;
double dField = 4.0;
char cField = 'a';
String strField = "test string";
System.out.println("====================== test CMUserEvent");
ue.setStringID("testID");
ue.setEventField(CMInfo.CM_INT, "intField", String.valueOf(nField));
ue.setEventField(CMInfo.CM_LONG, "longField", String.valueOf(lField));
ue.setEventField(CMInfo.CM_FLOAT, "floatField", String.valueOf(fField));
ue.setEventField(CMInfo.CM_DOUBLE, "doubleField", String.valueOf(dField));
ue.setEventField(CMInfo.CM_CHAR, "charField", String.valueOf(cField));
ue.setEventField(CMInfo.CM_STR, "strField", strField);
m_clientStub.send(ue, "SERVER");
```

Figure. Example of CMUserEvent creation

A CM application can catch a user event of the *CMUserEvent* type like other CM events. The

following example shows how the default server catches the user event sent by the previous example and prints out the field values.

```
…
public void processEvent(CMEvent cme) {
        switch(cme.getType())
        {
        case CMInfo.CM_USER_EVENT:
                processUserEvent(cme);
                break;
        default:
                return;
        }
}

private void processUserEvent(CMEvent cme)
{
        CMUserEvent ue = (CMUserEvent) cme;
        System.out.println("event ID: "+ue.getStringID());
        nField = Integer.parseInt( ue.getEventField(CMInfo.CM_INT,
"intField") );
        System.out.println("intField: "+ nField);
        lField = Long.parseLong( ue.getEventField(CMInfo.CM_LONG,
"longField") );
        System.out.println("longField: "+ lField);
        fField = Float.parseFloat( ue.getEventField(CMInfo.CM_FLOAT,
"floatField") );
        System.out.println("floatField: "+ fField);
        dField = Double.parseDouble( ue.getEventField(CMInfo.CM_DOUBLE,
"doubleField") );
        System.out.println("doubleField: "+ dField);
        cField = ue.getEventField(CMInfo.CM_CHAR, "charField").charAt(0);
        System.out.println("charField: "+ cField);
        strField = ue.getEventField(CMInfo.CM_STR, "strField");
        System.out.println("strField: " + strField);
}
```

Figure. Receiving a CMUserEvent event

9. Management of additional communication channels

When an application initializes CM, the application can use default TCP and UDP communication channels. If the communication architecture is set to the *CM_PS*, an application can also use default multicast channel which is assigned to a group. In addition to the default channels, an application sometimes might need more than one communication channel according to its requirement. To support this requirement, CM also provides a developer with APIs for managing additional channels.

9.1 Additional stream (TCP) channels

Unlike the other channel types, only the client CM can add and remove a TCP channel because the TCP server cannot request the connection establishment. Therefore, the CM client stub provides the management APIs. To add a TCP channel, a client can call one of the following methods.

```
boolean addNonBlockSocketChannel(int nChKey, String strServer)
SocketChannel syncAddNonBlockSocketChannel(int nChKey, String strServer)
boolean addBlockSocketChannel(int nChKey, String strServer)
SocketChannel syncAddBlockSocketChannel(int nChKey, String strServer)
```

Figure. APIs of adding a (TCP) socket channel

The *addNonBlockSocketChannel* method adds asynchronously a non-blocking (TCP) socket channel to a server. The *nChKey* parameter is the channel key that must be greater than 0, because the key 0 is occupied by the default TCP channel. The *strServer* parameter is the server name to which the client adds a TCP channel. For example, the name of the default server is 'SERVER'. This method returns true if the socket channel is successfully created at the client and requested to add the (key, socket) pair to the server. Otherwise, it returns false.

Although this method returns the reference to the valid socket channel at the client, it is unsafe for the client to use the socket before the server also adds the relevant channel information. The establishment of a new non-blocking socket channel at both sides (the client and the server) completes only when the client receives the ack event (*CMSessionEvent.ADD_NONBLOCK_SOCKET_CHANNEL_ACK*) from the server and the return code in the event is 1. The client event handler can catch the ack event, and the detailed event fields are described below:

Event type	CMInfo.CM_SESSION_EVENT
Event ID	CMSessionEvent.ADD_NONBLOCK_SOCKET_CHANNEL_ACK
Event field	Get method
Channel name (server name)	getChannelName()
Channel key	getChannelNum()
Return code	getReturnCode()

The *syncAddNonBlockSocketChannel* method adds synchronously a non-blocking (TCP) socket channel to a server. The parameters are the same as those of the *addNonBlockSocketChannel* method. This method returns true if the socket channel is successfully created both at the client and the server. Otherwise, it returns false.

The *addBlockSocketChannel* method adds asynchronously a blocking (TCP) socket channel to a server. The parameters are the same as those of the *addNonBlockSocketChannel* method except the *nChKey* parameter. The *nChKey* parameter of this method can be 0 because the CM does not have a default blocking socket channel that would occupy the channel key 0. This method returns a reference to the socket channel if it is successfully created at the client, or null otherwise.

Although this method returns the reference to the valid socket channel, the socket channel at the server side is always created as a non-blocking mode first due to the intrinsic CM architecture of event-driven asynchronous communication. The server sends the acknowledgement message after the non-blocking channel is changed to the blocking channel. It is unsafe for the client use its socket channel before the channel is changed to the blocking mode at the server. The establishment of a new blocking socket channel at both sides (the client and the server) completes only when the client receives the ack event (*CMSessionEvent.ADD_BLOCK_SOCKET_CHANNEL_ACK*) from the server, and the return code in the event is 1. The client event handler can catch the ack event, and the detailed event fields are described below:

Event type	CMInfo.CM_SESSION_EVENT
Event ID	CMSessionEvent.ADD_BLOCK_SOCKET_CHANNEL_ACK
Event field	Get method
Channel name (server name)	getChannelName()
Channel key	getChannelNum()
Return code	getReturnCode()

The *syncAddBlockSocketChannel* method adds synchronously a blocking (TCP) socket channel to a server. The parameters are the same as those of the *addBlockSocketChannel* method. This method returns a reference to the socket channel if it is successfully created both at the client and the server, or null otherwise.

CM maintains blocking channels separately from non-blocking channels. CM cannot put them to the selector because they are all the blocking mode. Therefore, developers can explicitly get a blocking socket channel by calling the *getBlockSocketChannel()* method to synchronously send or receive CM events or other byte messages.

To remove a TCP channel, a client can call the following methods.

```
boolean removeNonBlockSocketChannel(int nChKey, String strServer)
boolean removeBlockSocketChannel(int nChKey, String strServer)
boolean syncRemoveBlockSocketChannel(int nChKey, String strServer)
```

Figure. APIs of deleting a (TCP) socket channel

The *removeNonBlockSocketChannel* method removes a non-blocking (TCP) socket channel from a server. The parameters are the same as those of the *addNonBlockSocketChannel* method. If the default channel (key 0) is removed, the result behavior of the CM is not defined. This method returns true if the client successfully closes and removes the channel, or false otherwise. If the client removes the non-blocking socket channel, the server CM detects the disconnection and removes the channel at the server side as well.

The *removeBlockSocketChannel* method removes asynchronously the blocking socket (TCP) channel. The parameters are the same as those of the *addBlockSocketChannel* method. This method returns true if the client successfully requests the removal of the channel from the server, or false otherwise. The blocking socket channel is closed and removed only when the client receives the ack event from the server.

This method does not immediately remove the requested channel for safe and smooth close procedure between the client and the server. Before the removal of the client socket channel, the client first sends a request CM event to the server that then prepares the channel disconnection and sends the ack event (*CMSessionEvent.REMOVE_SOCKET_CHANNEL_ACK*) back to the client. The client closes and removes the target channel only if it receives the ack event and the return code is 1. The client event handler can catch the event in order to figure out the result of the removal request. The detailed event fields are described below:

Event type	CMInfo.CM_SESSION_EVENT
Event ID	CMSessionEvent.REMOVE_BLOCK_SOCKET_CHANNEL_ACK
Event field	Get method
Channel name (server name)	getChannelName()
Channel key	getChannelNum()
Return code	getReturnCode()

The *syncRemoveBlockSocketChannel* method removes synchronously the blocking socket (TCP) channel. The parameters are the same as those of the *syncAddBlockSocketChannel* method. This method returns true if the client successfully closed and removed the channel, or false otherwise. The blocking socket channel is closed and removed only when the client receives the ack event

from the server.

This method does not immediately remove the requested channel for safe and smooth close procedure between the client and the server. Before the removal of the client socket channel, the client first sends a request CM event to the server that then prepares the channel disconnection and sends the ack event (*CMSessionEvent.REMOVE_SOCKET_CHANNEL_ACK*) back to the client. The client closes and removes the target channel only if it receives the ack event and the return code is 1.

9.2 Additional datagram (UDP) channels

The CM stub provides methods to add and remove an additional UDP channel. Both a server and a client application can call these methods. The synopses of these methods are described below.

```
DatagramChannel addNonBlockDatagramChannel(int nChPort)
boolean removeNonBlockDatagramChannel(int nChPort)

DatagramChannel addBlockDatagramChannel(int nChPort)
boolean removeBlockDatagramChannel(int nChPort)
DatagramChannel getBlockDatagramChannel(int nChPort)
```

Figure. APIs of adding/removing UDP channel

The *addNonBlockDatagramChannel()* method adds a non-blocking datagram (UDP) channel to this CM node. A developer must note that the given port number (nChPort) is unique and different from that of the default channel in the configuration file. A UDP channel is identified by the port number as an index. The non-blocking datagram channel is also added to the selector of CM so that it can detect any incoming CM event asynchronously.

The *removeNonBlockDatagramChannel()* method removes a non-blocking datagram (UDP) channel from this CM node. Like the stream (TCP) channel, a developer should be careful not to remove the default channel.

The *addBlockDatagramChannel()* method adds a blocking datagram (UDP) channel to this CM node. CM maintains blocking channels separately from non-blocking channels. CM cannot put them to the selector because they are all the blocking mode. Therefore, developers can explicitly get a blocking datagram channel by calling the *getBlockDatagramChannel()* method to synchronously send or receive CM events or other byte messages.

The *removeBlockDatagramChannel()* method removes a blocking datagram (UDP) channel from this

CM node.

9.3 Additional multicast channels

To add and remove an additional multicast channel, the CM stub provides the *addMulticastChannel* and *removeAdditionalMulticastChannel* methods, respectively. Both a server and a client application can call these methods. The synopses of these methods are described below.

```
void addMulticastChannel(String strSession, String strGroup, String
strChAddress, int nChPort)

void removeAdditionalMulticastChannel(String strSession, String strGroup)
```

Figure. APIs of adding/removing multicast channel

The *addMulticastChannel* method requires four parameters to define a new multicast channel. The first and second parameters specify a session and group to which a new channel is assigned, because a multicast channel is always mapped to a specific group in a session. The third and fourth parameters are the multicast address and the port number of the new channel. A developer must note that the given address and the port number are unique and different from that of the default channel in a corresponding session configuration file. A multicast channel is identified by the four-tuple: a session name, a group name, a multicast address, and a port number. They are also the parameter of the *removeAdditionalDatagramChannel* method. Like the other cases, a developer should be careful not to remove the default channel.

10. Current client information

A CM client application can access information on its current states such as the client name, the current session, the current group, the current connection state with the default server, and so on. Such information is contained in an instance of the *CMUser* class which is a member variable of the *CMInteractionInfo* class. The reference of the *CMUser* object can be retrieved by the *getMyself* method in the *CMInteractionInfo* class which is a member of the CMInfo class. The reference to the instance of the *CMInteractionInfo* class can be got by calling the *getInteractionInfo* method of the *CMInfo* class. The reference to the instance of the *CMInfo* class can be retrieved by the *getCMInfo* method of the CM stub module.

From the reference to the *CMUser* class, a client application can get its state information as described below.

Method	Description
String getName()	To return the name of this client.
String getHost()	To return the host address of this client.
String getUDPPort()	To return the default UDP port number of this client.
String getCurrentSession()	To return the name of current session to which this client joins.
String getCurrentGroup()	To return the name of current group to which this client joins.
int getState()	To return the current connection state of this client to the default server. 1: CMInfo.CM_INIT 2: CMInfo.CM_CONNECT 3: CMInfo.CM_LOGIN 4: CMInfo.CM_SESSION_JOIN

Table. Member methods of the CMUser class

The following example shows sample codes where a client application prints out its current information.

```
// A client has already initialized CM.
CMInteractionInfo interInfo = m_clientStub.getCMInfo().getInteractionInfo();
CMUser myself = interInfo.getMyself();
System.out.println("------ for the default server");
System.out.println("name("+myself.getName()+"),
session("+myself.getCurrentSession()+"), group("+myself.getCurrentGroup()+"),
udp port("+myself.getUDPPort()+"), state("+myself.getState()+").");
```

Figure. Printing out current client states

11. File transfer management

CM applications that directly connect to each other can exchange a file with the *CMStub* class that is the parent class of the *CMClientStub* and the *CMServerStub* classes. In the client-server architecture, CM allows a client or a server to send (push) or receive (pull) a file to/from a server or a client. That is, any CM node can transfer a file to another CM node. If a client sends a file to another client, it is called P2P file transfer. When CM is initialized by an application, the default directory is configured by the path information that is set in the configuration file (the *FILE_PATH* field). If the default directory does not exist, CM creates it. If the FILE_PATH field is not set, the default path is set to the current working directory (“.”).

If the file transfer is requested, a sender (the server or the client) searches for the file in the default file path. If a client receives a file, CM stores the file in this file path. If a server receives a file, CM stores the file in a sub-directory of the default path. The sub-directory name is a sender (client) name.

A CM application can change the default file path by the *setFilePath* method in the *CMStub*. The synopsis of this method is described below.

```
void setFilePath(String strFilePath)
```

Figure. setFilePath method

The parameter is the new default directory information. The following example shows how a CM client changes the file path which is given by a user.

```
// A client has already initialized CM.
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("====== set file path");
String strPath = null;
System.out.print("file path: ");
try {
        strPath = br.readLine();
} catch (IOException e) {
        e.printStackTrace();
}
m_clientStub.setFilePath(strPath);
```

Figure. Setting a new file path

The CM file transfer manager supports two transmission modes: a push and pull modes. Detailed CM file-transfer service is described in [10].

11.1 Requesting (pulling) a file

In the pull mode, a CM application requests a file to another remote CM application. For example, a CM client can request that a CM server sends a file. A file is requested by the *requestFile* method of the *CMStub* class and its synopsis is described below.

```
boolean requestFile(String strFileName, String strFileOwner)
boolean requestFile(String strFileName, String strFileOwner, byte
byteFileAppend)
```

Figure. requestFile methods

The first *requestFile* method requires two parameters. The first parameter is the requested file name.

The second parameter is the name of a file owner.

The second requestFile method requires three parameters. The first and the second parameters are the same as those of the first method. The last parameter, *byteFileAppend*, is the file reception mode that specifies the behavior of the receiver if it already has the entire or part of the requested file. CM provides three reception modes which are default, overwrite, and append mode. The *byteFileAppend* parameter can be one of these three modes which are *CMInfo.FILE_DEFAULT(-1)*, *CMInfo.FILE_OVERWRITE(0)*, and *CMInfo.FILE_APPEND(1)*. If the *byteFileAppend* parameter is *CMInfo.FILE_DEFAULT*, the file reception mode is determined by the *FILE_APPEND_SCHEME* field of the CM configuration file. If the *byteFileAppend* parameter is set to one of the other two values, the reception mode of this requested file does not follow the CM configuration file, but this parameter value. The *CMInfo.FILE_OVERWRITE* is the overwrite mode where the receiver always receives the entire file even if it already has the same file. The *CMInfo.FILE_APPEND* is the append mode where the receiver skips existing file blocks and receives only remaining blocks.

The following example shows how a client requests a file to a server.

```
// A client has already initialized CM.
String strFileName = null;
String strFileOwner = null;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("====== request a file");
try {
	System.out.print("File name: ");
	strFileName = br.readLine();
	System.out.print("File owner(server name): ");
	strFileOwner = br.readLine();
} catch (IOException e) {
	e.printStackTrace();
}

m_clientStub.requestFile(strFileName, strFileOwner);
```

Figure. Requesting a file

If the requester calls the *requestFile()* method, the file owner receives the request event (*CMFileEvent.REQUEST_PERMIT_PULL_FILE*). The file owner then accepts or rejects to send the requested file automatically or manually. If the *PERMIT_FILE_TRANSFER* field of the owner's CM configuration file (cm-server.conf or cm-client.conf) is set to 1, the owner automatically accepts the request and proceeds the next step to send the file. If the *PERMIT_FILE_TRANSFER* field is set to 0, the owner application should notify CM of the answer of the request by calling *replyEvent(CMEvent, int)*. The second parameter of this method indicates the answer. If the parameter value is 1, the request is accepted. If the parameter is 0, the request is rejected. If the requested file does not exist

in the owner, the reply parameter is set to -1. The requester application can catch the reply event (*CMFileEvent.REPLY_PERMIT_PULL_FILE*) to check whether the request is accepted or not. The detailed information of the *REPLY_PERMIT_PULL_FILE* event is described below.

Event type		CMInfo.CM_FILE_EVENT	
Event ID		CMFileEvent.REPLY_PERMIT_PULL_FILE	
Event field	Field data type	Field definition	Get method
File sender	String	File sender name	getFileSender()
File receiver	String	File receiver name	getFileReceiver()
File name	String	File name	getFileName()
Return code	int	-1: the requested file does not exit. 0: the request is denied. 1: the request is accepted.	getReturnCode()
Attaching SNS content ID	int	>= 0: the requested file is an attachement of SNS content ID -1: the file is no attachment of SNS content	getContentID()

Table. CMFileEvent.REPLY_PERMIT_PULL_FILE event

```
…
public void processEvent(CMEvent cme) {
        switch(cme.getType())
        {
        ……
        case CMInfo.CM_FILE_EVENT:
                processFileEvent(cme);
                break;
        default:
                return;
        }
}

private void processFileEvent(CMEvent cme)
{
        CMFileEvent fe = (CMFileEvent) cme;
        switch(fe.getID())
        {
        ……
        case CMFileEvent.REPLY_PERMIT_PULL_FILE:
                if(fe.getReturnCode() == -1)
                {
                        System.err.print("["+fe.getFileName()+"] does not exist in
the owner!\n");
                }
                else if(fe.getReturnCode() == 0)
                {
                        System.err.print("["+fe.getFileSender()+"] rejects to send
```

```
file("+fe.getFileName()+").\n");
                }
                break;
        ……
        }
        return;
}
```

Figure. Checking out the result of the file transfer request

CM uses two strategies in the file-transfer service. If the *FILE_TRANSFER_SCHEME* field of the server configuration file (cm-server.conf) is set to 0, CM uses the default communication channel between a sender and a receiver to transfer a file. If both the sender and the receiver are clients in the client-server model, then the server relays all the events required to transfer a file. If the *FILE_TRANSFER_SCHEME* field is set to 1, CM uses a separate and dedicated communication channel and thread to transfer a file. The dedicated channel is directly connected between the sender and the receiver before the file-transfer task. If a receiver node is located in a different private network, a sender cannot make a dedicated channel and cannot transfer a file.

According to the file-transfer method, CM nodes exchanges different CM events. An easy way to figure out to which file-transfer method a CM event belongs is to check whether the event name ends with "*CHAN*" or "*CHAN_ACK*". If so, such an event belongs to the file-transfer using a dedicated channel; otherwise, the event belongs to the file-transfer using the default channel.

When the requested file is completely transferred, the sender CM sends *CMFileEvent.END_FILE_TRANSFER* or *CMFileEvent.END_FILE_TRANSFER_CHAN* events to the requester according to the file-transfer method. The requester can catch the completion event in the event handler if it needs to be notified when the entire file is transferred. The detailed information of the transfer completion event is described below.

Event type		CMInfo.CM_FILE_EVENT	
Event ID		CMFileEvent.END_FILE_TRANSFER or CMFileEvent.END_FILE_TRANSFER_CHAN	
Event field	Field data type	Field definition	Get method
File sender	String	File sender name	getFileSender()
File receiver	String	File receiver name	getFileReceiver()
File name	String	File name	getFileName()
File size	long	File size	getFileSize()
Attaching SNS content	int	>= 0: the requested file is an attachement of SNS content ID	getContentID()

ID		-1: the file is no attachment of SNS content	

11.2 Pushing a file

Unlike the pull mode, in the push mode, a CM application can send a file to another remote CM application. This method returns true if the permission request of pushing file is successfully notified to the receiver, or false otherwise. The synopsis of this method is described below.

```
boolean pushFile(String strFilePath, String strReceiver)
```

Figure. pushFile method

Like the *requestFile* method, the *pushFile* method requires two parameters. The first parameter is a path name of a file to be sent. The second parameter is a receiver name. The following example shows how a client sends a file to a server.

```
// A client has already initialized CM.
String strFilePath = null;
String strReceiver = null;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("====== push a file");
try {
	System.out.print("File path name: ");
	strFilePath = br.readLine();
	System.out.print("File receiver: ");
	strReceiver = br.readLine();
} catch (IOException e) {
	e.printStackTrace();
}

m_clientStub.pushFile(strFilePath, strReceiver);
```

Figure. Pushing a file

If the requester calls this method, the file receiver receives the request event (*CMFileEvent.REQUEST_PERMIT_PUSH_FILE*). The file receiver then accepts or rejects to receive the file automatically or manually. If the *PERMIT_FILE_TRANSFER* field of the receiver's CM configuration file (cm-server.conf or cm-client.conf) is set to 1, the receiver automatically accepts the request and proceeds the next step to receive the file. If the *PERMIT_FILE_TRANSFER* field is set to 0, the receiver application should notify CM of the answer of the request by calling *replyEvent(CMEvent, int)*. The second parameter of this method indicates the answer. If the parameter value is 1, the request is accepted. If the parameter is 0, the request is rejected. The requester application can catch the reply event (*CMFileEvent.REPLY_PERMIT_PUSH_FILE*) to check whether the request is accepted or not. The

following table shows the detailed information of the reply event.

Event type		CMInfo.CM_FILE_EVENT	
Event ID		CMFileEvent.REPLY_PERMIT_PUSH_FILE	
Event field	Field data type	Field definition	Get method
File sender	String	File sender name	getFileSender()
File receiver	String	File receiver name	getFileReceiver()
File path	String	File path name	getFilePath()
File size	long	File size	getFileSize()
Attaching SNS content ID	int	>= 0: the requested file is an attachement of SNS content ID -1: the file is no attachment of SNS content	getContentID()
Return code	int	0: the request is denied. 1: the request is accepted.	getReturnCode()

When the receiver receives the file completely from the requester (sender), the receiver CM sends *CMFileEvent.END_FILE_TRANSFER_ACK* or *CMFileEvent.END_FILE_TRANSFER_CHAN_ACK* events to the requester according to the file-transfer method. The requester can catch the completion event in the event handler if it needs to be notified when the receiver receives the entire file blocks. The detailed information of the completion ack event is shown below:

Event type		CMInfo.CM_FILE_EVENT	
Event ID		CMFileEvent.END_FILE_TRANSFER_ACK or CMFileEvent.END_FILE_TRANSFER_CHAN_ACK	
Event field	Field data type	Field definition	Get method
File sender	String	File sender name	getFileSender()
File receiver	String	File receiver name	getFileReceiver()
File name	String	File name	getFileName()
File size	long	File size	getFileSize()
Return code	int	1: the entire file successfully received 0: reception error at the receiver	getReturnCode()
Attaching SNS content ID	int	>= 0: the requested file is an attachement of SNS content ID -1: the file is no attachment of SNS content	getContentID()

11.3 Cancellation of the file transfer

Ongoing file transfer can be cancelled by calling *cancelPushFile* or *cancelPullFile* method of the *CMStub* class. The synopses of these methods are as follows.

```
boolean cancelPushFile(String strReceiver)
boolean cancelPullFile(String strSender)
```

The *cancelPushFile* method cancels sending (pushing) files to a receiver. The *strReceiver* parameter is the receiver name. A sender can cancel all of its sending tasks to the receiver by calling this method. The file pushing task can be cancelled regardless of the file transfer scheme that is determined by the *FILE_TRANSFER_SCHEME* field of the server CM configuration file. This method returns true if the cancellation is succeeded, or false otherwise. The cancellation is also notified to the receiver. The result of the receiver's cancellation is sent to the sender as the *CANCEL_FILE_SEND_ACK* or the *CANCEL_FILE_SEND_CHAN_ACK* event. The former event is sent if the file transfer service uses the default channel (,that is, if the *FILE_TRANSFER_SCHEME* field is 0 in the server CM configuration file), and the latter event is sent if the file transfer service uses the separate channel (, that is, if the *FILE_TRANSFER_SCHEME* field is 1 in the server CM configuration file). The detailed information of these events is described below.

Event type		CMInfo.CM_FILE_EVENT	
Event ID		CMFileEvent.CANCEL_FILE_SEND_ACK CMFileEvent.CANCEL_FILE_SEND_CHAN_ACK	
Event field	Field data type	Field definition	Get method
File sender	String	File sender name	getFileSender()
File receiver	String	File receiver name	getFileReceiver()
Return code	int	1: successfully cancelled at the receiver 0: cancellation error at the receiver	getReturnCode()

The *cancelPullFile* method cancels receiving (pulling) files from a sender. The *strSender* parameter is the sender name. A receiver can cancel all of its receiving tasks from the sender by calling this method. Unlike the cancellation of the file pushing task, the file pulling task can be cancelled only if the file transfer scheme is on using the separate channel (, that is, if the *FILE_TRANSFER_SCHEME*

field is 1 in the server CM configuration file). This method returns true if the cancellation is succeeded, or false otherwise. The cancellation is also notified to the sender. The result of the sender's cancellation is sent to the receiver as the *CANCEL_FILE_RECV_CHAN_ACK* event. The detailed information of this events is described below.

Event type		CMInfo.CM_FILE_EVENT	
Event ID		CMFileEvent.CANCEL_FILE_RECV_CHAN_ACK	
Event field	Field data type	Field definition	Get method
File sender	String	File sender name	getFileSender()
File receiver	String	File receiver name	getFileReceiver()
Return code	int	1: successfully cancelled at the sender 0: cancellation error at the sender	getReturnCode()

11.4 Measurement of the network throughput

A CM application can measure the incoming/outgoing network throughput from/to another CM application that is directly connected to the application. For example, a client can measure the network throughput from/to the server. The incoming network throughput of a CM node *A* from *B* measures how many bytes *A* can receive from *B* in a second. The outgoing network through of a CM node *A* to *B* measures how many bytes *A* can send to *B*. The application can measure its incoming/outgoing network throughput by calling the *measureInputThroughput* or *measureOutputThroughput* methods in the *CMStub* class. The synopses of these methods are described below:

```
float measureInputThroughput(String strTarget)
float measureOutputThroughput(String strTarget)
```

Both methods require one parameter, *strTarget*, which specifies a target CM node with which the calling application measures the network throughput. The target CM node should be directly connected to the calling application. If the network throughput is successfully measured, these methods return the measurement value by the unit of Megabytes per second (MBps). If an error occurs, the return value is -1.

12. User management

CM also provides an application with user management functionalities. The user management includes the registration, deregistration, searching for a user. Because user profile information should be stored in DB, a developer must set the server configuration file to use the CM DB (the *DB_USE* and other relevant fields) in order to use the user management support of CM. A CM client can send a user management request to the default server, and the default server then processes the request using the CM DB. User management APIs are provided by the CM client stub module. The default server sends the result of a request as a CM event which can be caught by the client event handler as done before.

12.1 User registration

A user can be registered to CM by the *registerUser* method of the CM client stub. If a CM client is connected to the default server, it can call this method. CM uses the registered user information for the user authentication when a user logs in to the default server. The synopsis of the method is shown below.

```
void registerUser(String strName, String strPasswd)
```

Figure. User registration method

The *registerUser* method requires only two parameters: a user name (*strName*) and password (*strPasswd*). If the same user name already exists in CM DB, the registration fails. If the user name is uniquely specified, the registration succeeds. The following example shows how a client can simply request a user registration.

```
// A client has already initialized CM.
String strName = null;
String strPasswd = null;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("====== register a user");
try {
	System.out.print("Input user name: ");
	strName = br.readLine();
	System.out.print("Input password: ");
	strPasswd = br.readLine();
} catch (IOException e) {
	e.printStackTrace();
}
m_clientStub.registerUser(strName, strPasswd);
```

Figure. Example of user registration request

Whether the registration request is successful or not is set to a return code of a reply session event,

REGISTER_USER_ACK. If the request is successful, the reply event also contains the registration time at the server. The details of the *REGISTER_USER_ACK* event are described below.

Event Type		CMInfo.CM_SESSION_EVENT	
Event ID		CMSessionEvent.REGISTER_USER_ACK	
Event field	Field Data type	Field definition	Get method
Return code	Int	Result code of the request 1: succeeded 0: failed	getReturnCode()
User name	String	Requester user name	getUserName()
Creation time	String	Time to create the user at DB	getCreationTime()

Table. REGISTER_USER_ACK event

When the requesting client receives the *REGISTER_USER_ACK* event of the *CMSessionEvent* class, it can figure out the result of the request as the following example which is only relevant part of the client event handler.

```
…
private void processSessionEvent(CMEvent cme)
{
        CMSessionEvent se = (CMSessionEvent)cme;
        switch(se.getID())
        {
        case CMSessionEvent.REGISTER_USER_ACK:
                if( se.getReturnCode() == 1 )
                {
                        // user registration succeeded
                        System.out.println("User["+se.getUserName()+"]
successfully registered at time["+se.getCreationTime()+"].");
                        }
                else
                {
                        // user registration failed
                        System.out.println("User["+se.getUserName()+"] failed to
register!");
                }
                break;
        default:
                return;
        }
}
```

Figure. Handling the result of user registration

12.2 User deregistration

A user can cancel his/her registration from CM by the *deregisterUser* method of the CM client stub. If a client is connected to the default server, it can call this method. When requested, CM removes

the registered user information from the CM DB. The synopsis of the method is shown below.

```
void deregisterUser(String strName, String strPasswd)
```

Figure. User deregistration method

Like the *registerUser* method, the *deregisterUser* method requires only two parameters: a user name (*strName*) and password (*strPasswd*). If the given user name with the correct password exists in the CM DB, the deregistration request is successful. Otherwise, the request fails. The following example shows how a client can simply request a user deregistration.

```
// A client has already initialized CM.
String strName = null;
String strPasswd = null;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("====== Deregister a user");
try {
        System.out.print("Input user name: ");
        strName = br.readLine();
        System.out.print("Input password: ");
        strPasswd = br.readLine();
} catch (IOException e) {
        e.printStackTrace();
}
m_clientStub.deregisterUser(strName, strPasswd);
```

Figure. Example of user deregistration request

Whether the deregistration request is successful or not is set to a return code of a reply session event, *DEREGISTER_USER_ACK* as described below.

Event Type		CMInfo.CM_SESSION_EVENT	
Event ID		CMSessionEvent.DEREGISTER_USER_ACK	
Event field	Field Data type	Field definition	Get method
Return code	Int	Result code of the request 1: succeeded 0: failed	getReturnCode()
User name	String	Requester user name	getUserName()

Table. DEREGISTER_USER_ACK event

When the requesting client receives the *DEREGISTER_USER_ACK* event of the *CMSessionEvent* class, it can figure out the result of the request as the following example which is only relevant part of the client event handler.

```
…
private void processSessionEvent(CMEvent cme)
```

```
{
	CMSessionEvent se = (CMSessionEvent)cme;
	switch(se.getID())
	{
	case CMSessionEvent.DEREGISTER_USER_ACK:
		if( se.getReturnCode() == 1 )
		{
			// user deregistration succeeded
			System.out.println("User["+se.getUserName()+"]
successfully deregistered.");
		}
		else
		{
			// user registration failed
			System.out.println("User["+se.getUserName()+"] failed to
deregister!");
		}
		break;
	default:
		return;
	}
}
```

Figure. Handling the result of user deregistration

12.3 User search

A user can search for another user by the *findRegisteredUser* method of the CM client stub. If a client is connected to the default server, it can call this method. When requested, CM provides the basic profile of the target user such as a name and registration time. The synopsis of the method is shown below.

```
void findRegisteredUser(String strName)
```

Figure. User search method

The *findRegisteredUser* method requires only the user name parameter. If the given user name exists in the CM DB, the search request is successful. Otherwise, the request fails. The following example shows how a client can simply search for a user.

```
// A client has already initialized CM.
String strName = null;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("====== search for a registered user");
try {
	System.out.print("Input user name: ");
	strName = br.readLine();
} catch (IOException e) {
	e.printStackTrace();
}
m_clientStub.findRegisteredUser(strName);
```

Figure. Example of user search request

Whether the requested user is found or not is set to a return code of a reply session event, *FIND_REGISTERED_USER_ACK* as described below.

Event Type		CMInfo.CM_SESSION_EVENT	
Event ID		CMSessionEvent.FIND_REGISTERED_USER_ACK	
Event field	Field Data type	Field definition	Get method
Return code	Int	Result code of the request 1: succeeded 0: failed	getReturnCode()
User name	String	Requested user name	getUserName()
Creation time	String	Time to create the user at DB	getCreationTime()

Table. FIND_REGISTERED_USER_ACK event

When the requesting client receives the *FIND_REGISTERED_USER_ACK* event of the *CMSessionEvent* class, it can figure out the result of the request as the following example which is only relevant part of the client event handler.

```
…
private void processSessionEvent(CMEvent cme)
{
        CMSessionEvent se = (CMSessionEvent)cme;
        switch(se.getID())
        {
        case CMSessionEvent.FIND_REGISTERED_USER_ACK:
                if( se.getReturnCode() == 1 )
                {
                        System.out.println("User profile search succeeded:
user["+se.getUserName()+"], registration time["+se.getCreationTime()+"].");
                }
                else
                {
                        System.out.println("User profile search failed:
user["+se.getUserName()+"]!");
                }
                break;
        default:
                return;
        }
}
```

Figure. Handling the result of user search

13. Social Networking Service (SNS) management

Using CM, a developer can easily implement SNS-related functionalities such as the management

of friends and the upload and download of SNS content. In order to enable the SNS management of CM, a developer needs to set to use the CM DB (the *DB_USE* and other relevant fields) in the server configuration file. These functionalities are provided as the APIs of the CM client stub, and a client can call methods to request to the default server. In order to use all the SNS APIs, a client must log in to the default server. When the default server receives a SNS management request from a client, it sends the result of the request as a CM SNS event to the client, and the client can figure out the result by catching the received event.

13.1 Friend management

In a SNS application, it is normal to support the management of friends of a user. CM makes it easy for a client to add, delete, and get friend information. The following table shows the synopses of the relevant methods of the CM client stub.

```
void addNewFriend(String strFriendName)
void removeFriend(String strFriendName)
void requestFriendsList()
void requestFriendRequestersList()
void requestBiFriendsList()
```

Table. Methods for friend management

A client can call the *addNewFriend* method to add a new friend. This method requires one parameter which is a friend name. A client can add a user as its friend only if the user name has been registered to CM. When the default server receives the request for adding a new friend, it first checks if the friend is a registered user or not. If the friend is a registered user, the server adds it to the friend table of the CM DB as a friend of the requesting user. Otherwise, the request fails. In any case, the server sends the *ADD_NEW_FRIEND_ACK* event with a result code to the requesting client so that it can figure out the request result.

A client can delete a friend by calling the *removeFriend* method which has the same parameter as the *addNewFriend* method. When the default server receives the request for deleting a friend, it searches for the friend of the requesting user. If the friend is found, the server deletes the corresponding entry from the friend table. Otherwise, the request fails. The result of the request is sent to the requesting client as the *REMOVE_FRIEND_ACK* event with a result code. Actually the event fields of both the *ADD_NEW_FRIEND_ACK* and *REMOVE_FRIEND_ACK* are the same as described below.

Event Type		CMInfo.CM_SNS_EVENT	
Event ID		CMSNSEvent.ADD_NEW_FRIEND_ACK CMSNSEvent.REMOVE_FRIEND_ACK	
Event field	Field Data type	Field definition	Get method
Return code	int	Result code of the request 1: succeeded 0: failed	getReturnCode()
User name	String	The name of a requesting user	getUserName()
Friend name	String	The name of a friend to add or remove	getFriendName()

Table. ADD_NEW_FRIEND_ACK and REMOVE_FRIEND_ACK events

Different SNS applications use the concept of a friend in different ways. In some applications, a user can add another user in his/her friend list without the agreement of the target user. In other applications, a user can add a friend only if the other user accepts the friend request. CM supports such different policies of the friend management by methods that request different user lists. The *requestFriendsList* method requests the list of users whom the requesting user adds as his/her friends regardless of the acceptance of the others. The *reqeustFriendRequestersList* method requests the list of users who add the requesting user as a friend, but whom the requesting user has not added as friends yet. The *requestBiFriendsList* method requests the list of users who add the requesting user as a friend and whom the requesting user adds as friends. For example, three different friend relationships can be illustrated as a directed graph as shown in the following figure.

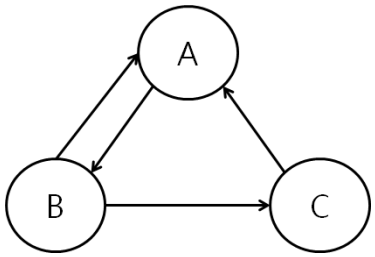


Figure. Different friend relationships

A node means a user, and an edge is a friend relationship. In the figure, there are three users, A, B, and C. The edge from the user A to the user B means that A adds B as its friend. If there are edges in both directions between two nodes, the corresponding users add each other as a friend. If the user A, B, and C call the above three methods, the result is described as below.

Method name	A result	B result	C result
requestFriendsList	B	A, C	A

requestFriendRequestersList	C	N/A	B
requestBiFriendsList	B	A	N/A

Table. Result of the three methods call

The following example shows how a client requests the aforementioned friend management functionalities.

```
// A client has already initialized CM.
String strFriendName = null;
String strDeleteName = null;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
try {
	System.out.print("Input a friend name to add: ");
	strFriendName = br.readLine();
	System.out.print("Input a friend name to delete: ");
	strDeleteName = br.readLine();
} catch (IOException e) {
	e.printStackTrace();
	return;
}

m_clientStub.addNewFriend(strFriendName);
m_clientStub.removeFriend(strDeleteName);
System.out.println("====== request current friends list");
m_clientStub.requestFriendsList();
System.out.println("====== request friend requesters list");
m_clientStub.requestFriendRequestersList();
System.out.println("====== request bi-directional friends list");
m_clientStub.requestBiFriendsList();
```

Figure. Example of friend management requests

When the default server receives the request for friends, requesters, or bi-friends from a client, it sends corresponding user list as the *RESPONSE_FRIEND_LIST*, *RESPONSE_FRIEND_REQUESTER_LIST*, or *RESPONSE_BI_FRIEND_LIST* event to the requesting client. The three events have the same event fields as described below. One of the event fields is the friend list, but the meaning of the list is different according to an event ID. The friend list contains a maximum of 50 user names. If the total number exceeds 50, the server then sends the event more than once.

Event Type		CMInfo.CM_SNS_EVENT	
Event ID		CMSNSEvent.RESPONSE_FRIEND_LIST CMSNSEvent.RESPONSE_FRIEND_REQUESTER_LIST CMSNSEvent.RESPONSE_BI_FRIEND_LIST	
Event field	Field Data type	Field definition	Get method
User name	String	The name of a requesting user	getUserName()
Total number	int	Total number of requested	getTotalNumFriends()

of friends		users	
Number of friends	int	Number of requested users in this event	getNumFriends()
Friend list	ArrayList<String>	List of requested user names	getFriendList()

Table. Response events of the three list requests

When the requesting client receives a reply event of the request for the friend management, it can check the result as the following example which is only relevant part of the client event handler.

```
…
private void processSNSEvent(CMEvent cme)
{
        CMSNSInfo snsInfo = m_clientStub.getCMInfo().getSNSInfo();
        CMSNSContentList contentList = snsInfo.getSNSContentList();
        CMSNSEvent se = (CMSNSEvent) cme;
        int i = 0;

        switch(se.getID())
        {
        case CMSNSEvent.ADD_NEW_FRIEND_ACK:
                if(se.getReturnCode() == 1)
                {
                        System.out.println("["+se.getUserName()+"] succeeds to add
a friend["+se.getFriendName()+"].");
                }
                else
                {
                        System.out.println("["+se.getUserName()+"] fails to add a
friend["+se.getFriendName()+"].");
                }
                break;
        case CMSNSEvent.REMOVE_FRIEND_ACK:
                if(se.getReturnCode() == 1)
                {
                        System.out.println("["+se.getUserName()+"] succeeds to
remove a friend["+se.getFriendName()+"].");
                }
                else
                {
                        System.out.println("["+se.getUserName()+"] fails to remove
a friend["+se.getFriendName()+"].");
                }
                break;
        case CMSNSEvent.RESPONSE_FRIEND_LIST:
                System.out.println("["+se.getUserName()+"] receives
"+se.getNumFriends()+" friends "+"of total
"+se.getTotalNumFriends()+"friends.");
                System.out.print("Friends: ");
                for(i = 0; i < se.getFriendList().size(); i++)
                {
                        System.out.print(se.getFriendList().get(i)+" ");
                }
                System.out.println();
```

```
                break;
        case CMSNSEvent.RESPONSE_FRIEND_REQUESTER_LIST:
                System.out.println("["+se.getUserName()+"] receives
"+se.getNumFriends()+" requesters "+"of total "+se.getTotalNumFriends()+"
requesters.");
                System.out.print("Requesters: ");
                for(i = 0; i < se.getFriendList().size(); i++)
                {
                        System.out.print(se.getFriendList().get(i)+" ");
                }
                System.out.println();
                break;
        case CMSNSEvent.RESPONSE_BI_FRIEND_LIST:
                System.out.println("["+se.getUserName()+"] receives
"+se.getNumFriends()+" bi-friends "+"of total "+se.getTotalNumFriends()+" bi-
friends.");
                System.out.print("Bi-friends: ");
                for(i = 0; i < se.getFriendList().size(); i++)
                {
                        System.out.print(se.getFriendList().get(i)+" ");
                }
                System.out.println();
                break;
        }
        return;
}
```

Figure. Handling the result of the friend management requests

13.2 SNS content management

Using the functionality of SNS content management of CM, a client application can request to upload and download SNS content which is managed by the CM default server. For uploading and downloading content, a client can call the *requestSNSContentUpload* and *requestSNSContent* methods in the CM client stub. If the CM DB is used for persistency, the CM default server stores the uploaded content to a SNS content table of the CM DB. If the default server does not use the CM DB, it maintains the uploaded content only in its memory while the server is running. Therefore, it is recommended to use the CM DB for supporting persistency of SNS content. To request content upload or download, a client must log in to the default server.

13.2.1 Content upload

The synopsis of the *requestSNSContentUpload* method is described below.

```
void requestSNSContentUpload(String user, String message, int nNumAttachedFiles,
        int nReplyOf, int nLevelOfDisclosure, ArrayList<String> filePathList)
```

Figure. requestSNSContentUpload method

A client can call this method to upload a message to the default server. The method requires six

parameters. The first parameter, '*user*', is the name of a user who uploads a message. The second parameter, '*message*', is a text message. The third parameter, '*nNumAttachedFiles*', is the number of attached files in this message. The parameter value must be the same as the number of elements in a given file path list as the last parameter. The fourth parameter, '*nReplyOf*, indicates an ID (greater than 0) of content to which this message replies. If the value is 0, it means that the uploaded content is not a reply but an original one. The fifth parameter, '*nLevelOfDisclosure*', specifies the level of disclosure (LoD) of the uploaded content. CM provides four levels of disclosure of content from 0 to 3. LoD 0 is to open the uploaded content to public. LoD 1 allows only users who added the uploading user as friends to access the uploaded content. LoD 2 allows only bi-friends of the uploading user to access the uploaded content. (Refer to the friend management section for details of different friend concepts.) LoD 3 does not open the uploaded content and makes it private. The last parameter, '*filePathList*', is the list of attached files. Path names of attached files should be given as the *ArrayList* type, and the number of array elements must be the same as the '*nNumAttachedFiles*' parameter.

The following example shows how a client uploads content.

```
// A client has already initialize CM and logged in to the server.
String strMessage = null;
int nNumAttachedFiles = 0;
int nReplyOf = 0;
int nLevelOfDisclosure = 0;
ArrayList<String> filePathList = null;
System.out.println("====== test SNS content upload");
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
try {
	System.out.print("Input message: ");
	strMessage = br.readLine();
	System.out.print("Number of attached files: ");
	nNumAttachedFiles = Integer.parseInt(br.readLine());
	System.out.print("Content ID to which this content replies (0 for no
reply): ");
	nReplyOf = Integer.parseInt(br.readLine());
	System.out.print("Level of Disclosure (0: to everyone, 1: to my
followers, 2: to bi-friends, 3: nobody): ");
	nLevelOfDisclosure = Integer.parseInt(br.readLine());

	if(nNumAttachedFiles > 0)
	{
		String strPath = null;
		filePathList = new ArrayList<String>();
		System.out.println("Input path names of attahced files..");
		for(int i = 0; i < nNumAttachedFiles; i++)
		{
			System.out.print(i+": ");
			strPath = br.readLine();
			filePathList.add(strPath);
```

```
                }
        }

} catch (IOException e) {
        e.printStackTrace();
        return;
}

String strUser =
m_clientStub.getCMInfo().getInteractionInfo().getMyself().getName();
m_clientStub.requestSNSContentUpload(strUser, strMessage, nNumAttachedFiles,
nReplyOf, nLevelOfDisclosure, filePathList);
```

Figure. Example of request for uploading content

If the server receives the content upload request, it stores the requested message with the user name, the index of the content, the upload time, the number of attachments, the reply ID, and the level of disclosure. If the content has attached files, the client separately transfers them to the server. After the upload task is completed, the server sends the *CONTENT_UPLOAD_RESPONSE* event to the requesting client so that the client can check the result of the request. The detailed event fields of the *CONTENT_UPLOAD_RESPONSE* event is described below.

Event Type		CMInfo.CM_SNS_EVENT	
Event ID		CMSNSEvent.CONTENT_UPLOAD_RESPONSE	
Event field	Field Data type	Field definition	Get method
Return code	int	Result code of the request 1: succeeded 0: failed	getReturnCode()
Content ID	int	An index of the uploaded content in a content table	getContentID()
Date and time	String	Date and time of the upload	getDate()
User name	String	Requesting user name	getUserName()

Table. CONTENT_UPLOAD_RESPONSE event

A client can catch the CONTENT_UPLOAD_RESPONSE event in the client event handler and can figure out the result of the upload request as described in the following example which is only the relevant part of the client event handler. The example checks whether the request is succeeded or not, and prints out the upload information such as the user name, content ID, and date and time.

```
…
private void processSNSEvent(CMEvent cme)
{
        CMSNSEvent se = (CMSNSEvent) cme;
        switch(se.getID())
```

```
        {
        case CMSNSEvent.CONTENT_UPLOAD_RESPONSE:
                if( se.getReturnCode() == 1 )
                {
                        System.out.println("Content upload succeeded.");
                }
                else
                {
                        System.out.println("Content upload failed.");
                }
                System.out.println("user("+se.getUserName()+"),
seqNum("+se.getContentID()+"), time("+se.getDate()+").");
                break;
        }
        return;
}
```

Figure. Handling the result of the content upload request

13.2.2 Content download

A client can request to download content by calling the *requestSNSContent* method of which synopsis is described below.

```
void requestSNSContent(String strWriter, int nOffset)
```

Figure. requestSNSContent method

The first parameter of the method, '*strWriter*', specifies a user who uploaded content. In this parameter, the client can designate a specific writer name or a friend group. If the parameter value is a specific user name, the client downloads only content that was uploaded by the specified name and that is accessible by the requester. If the parameter value is '*CM_MY_FRIEND*', the client downloads content that was uploaded by the requester's friends. If the parameter is '*CM_BI_FRIEND*', the client downloads content that was uploaded by the requester's bi-friends. If the '*strWriter*' parameter is an empty string (""), the client does not specify a writer name and it downloads all content that the requester is eligible to access. The last parameter, '*nOffset*' is an offset from the beginning of the requested content list. That is, a client wants to download some number of SNS messages starting from the *nOffset*-th most recent content. The *nOffset* value is greater than or equal to 0. The following example shows how a client requests to download content.

```
// A client has already initialize CM and logged in to the server.
String strWriterName = null;
int nContentOffset = 0;
String strUserName =
m_clientStub.getCMInfo().getInteractionInfo().getMyself().getName();

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
try {
```

```
        System.out.print("Input offset(>= 0): ");
        nContentOffset = Integer.parseInt(br.readLine());
        System.out.print("Content writer(Enter for no designation, "
                + "CM_MY_FRIEND for my friends, CM_BI_FRIEND for bi-friends, or
specify a name): ");
        strWriterName = br.readLine();
} catch (NumberFormatException e) {
        e.printStackTrace();
        return;
} catch (IOException e) {
        e.printStackTrace();
        return;
}

m_clientStub.requestSNSContent(strUserName, strWriterName, nContentOffset);
```

Figure. Example of request for downloading content

When the server receives the download request, it first determines how many SNS messages will be sent. The number of messages is decided by the *DOWNLOAD_SCHEME* field of the server configuration file. If this field is set to 0, the server uses the fixed maximum number of messages per request, and the number is decided as the *DOWNLOAD_NUM* field value of the configuration file. If the *DOWNLOAD_SCHEME* is set to 1, the server uses our dynamic downloading scheme that determines the number of downloaded messages according to the round-trip delay between the server and the requesting client. Each SNS message is then sent to the client as the *CONTENT_DOWNLOAD* event that can be caught in the client event handler. The event fields of the *CONTENT_DOWNLOAD* event is described below.

Event Type		CMInfo.CM_SNS_EVENT	
Event ID		CMSNSEvent.CONTENT_DOWNLOAD	
Event field	Field Data type	Field definition	Get method
User name	String	Requester name	getUserName()
Offset	int	Requested content offset	getContentOffset()
Content ID	int	Content ID	getContentID()
Date and time	String	Written date and time of the content	getDate()
Writer name	String	Writer name of the content	getWriterName()
Text message	String	Text message of the content	getMessage()
Number of attachments	int	Number of attached files	getNumAttachedFiles()
Reply ID	int	Content ID to which this message replies (0 for no reply)	getReplyOf()

Level of disclosure	int	Level of disclosure of the message 0: open to public 1: open only to friends 2: open only to bi-friends 3: private	getLevelOfDisclosure()
File name list	ArrayList<String>	The list of attached file name	getFileNameList()

Table. DOWNLOAD_CONTENT event

In most cases, the server sends multiple *CONTENT_DOWNLOAD* events due to the corresponding number of SNS messages, and it sends the *CONTENT_DOWNLOAD_END* event as the end signal of current download. This event contains a field that is the number of downloaded messages. A client can send another download request by updating the offset parameter with the number of previously downloaded messages. The detailed event fields of the CONTENT_DOWNLOAD_END event is described below.

Event Type		CMInfo.CM_SNS_EVENT	
Event ID		CMSNSEvent.CONTENT_DOWNLOAD_END	
Event field	Field Data type	Field definition	Get method
User name	String	Requester name	getUserName()
Offset	int	Requested content offset	getContentOffset()
Content ID	int	Content ID	getContentID()
Download number	int	Number of downloaded SNS messages	getNumContents()

Table. DOWNLOAD_CONTENT_END event

The following simple example shows how a client prints out downloaded messages and their number in the client event handler.

```
…
private void processSNSEvent(CMEvent cme)
{
        CMSNSEvent se = (CMSNSEvent) cme;
        switch(se.getID())
        {
        case CMSNSEvent.CONTENT_DOWNLOAD:
                System.out.println("---------------------------------------");
                System.out.println("ID("+cont.getContentID()+"),
Date("+cont.getDate()+"), Writer("+cont.getWriterName()+"),
#attachment("+cont.getNumAttachedFiles()+"), replyID("+cont.getReplyOf()+"),
lod("+cont.getLevelOfDisclosure()+")");
                System.out.println("Message: "+cont.getMessage());
                if(cont.getNumAttachedFiles() > 0)
```

```
                {
                        ArrayList<String> fNameList = cont.getFileNameList();
                        for(int i = 0; i < fNameList.size(); i++)
                        {
                                System.out.println("attachment:
"+fNameList.get(i));
                        }
                }
                break;
        case CMSNSEvent.CONTENT_DOWNLOAD_END:
                System.out.println("# downloaded contents:
"+se.getNumContents());
                break;
        }
        return;
}
```

Figure. Handling the result of the content download request

If content has attached files, the client separately stores them in the default directory that is set in the *FILE_PATH* field of the client configuration file. The client should use such directory information from the *CMFileTransferInfo* classes in order to access the downloaded file because the *CMSNSEvent.CONTENT_DOWNLOAD* event includes only attached file names.

14. Publish-Subscribe messaging service

CM also provides clients with publish-subscribe messaging service that is implemented based on MQTT v3.1.1 (http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html). CM modifies 14 MQTT packets into corresponding CM MQTT events as shown in the following table.

Table. MQTT packets vs. CM MQTT event classes

MQTT packets	CM MQTT event classes	Event type	Event ID
CONNECT	CMMqttEventCONNECT	CMInfo.CM_MQTT_EVENT	CMMqttEvent.CONNECT
CONNACK	CMMqttEventCONNACK	CMInfo.CM_MQTT_EVENT	CMMqttEvent.CONNACK
PUBLISH	CMMqttEventPUBLISH	CMInfo.CM_MQTT_EVENT	CMMqttEvent.PUBLISH
PUBACK	CMMqttEventPUBACK	CMInfo.CM_MQTT_EVENT	CMMqttEvent.PUBACK
PUBREC	CMMqttEventPUBREC	CMInfo.CM_MQTT_EVENT	CMMqttEvent.PUBREC
PUBREL	CMMqttEventPUBREL	CMInfo.CM_MQTT_EVENT	CMMqttEvent.PUBREL
PUBCOMP	CMMqttEventPUBCOMP	CMInfo.CM_MQTT_EVENT	CMMqttEvent.PUBCOMP
SUBSCRIBE	CMMqttEventSUBSCRIBE	CMInfo.CM_MQTT_EVENT	CMMqttEvent.SUBSCRIBE
SUBACK	CMMqttEventSUBACK	CMInfo.CM_MQTT_EVENT	CMMqttEvent.SUBACK
UNSUBSCRIBE	CMMqttEventUNSUBSCRIBE	CMInfo.CM_MQTT_EVENT	CMMqttEvent.UNSUBSCRIBE

UNSUBACK	CMMqttEventUNSUBACK	CMInfo.CM_MQTT_EVENT	CMMqttEvent.UNSUBACK
PINGREQ	CMMqttEventPINGREQ	CMInfo.CM_MQTT_EVENT	CMMqttEvent.PINGREQ
PINGRESP	CMMqttEventPINGRESP	CMInfo.CM_MQTT_EVENT	CMMqttEvent.PINGRESP
DISCONNECT	CMMqttEventDISCONNECT	CMInfo.CM_MQTT_EVENT	CMMqttEvent.DISCONNECT

A CM MQTT event class is inherited from the CMMqttEventFixedHeader class that is inherited from the CMMqttEvent class that is inherited from the CMEvent class. For example, the following figure is the structure of the CMMqttEventCONNECT class.

Figure. CMMqttEventCONNECT class inheritance

In the CM publish-subscriber service, the server CM has the role of an event broker and the client CM can become a publisher or a subscriber. A client application can use the publish-subscribe service via the CMMqttManager class after it logs in to the server. The following code snippet shows how the client gets the CMMqttManager reference from the CM client stub.

```
// m_clientStub is the reference to the CM client stub.
CMMqttManager mqttManager =
(CMMqttManager) m_clientStub.findServiceManager(CMInfo.CM_MQTT_MANAGER);
```

Figure. Getting the CMMqttManager from the client stub

14.1 Connect

A client must call one of the connect() methods in order to use the publish-subscribe service of CM. To call the connect() method, the client must log in to the server. The following figure shows synopses of the connect() methods.

```
public boolean connect()
public boolean connect(String strWillTopic, String strWillMessage, boolean bWillRetain, byte willQoS, boolean bWillFlag, boolean bCleanSession)
```

Figure. connect() methods in CMMqttManager

The connect() method is the same as connect(null, null, false, (byte)0, false, false). The parameters of the second connect() method are described below:

- strWillTopic – will topic. The default value is null.

- strWillMessage – will message. The default value is null.
- bWillRetain – will retain. The default value is false.
- willQoS – will QoS. The default value is 0.
- bWillFlag – will flag. The default value is null.
- bCleanSession – clean session flag. The default value is false.

The parameter details are described in the specification of the CONNECT[1] packet of MQTT. The following code snippet shows how a client connects to the server for the publish-subscribe service.

```
// mqttManager is the CMMqttManager and the client has logged in to the server.
mqttManager.connect();
```

Figure. Connect the CM publish-subscribe service

When the client calls the connect() method, the client CM generates a CMMqttEventCONNECT event and sends it to the server. After processing the connect event, the server replies by sending a CMMqttEventCONNACK event. The client can catch the reply event in the client event handler. Event fields of the CMMqttEventCONNACK event is described below.

Event Type		CMInfo.CM_MQTT_EVENT	
Event ID		CMMqttEvent.CONNACK	
Event field	Field Data type	Field definition	Get method
Connect Acknowledgement Flags (Session Present Flag)	byte	Please refer to the CONNACK[2] packet of MQTT	boolean isConnAckFlag()
Connect Return Code	byte		getReturnCode()

Table. CMMqttEventCONNACK event

The following code snippet shows how the client event handler catches the CMMqttEventCONNACK event.

[1] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718028

[2] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718033

```
public void processEvent(CMEvent cme) {
        switch(cme.getType()){
        …
        case CMInfo.CM_MQTT_EVENT:
                processMqttEvent(cme);
                break;
        }
}

private void processMqttEvent(CMEvent cme)
{
        switch(cme.getID())
        {
        case CMMqttEvent.CONNACK:
                CMMqttEventCONNACK conackEvent = (CMMqttEventCONNACK)cme;
                System.out.println("received "+conackEvent.toString());
                break;
        …
        }
        return;
}
```

Figure. Handling CMMqttEventCONNACK in the event handler

The following figure shows event transmission of the connect process between the client and the server.

Figure. The connect process of the publish-subscribe service

14.2 Publish

To publish an event, a client can call one of the publish() methods as shown in the following synopses.

```
Public boolean publish(String strTopic, String strMsg)
public boolean publish(String strTopic, String strMsg, byte qos)
public boolean publish(String strTopic, String strMsg, byte qos, boolean bDupFlag, boolean
```

```
bRetainFlag)
```

Figure. publish() methods in CMMqttManager

The call of publish(strTopic, strMsg) is the same as the call of publish(strTopic, strMsg, 0) that is the same as the call of publish(strTopic, strMsg, 0, false, false). Parameters of the publish() methods are described below:

- strTopic – topic name
- strMsg – application message
- qos – QoS. The default value is 0.
- bDupFlag – DUP flag. The default value is false.
- bRetainFlag – retain flag. The default value is false.

The parameter details are described in the specification of the PUBLISH[3] packet of MQTT. When the client calls the publish() method, the client CM generates a CMMqttEventPUBLISH event and sends it to the server. The following code snippet shows how the client call the publish() method.

```
// mqttManager is the CMMqttManager and the client has logged in to the server.
mqttManager.publish("CM/test", "test message");
```

Figure. Publish a message with a topic name

After processing the publish event, the server may or may not send a reply event. The reply event is determined by the QoS value of the publish event as described below.

- QoS 0 – no reply event
- QoS 1 – CMMqttEventPUBACK event
- QoS 2 – CMMqttEventPUBREC event

In the QoS 2 case, after the client receives the reply event, it sends the CMMqttEventPUBREL event to the server that then sends the CMMqttEventPUBCOMP event to the client as the second reply event. The client can catch the reply events in the client event handler. Event fields are described

[3] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718037

below.

Event Type		CMInfo.CM_MQTT_EVENT	
Event ID		CMMqttEvent.PUBACK CMMqttEvent.PUBREC CMMqttEvent.PUBCOMP	
Event field	Field Data type	Field definition	Get method
Packet Identifier	int	Please refer to the PUBACK[4], PUBREC[5], and PUBCOMP[6] packets of MQTT	getPacketID()

The following code snippet shows that the client catches and prints out the reply events regarding the process of the publish operation.

```
…
private void processMqttEvent(CMEvent cme)
{
        switch(cme.getID())
        {
        case CMMqttEvent.PUBACK:
                CMMqttEventPUBACK pubackEvent = (CMMqttEventPUBACK)cme;
                System.out.println("received "+pubackEvent.toString());
                break;
        case CMMqttEvent.PUBREC:
                CMMqttEventPUBREC pubrecEvent = (CMMqttEventPUBREC)cme;
                System.out.println("received "+pubrecEvent.toString());
                break;
        case CMMqttEvent.PUBCOMP:
                CMMqttEventPUBCOMP pubcompEvent = (CMMqttEventPUBCOMP)cme;
                System.out.println("received "+pubcompEvent.toString());
                break;
        …
        }
        return;
```

[4] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718043

[5] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718048

[6] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718058

```
}
```

Figure. Handling reply events regarding the process of publish

The following figures show event transmission of the publish process between the client and the server according to different QoS levels.

Figure. Publish process of the publish-subscribe service (QoS 0)

Figure. Publish process of the publish-subscribe service (QoS 1)

Figure. Publish process of the publish-subscribe service (QoS 2)

14.3 Subscribe

To subscribe pairs of a topic filter and QoS, the client can call one of the subscribe() methods as shown in the following synopses.

```
public boolean subscribe(String strTopicFilter, byte qos)
public boolean subscribe(CMList<CMMqttTopicQoS> topicQoSList)
```

Figure. subscribe() methods in CMMqttManager

The first subscribe() method is used to subscribe a pair of topic filter and QoS, and the second method is used to subscribe multiple pairs. Parameters are described below:

- strTopicFilter – topic filter
- qos – requested QoS

- topicQoSList – a list of (topic filters, QoS) pairs

The parameter details are described in the specification of the SUBSCRIBE[7] packet of MQTT. When the client calls the subscribe() method, the client CM generates a CMMqttEventSUBSCRIBE event and sends it to the server. The following code snippet shows that the client calls the subscribe() method with a topic filter pair.

```
// mqttManager is the CMMqttManager and the client has logged in to the server.
mqttManager.subscribe("CM/#", (byte)0);
```

Figure. Usage of the subscribe() method

After processing the subscribe event, the server sends a reply event (CMMqttEventSUBACK) to the client. The client can catch the reply event in its event handler. Fields information of the reply event is described below.

Event Type		CMInfo.CM_MQTT_EVENT	
Event ID		CMMqttEvent.SUBACK	
Event field	Field Data type	Field definition	Get method
Packet Identifier	int	Please refer to the SUBACK[8] packets of MQTT	getPacketID()
List of return codes	CMList<Byte>		getReturnCodeList()

Figure. CMMqttEventSUBACK event fields

The following code snippet shows that the client catches and prints out the SUBACK event in the event handler.

```
…
private void processMqttEvent(CMEvent cme)
{
        switch(cme.getID())
        {
        case CMMqttEvent.SUBACK:
```

[7] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718063

[8] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718068

```
                    CMMqttEventSUBACK subackEvent = (CMMqttEventSUBACK)cme;
                    System.out.println("received "+subackEvent.toString());
                    break;
            …
            }
            return;
}
```

Figure. Handling CMMqttEventSUBACK event

The following figure shows the event transmission of the subscribe process between the client (subscriber) and the server (event broker).

Figure. Subscribe process

14.4 Unsubscribe

To unsubscribe from a topic filter, the client can call one of the unsubscribe() methods as shown in the following synopses.

```
public boolean unsubscribe(String strTopic)
public boolean unsubscribe(CMList<String> topicList)
```

Figure. unsubscribe() methods in CMMqttManager

The first unsubscribe() method is used to unsubscribe from a topic filter, and the second method is used to unsubscribe from multiple topics. Parameters are described below:

- strTopic – topic filter
- topicList – list of topic filters

The parameter details are described in the specification of the UNSUBSCRIBE[9] packet of MQTT. When the client calls the unsubscribe() method, the client CM generates a CMMqttEventUNSUBSCRIBE event and sends it to the server. The following code snippet shows that

[9] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718072

the client calls the unsubscribe() method with a topic filter.

```
// mqttManager is the CMMqttManager and the client has logged in to the server.
mqttManager.unsubscribe("CM/#");
```

Figure. Usage of unsubscribe() method

After processing the unsubscribe event, the server sends a reply event (CMMqttEventUNSUBACK) to the client. The client can catch the reply event in its event handler. Fields information of the reply event is described below.

Event Type		CMInfo.CM_MQTT_EVENT	
Event ID		CMMqttEvent.UNSUBACK	
Event field	Field Data type	Field definition	Get method
Packet Identifier	int	Please refer to the UNSUBACK[10] packets of MQTT	getPacketID()

Figure. CMMqttEventUNSUBACK event fields

The following code snippet shows that the client catches and prints out the UNSUBACK event in the event handler.

```
…
private void processMqttEvent(CMEvent cme)
{
        switch(cme.getID())
        {
        case CMMqttEvent.UNSUBACK:
                CMMqttEventUNSUBACK unsubackEvent = (CMMqttEventUNSUBACK)cme;
                System.out.println("received "+unsubackEvent.toString());
                break;
        …
        }
        return;
}
```

Figure. Handling CMMqttEventUNSUBACK event

The following figure shows the event transmission of the unsubscribe process between the client

[10] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718077

(subscriber) and the server (event broker).

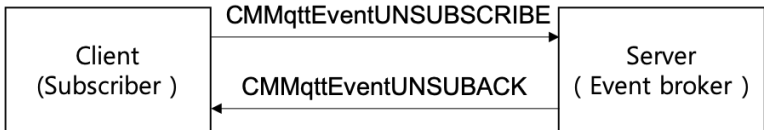


Figure. Unsubscribe process

14.5 Disconnect

To disconnect from the publish-subscribe service, the client can call the disconnect() method as shown in the following synopsis.

```
public boolean disconnect()
```

Figure. disconnect() method in CMMqttManager

The operation of the disconnect() method is similar to that of the DISCONNECT[11] packet of MQTT. However, the client call the disconnect() method, it does not disconnect communication channels from the server because it can still use other CM services instead of the publish-subscribe service. That is, the disconnect() method of the CMMqttManager class just disconnects from the publish-subscribe service. When the client calls the disconnect() method, it generates a CMMqttEventDISCONNECT event, and send it to the server. The following code snippet shows that the client calls the disconnect() method.

```
// mqttManager is the CMMqttManager and the client has logged in to the server.
mqttManager.disconnect();
```

Figure. Usage of the disconnect() method

After processing the disconnect event, the server does not send a reply event. The event transmission of the disconnect process between the client and the server is shown in the following figure.

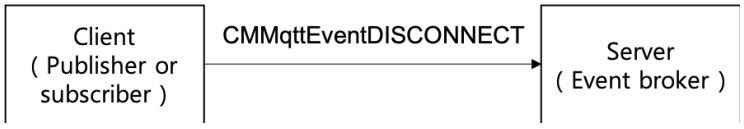


Figure. Disconnect process

[11] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718090

15. MySQL server configuration for the CM DB management

To make the CM server access DB, we need to install and set up MySQL server and MySQL Workbench. We can download and install MySQL Community Server and MySQL Workbench from the URL (https://dev.mysql.com/downloads/mysql/). This guide describes the configuration of the CM DB with MySQL Community Server 5.7 and the MySQL Workbench 6.3. After successful installation and start of the MySQL server, we can check whether the MySQL server is running or not as follows according to different OS:

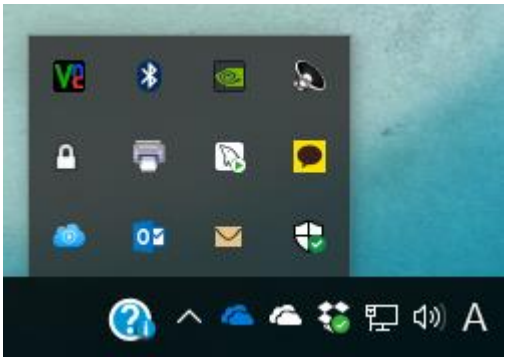
15.1 Mac OS (Sierra)

- Go to the System Configuration and click the "MySQL" icon.

- Check whether the MySQL server is currently running or not. If not, click the "Start MySQL Server" button.

15.2 MS Windows

- Click the MySQL notifier icon in the system tray.

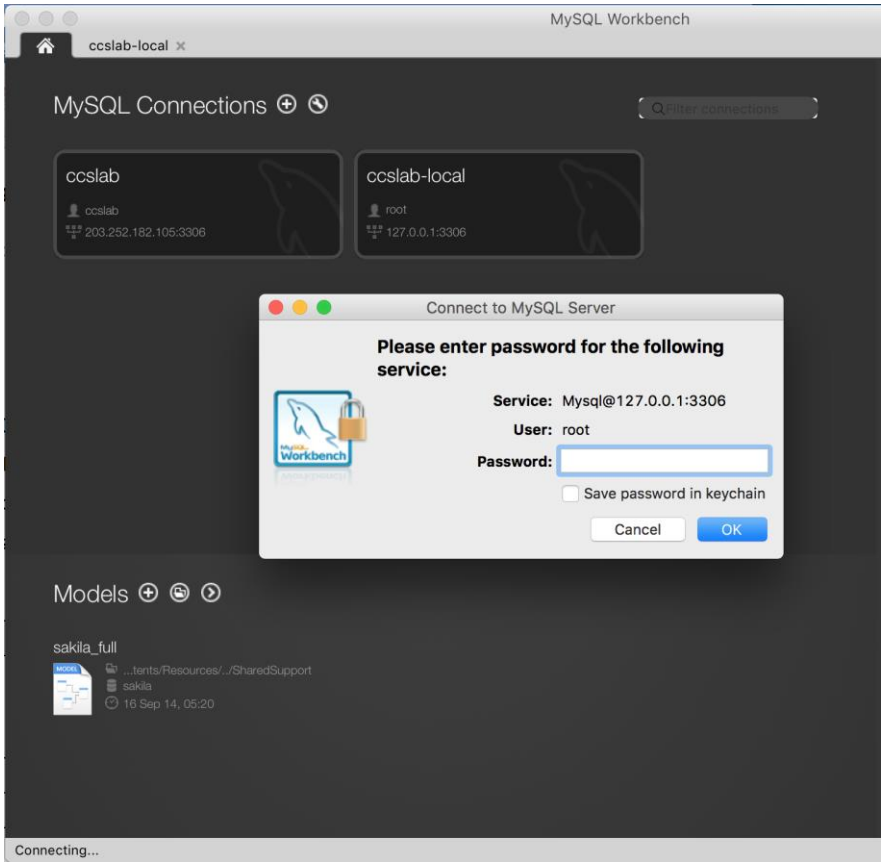


- Check whether MySQL server is currently running or not. If not, click the “Start” menu.

After we start MySQL server, we should set up the MySQL server so that the CM server can access the CM DB. We use MySQL Workbench to create the new schema and import CM DB table data. We describe the configuration procedure with the MacOS version of Workbench. The Windows version of Workbench also has almost the same UI.

1. Run the Workbench program.

2. Click the default “local” connection and input the MySQL root password.



3. (Create the new schema)

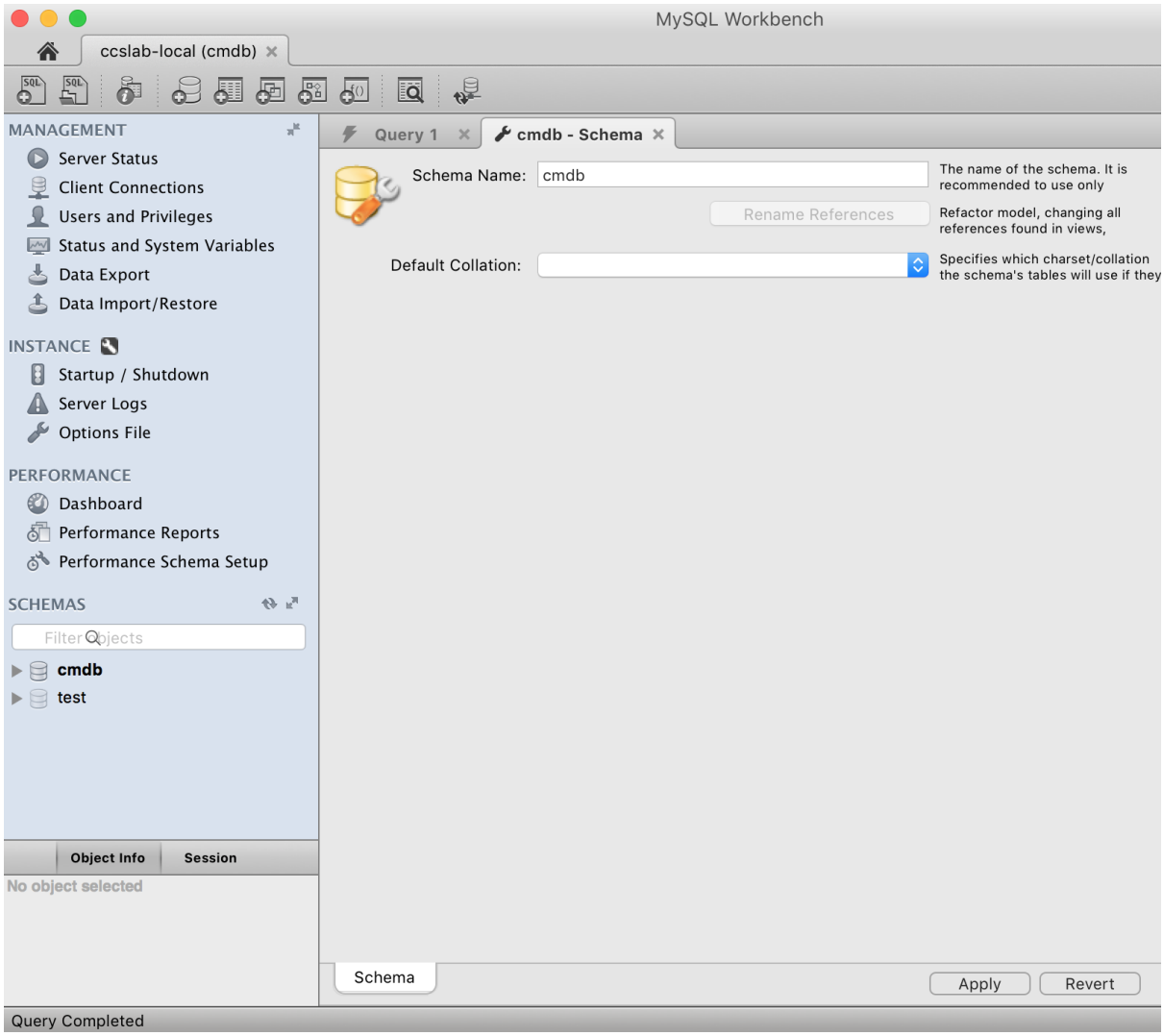
   A. Click the "create a new schema" icon.

   B. Input the schema name as "cmdb".

      The name "cmdb" will be set in the CM server configuration file (cm-server.conf).

   C. Click the "Apply" button.

   D. The new "cmdb" schema is created in the left SCHEMAS view of Workbench.



4. (Add a new user that the CM server will use to connect to the CM DB)

   A. Click the "Users and Privileges" menu in the left MANAGEMENT view, and input the root password.

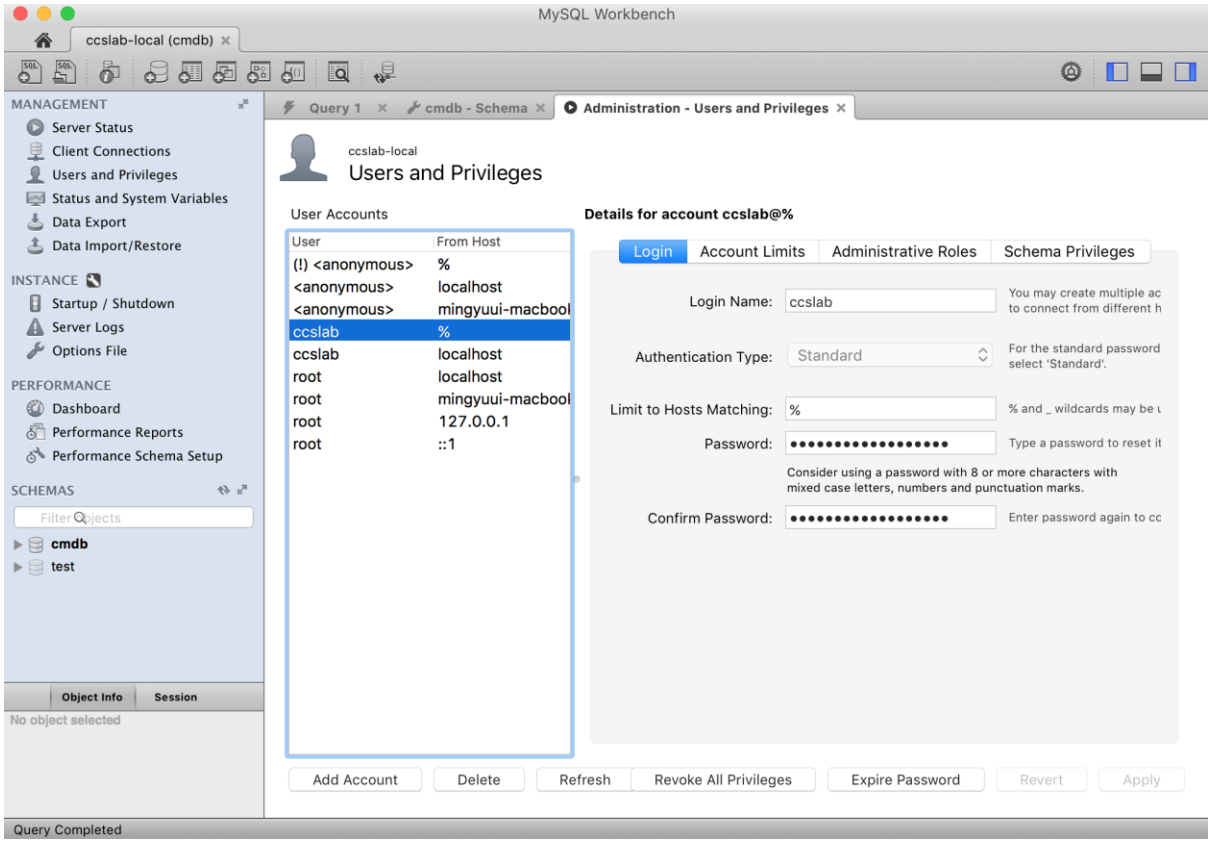
   B. Click the "Add Account" button, and input the new user information as follows:

      i. Login name: any user name (ex. ccslab)

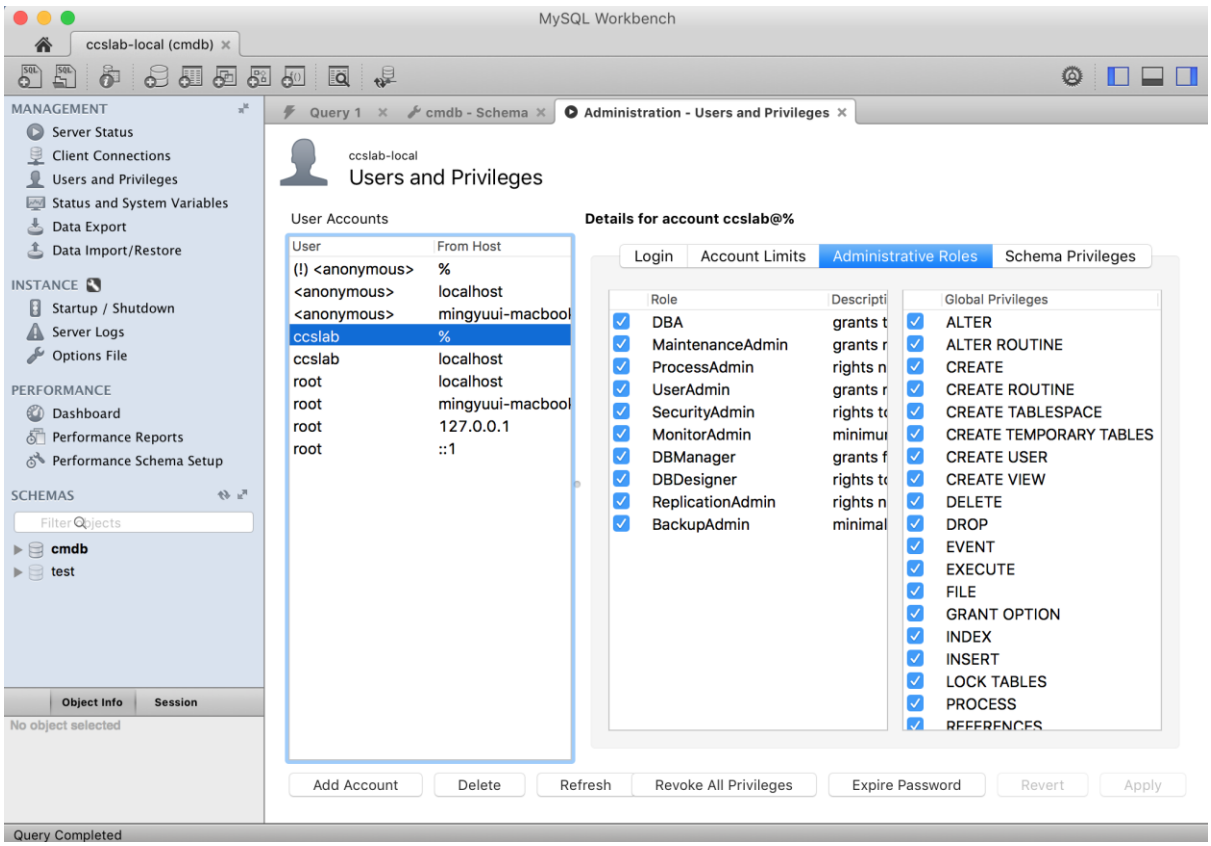
      ii. Limit to Hosts Matching: % (for accessing the DB from any remote host)

      iii. Password: new password (ex. ccslab)

      The login name and password will be set in the cm server configuration file (cm-server.conf).



C. Click the "Administrative Roles" tab, and select all roles.



5. (Add another new user with the same name)

   A. Follow the step 4 again. Only difference of the new user is the value of the "Limits to Host Matching" field. This time the value is "localhost".

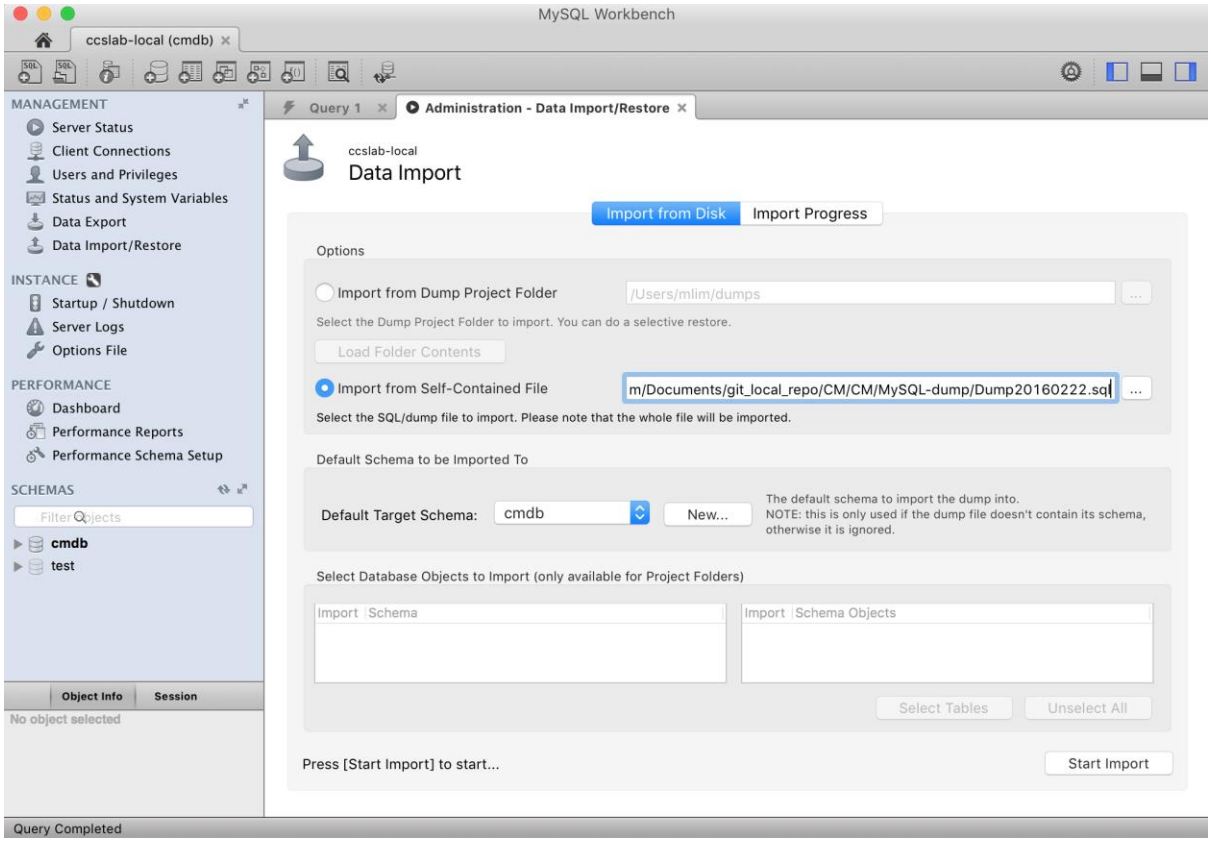
   B. Check that the two same user names are created with different hosts (% and localhost).

6. (Import the CM DB table data)

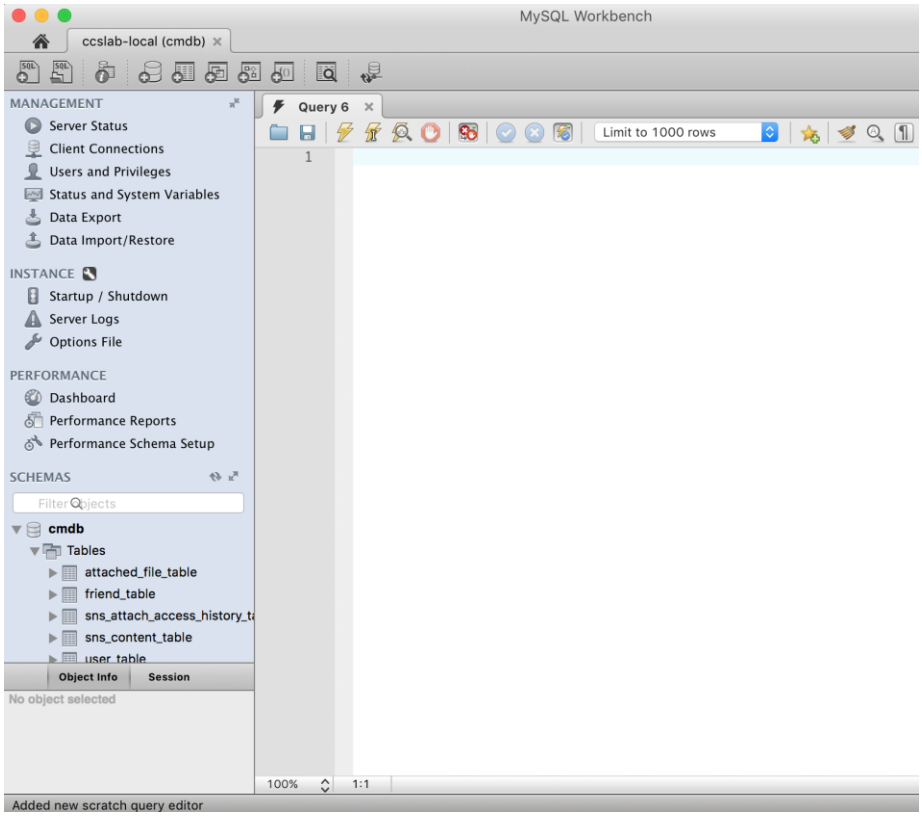
   A. Click the "Data Import/Restore" menu in the left MANAGEMENT view.

   B. Select the "Import from Self-Contained File" option, and select the dump file (Dump20160222.sql) that is included in the "SQL-dump" directory of the CMTest project.

   C. Choose the "cmdb" in the "Default Target Schema" field, and click the "Start Import" button.



   D. Check whether the "cmdb" schema successfully imports all the CM DB tables data.



After the CM DB is imported to the cmdb schema, we can set up the DB connection information in the CM server configuration file as follows:

- DB_USE 1
- DB_HOST localhost
- DB_USER ccslab
- DB_PASS ccslab
- DB_PORT 3306
- DB_NAME cmdb

The DB_HOST field is localhost if the CM DB is configured in the same host as the CM server host. If the CM DB is located in a different host, the DB_HOST value can be the corresponding IP address. The DB_USER and DB_PASS fields are the user name and password that are added with Workbench. The DB_PORT value is 3306 that is the default port number for accessing the MySQL DB. The DB_NAME is "cmdb" that is the DB name of the newly added schema in Workbench.

15.3 CM DB management

Some CM functionalities require DB access for keeping large amount of information persistently such as user profile, friend list, SNS content, and attachment files. The aforementioned APIs for these functionalities then works because the CM DB has set four tables that contain minimum fields for the management of users, friends, and SNS content. The following tables show the default fields of four DB tables.

Field name	Data type	Attributes	Description
seqNum	int(10)	UN,AI,PK	A sequence number. It is a primary key, increments automatically, and must be unique.
userName	varchar(80)		User name
password	varchar(80)		User password
creationTime	datetime		Date and time when the user is registered

Table. User profile table

Field name	Data type	Attributes	Description
userName	varchar(80)	PK	User name
friendName	varchar(80)	PK	Friend name

Table. Friend table

Field name	Data type	Attributes	Description
seqNum	int(10)	UN, AI, PK	A sequence number. It is a primary key, increments automatically, and must be unique.
creationTime	Datetime		Date and time when the content is uploaded
userName	varchar(80)		User name
textMessage	varchar(256)		Text message
numAttachedFiles	Int(10)		Number of attached files
replyOf	Int(10)		Index (seqNum) of the source content
levelOfDisclosure	Int(10)		Level of information disclosure (0 or null) / 1 / 2 / 3

Table. SNS content table

Field name	Data type	Attributes	Description
seqNum	int(10)	UN, AI, PK	A sequence number. It is a primary key, increments automatically, and must be

			unique.
contentID	int(10)		An associated content ID
filePath	varchar(256)		A path where the file is located
fileName	varchar(256)		A file name

Table. Attached file table

If a developer wants to use other DB tables instead of the default tables provided by the CM DB, he/she can make and use other tables in the CM DB, and can directly use DB queries to manage the user profile, the friend list, and SNS content in a different way. To support the direct access to the CM DB, CM provides a utility class, *CMDBManager*.

15.3.1 DB configuration

As described in the CM initialization section, CM uses MySQL as a DBMS and DB is configured in the server configuration file. The relevant fields include the *DB_HOST*, *DB_USER*, *DB_PASS*, *DB_PORT*, and *DB_NAME*. When the server CM starts, it connects to the configured DB so that a server application can access to it.

15.3.2 CMDBManager class

The *CMDBManager* class is a static utility class and provides methods to directly access to the CM DB. Using this class, a developer can make a query to not only the default three (users, friends, and SNS content) tables but also any other table. For the general management of the CM DB, the *CMDBManager* class provides following methods.

Synopsis	Description
`ResultSet sendSelectQuery(String strQuery, CMInfo cmInfo)`	Send a MySQL query for SELECT statement
`int sendUpdateQuery(String strQuery, CMInfo cmInfo)`	Send a MySQL query for the statement such as UPDATE, INSERT, DELETE, and so on
`long sendRowNumQuery(String table, CMInfo cmInfo)`	Return the total number of rows in a table

Table. Methods for general management of CM DB

All the methods of the *CMDBManager* class require a common parameter: a reference to the instance of the *CMInfo* class which contains the entire internal information of CM. The other parameter of the above two methods is a query string which follows the statement rule of MySQL. The *sendSelectQuery* method is used specifically for *SELECT* statement and it returns the result as the *ResultSet* type of JDBC.

The *sendUpdateQuery* method is used to send any query which requests to update data in DB. The return value is an integer number which indicates whether the query succeeds or fails. If the return value is 1, the query is successful. Otherwise, the query fails. If a developer adds new tables in DB, he/she can directly manage the tables with these two methods. The *sendRowNumQuery* method is used if a developer wants to know how many rows are in a table.

For the default tables of users, friends, and SNS content, the *CMDBManager* also provides supplementary methods to manage the data in the tables.

15.3.3 Management of the user profile table

The following table shows methods to access and manage data in the default table for user profile.

Synopsis	Description
`int queryInsertUser(String name, String password, CMInfo cmInfo)`	Add a user profile name: a user name password: a user password
`boolean authenticateUser(String strUserName, String strPassword, CMInfo cmInfo)`	Conduct the authentication of a user strUserName: a user name strPassword: a user password
`ResultSet queryGetUsers(int index, int num, CMInfo cmInfo)`	Retrieve user profile information index: a starting index in the user table num: number of users to be retrieved
`int queryUpdateUser(String name, String fieldName, String value, CMInfo cmInfo)`	Update a column value of the user table name: a user name fieldName: a column name of the user table value: a value of the column
`int queryDeleteUser(String name, CMInfo cmInfo)`	Delete a user profile name: a user name
`void queryTruncateUserTable(CMInfo cmInfo)`	Clear the user table

Table. Methods for the user profile table

15.3.4 Management of the friend table

The following table shows methods to access and manage data in the default table for friends.

Synopsis	Description
`int queryInsertFriend(String strUserName, String strFriendName, CMInfo cmInfo)`	Add a friend strUserName: a user name

	strFriendName: a friend name
`int queryDeleteFriend(String strUserName, String strFriendName, CMInfo cmInfo)`	Delete a friend strUserName: a user name strFriendName: a friend name
`ArrayList<String> queryGetFriendsList(String strUserName, CMInfo cmInfo)`	Retrieve friends that the user(*strUserName*) adds strUserName: a user name
`ArrayList<String> queryGetRequestersList(String strUserName, CMInfo cmInfo)`	Retrieve users who add the user(*strUserName*) as a friend strUserName: a user name

Table. Methods for the friends table

15.3.5 Management of the SNS content table

The following table shows methods to access and manage data in the default table for SNS content.

Synopsis	Description
`int queryInsertSNSContent(String userName, String text, int nNumAttachedFiles, int nReplyOf, int nLevelOfDisclosure, CMInfo cmInfo)`	Add SNS content userName: a user name text: a text message nNumAttachedFiles: number of attached files nReplyOf: a replying ID nLevelOfDisclosure: level of disclosure
`CMSNSContentList queryGetSNSContent(String strRequester, String strWriter, int index, int num, CMInfo cmInfo)`	Retrieve SNS content list strRequester: a requester name strWriter: a content writer name index: a starting index in the requested content list num: number of messages to be retrieved
`int queryUpdateSNSContent(int seqNum, String fieldName, String value, CMInfo cmInfo)`	Update a column value of the content table seqNum: a sequence number of content fieldName: a column name of the content table value: a value of the column
`int queryDeleteSNSContent(int seqNum, CMInfo cmInfo)`	Delete SNS content seqNum: a sequence number of content
`void queryTruncateSNSContentTable(CMInfo cmInfo)`	Clear the content table

Table. Methods for the SNS content table

The CMSNSContentList class that is the return type of the queryGetSNSContent() method stores a list of the retrieved SNS content. The content list is actually the Vector<CMSNSContent> type and we can get the list by calling the getConentList() method of the CMSNSContentList class. Then, we can access each SNS content object of which type is CMSNSContent class.

The CMSNSContent class defines a SNS content item and provides member methods to set and get various SNS attributes of the content as described in the following table.

Method name	Description
CMSNSContent() CMSNSContent(int id, String date, String writer, String msg, int nAttachment, int replyID, int lod)	Constructor
void setContentID(int id) int getContentID()	Set/get content ID.
void setDate(String date) String getDate()	Set/get date in which this content was created.
void setWriterName(String name) String getWriterName()	Set/get a writer name who created this content.
void setMessage(String msg) String getMessage()	Set/get a text message.
void setNumAttachedFiles(int num) int getNumAttachedFiles()	Set/get the number of attached files to this content.
void setReplyOf(int id) int getReplyOf()	Set/get a content ID to which this content is a reply content.
void setLevelOfDisclosure(int lod) int getLevelOfDisclosure()	Set/get the level of disclosure(lod) of this content. The lod value is one of 0 to 3. 0: open to public 1: open only to friends 2: open only to bi-friends 3: private
void setFilePathList(ArrayList<String> list) ArrayList<String> getFilePathList()	Set/get the list of attached file paths. For content upload, it requires full file paths. For content download, it retrieves only file names.

15.3.6 Management of the attached file table of SNS content

The following table shows methods to access and manage data in the default table for attached

files of SNS content.

Synopsis	Description
`int queryInsertSNSAttachedFile(int nContentID, String strFilePath, String strFileName, CMInfo cmInfo)`	Add an attached file nContentID: an associated content ID strFilePath: a directory of a file strFileName: a file name
`ArrayList<String> queryGetSNSAttachedFile(int nContentID, CMInfo cmInfo)`	Retrieve attached files nContentID: an associated content ID
`int queryUpdateSNSAttachedFile(int seqNum, String fieldName, String value, CMInfo cmInfo)`	Update a column value of the attached file table seqNum: a sequence number of the table fieldName: a column name of the table value: a value of the column
`int queryDeleteSNSAttachedFile(int seqNum, CMInfo cmInfo)`	Delete an attached file seqNum: a sequence number of the attached file table
`void queryTruncateSNSAttachedFileTable(CMInfo cmInfo)`	Clear the attached file table

Table. Methods for the attached file table

16. Multiple server management

In the client-server applications using CM, there is a single server which is the default server, and a client always interacts with the default server. Some distributed applications require multiple servers to provide a distributed processing functionality. To this end, CM allows a developer to implement and manage more than one CM server. An additional server can be developed with almost the same functionalities as the default server. The default server then manages all additional servers so that a client can also interact with them. In this section, we describe how an additional server is developed, participates in CM network, and interacts with clients.

16.1 Additional server configuration

Basically, the development and initialization procedure of an additional server with CM is the same as that of the default server. An additional server distinguishes from the default server when a developer sets the server configuration file. A developer should make sure that an additional server

has a different server configuration file from the default server even though the file name is the same (server-cm.conf). The part of the configuration file for an additional server is shown below.

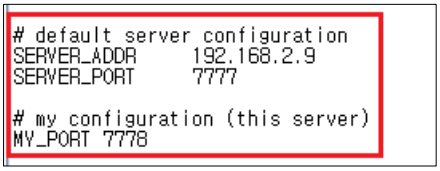


Figure. Configuration file (server-cm.conf) of an additional server

When it is initialized, the server CM checks whether its application is the default server or not by comparing the pair of the *SERVER_ADDR* and the *SERVER_PORT* values in the configuration file with the pair of the real IP address of this machine and the *MY_PORT* value of the configuration file. If two pairs are the same, this application becomes the default server. If the *SERVER_ADDR* value is different from the real IP address of this machine, or if the *SERVER_PORT* value is different from the *MY_PORT* value, this server is an additional server.

16.2 Management of an additional server

When an additional server starts, it automatically connects to the default server. The additional server then needs to request for registration to the default server in order to participate in current CM network. The CM server stub module provides a registration method as described below.

```
void requestServerReg(String server)
```

Figure. requestServerReg method

Only an additional server should call the *requestServerReg* method with a desired server name. Because the default server has the reserved name, "SERVER", the additional server must specify a different name as the parameter of this method. The following example shows how an additional server with a given name registers to the default server. In the same way, any number of additional servers can participate in the CM network.

```
// A server application has already initialized CM.
String strServerName = null;
System.out.println("====== request registration to the default server");
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Enter registered server name: ");
try {
        strServerName = br.readLine();
} catch (IOException e) {
        e.printStackTrace();
}
m_serverStub.requestServerReg(strServerName);
```

Figure. Example of server registration

In order for a requesting server to check the result of the registration request, the server can catch the *RES_SERVER_REG* event of the *CMMultiServerEvent* class in its event handler routine. The event fields of this event are described below.

Event Type		CMInfo.CM_MULTI_SERVER_EVENT	
Event ID		CMMultiServerEvent.RES_SERVER_REG	
Event field	Field Data type	Field definition	Get method
Server name	String	Requesting server name	getServerName()
Return code	int	Result code of the registration request 1: succeeded 0: failed	getReturnCode()

Table. RES_SERVER_REG event

If an additional server leaves current CM network, it can request to deregister from the default server by calling the *requestServerDereg* method of the CM server stub as shown below.

```
void requestServerDereg()
```

Figure. requestServerDereg method

Although it leaves the CM network, the additional server still maintains the connection with the default server. If required, this connection can also be managed by the methods of the CM server stub as shown below.

Synopsis	Description
`boolean connectToServer()`	Connect to the default server
`boolean disconnectFromServer()`	Disconnect from the default server

Figure. Connection management with the default server

16.3 Notification of additional server information

When the default server registers an additional server, it then notifies clients of the new server information. If a client is a late comer to the CM network, it can also explicitly request the information of additional servers from the default server by calling the *requestServerInfo* method of the CM client stub. The synopsis of this method is described below.

```
void requestServerInfo()
```

Figure. requestServerInfo method

In any of the above two cases, the default server sends the *NOTIFY_SERVER_INFO* event of the *CMMultiServerEvent* class. This event contains the list of additional server information such as a server name, address, port number, and UDP port number. The detailed event fields of the *NOTIFY_SERVER_INFO* event is described below.

Event Type		CMInfo.CM_MULTI_SERVER_EVENT	
Event ID		CMMultiServerEvent.NOTIFY_SERVER_INFO	
Event field	Field Data type	Field definition	Get method
Number of servers	int	Number of additional servers	getServerNum()
Server list	Vector<CMServerInfo>	List of additional server information	getServerInfoList()

Table. NOTIFY_SERVER_INFO event

When the default server deletes an additional server by the deregistration request, it then sends the *NOTIFY_SERVER_LEAVE* event to clients. The event fields of the event are described below.

Event Type		CMInfo.CM_MULTI_SERVER_EVENT	
Event ID		CMMultiServerEvent.NOTIFY_SERVER_LEAVE	
Event field	Field Data type	Field definition	Get method
Server name	String	Name of an additional server that leaves the CM network	getServerName()

Table. NOTIFY_SERVER_LEAVE

A client can figure out current available additional servers by catching the *NOTIFY_SERVER_INFO* and *NOTIFY_SERVER_LEAVE* events in the client event handler. The following example shows the relevant part of the client event handler method where a client receives these events and prints out the received information.

```
…
private void processMultiServerEvent(CMEvent cme)
{
        CMMultiServerEvent mse = (CMMultiServerEvent) cme;
        switch(mse.getID())
        {
        case CMMultiServerEvent.NOTIFY_SERVER_INFO:
                System.out.println("New server info received: num servers:
"+mse.getServerNum() );
                Iterator<CMServerInfo> iter = mse.getServerInfoList().iterator();
                System.out.format("%-20s %-20s %-10s %-10s%n", "name", "addr",
"port", "udp port");
                System.out.println("----------------------------------------
----------------");
```

```
                while(iter.hasNext())
                {
                        CMServerInfo si = iter.next();
                        System.out.format("%-20s %-20s %-10d %-10d%n",
si.getServerName(), si.getServerAddress(), si.getServerPort(),
si.getServerUDPPort());
                }
                break;
        case CMMultiServerEvent.NOTIFY_SERVER_LEAVE:
                System.out.println("An additional server["+mse.getServerName()+"]
left the " + "default server.");
                break;
        }

        return;
}
```

Figure. Handling additional server information

16.4 Client's interaction with an additional server

Once a client receives the information of additional servers from the default server, it can then interact with any number of additional servers with establishing the corresponding number of additional connections while keeping the connection with the default server. That is, now a client can send and receive events with multiple servers in a distributed and independent manner. The process of login/logout and joining/leaving a session and a group of an additional server is the same as that of the default server except that a client now must specify a server name. The following table enumerates the methods of the CM client stub for the interaction with a designated server.

Synopsis	Description
`boolean connectToServer(String strServerName)`	Connect to a specific server strServerName: server name
`boolean disconnectFromServer(String strServerName)`	Disconnect from a specific server strServerName: server name
`void loginCM(String strServer, String strUser, String strPasswd)`	Request to log in a specific server strServer: server name strUser: user name strPasswd: user password
`void logoutCM(String strServer)`	Log out from a specific server strServer: server name
`void requestSessionInfo(String strServerName)`	Request session information of a specific server strServerName: server name
`void joinSession(String strServer, String strSession)`	Join a session of a specific server strServer: server name

	strSession: session name
`void leaveSession(String strServer)`	Leave a current session of a specific server strServer: server name

Table. Interaction methods with a designated server

With the above methods, a client can interact with any server including the default server. For example, if a client sets "*SERVER*" to the server name parameter, the designated server is the default server.

16.5 Event transmission to other clients of a designated server

To send an event, a client or a server can call one of the *send*, *cast*, *multicast*, and *broadcast* methods according to the chosen transmission mode. These methods can be used without a problem when a client sends an event to a specific server or when a server sends an event to its clients. However, if a client sends an event to other clients with one-to-one communication, these methods always send to other clients of the default server. In order to send an event to other clients of a different server, CM provides the extended version of the transmission methods where a sender client can specify a server. The following table enumerates such methods.

Synopsis	Description
`boolean send(CMEvent cme, String serverName, String userName)`	Send an event to a user of a specific server cme: CM event to send servername: target server name userName: target user name
`boolean send(CMEvent cme, String serverName, String userName, int opt)`	Send an event to a user of a specific server cme: CM event to send servername: target server name userName: target user name opt: transmission option
`boolean send(CMEvent cme, String serverName, String userName, int opt, int nChNum) (not supported yet)`	Send an event to a user of a specific server cme: CM event to send servername: target server name userName: target user name opt: transmission option nChNum: sending transmission channel
`boolean cast(CMEvent cme, String serverName, String sessionName, String groupName)`	Send an event to a group of users of a specific server

	cme: CM event to send serverName: target server name sessionName: target session name groupName: target group name
`boolean cast(CMEvent cme, String serverName, String sessionName, String groupName, int opt)`	Send an event to a group of users of a specific server cme: CM event to send serverName: target server name sessionName: target session name groupName: target group name opt: transmission option
`boolean cast(CMEvent cme, String serverName, String sessionName, String groupName, int opt, int nChNum)` (not supported yet)	Send an event to a group of users of a specific server cme: CM event to send serverName: target server name sessionName: target session name groupName: target group name opt: transmission option nChNum: sending transmission channel

Table. Extended event transmission methods

Currently, a client or a server cannot call the multicast and broadcast methods for a different server, but we are researching on an efficient way to support such functionalities.

References

[1] M. Lim, B. Kevelham, N. Nijdam, N. Magnenat-Thalmann, "Rapid Development of Distributed Applications Using High-level Communication Support," *Journal of Network and Computer Applications*, Vol.34, No.1, January 2011, pp.172-182.

[2] M. Lim, "Adaptation of content transmission for social network systems," *International Journal of Information Processing and Management (IJIPM)*, Vol. 4, No. 6, pp.60-67, September 2013.

[3] M. Lim, "Improving the Architecture of Communication Middleware for Social Networking Services," *International Journal of Software Engineering and Its Applications*, Vol. 9, No. 11, November 2015, pp. 25-41.

[4] Y. Lee, M. Lim, Y. Moon, "Application-level Communication Services for Development of Social Networking Systems," *International Journal of Electrical and Computer Engineering (IJECE)*, Vol.5, No.3, June 2015, pp. 586-598.

[5] M. Lim, “Multi-level Content Transmission Mechanism for Intelligent Quality of Service in Social Networking Services," *The Transactions on the Korean Institute of Electrical Engineers*, Vol. 65, No. 8, August 2016, pp. 1407-1417.

[6] M. Lim, "Prefetching Mechanism of Image Content Based on Users of Interest in Social Networking Services," *Information - An Interdisciplinary Journal*, Vol.20, No.1B, January 2017, pp. 631-642.

[7] M. Lim, "CMSNS: A Communication Middleware for Social Networking and Networked Multimedia Systems," *Multimedia Tools and Applications*, Vol.76, No.17, September 2017, pp.18119-18135.

[8] M. Lim, "Supporting Synchronous and Asynchronous Communications in Event-based Communication Framework for Client-Server Applications," *International Journal of Advanced Computer Research*, Vol.9, Issue 40, January 2019, pp.11-19.

[9] M. Lim, "Directly and Indirectly Synchronous Communication Mechanisms for Client-server Systems Using Event-based Asynchronous Communication Framework," *IEEE Access*, Vol. 7, Issue 1, June 2019, pp.81969-81982.

[10] Y. Moon, M. Lim, "An Enhanced File Transfer Mechanism Using Additional Blocking Communication Channel and Thread for IoT Environments," *Sensors*, Vol.19, Issue 6, 13 March 2019, pp.1271.

