IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 22, 2019, accepted June 19, 2019, date of publication June 24, 2019, date of current version July 8, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2924497

Directly and Indirectly Synchronous Communication Mechanisms for Client-Server Systems Using Event-Based Asynchronous Communication Framework

MINGYU LIM
Department of Smart ICT Convergence, Konkuk University, Seoul 05029, South Korea
e-mail: mlim@konkuk.ac.kr

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education under Grant NRF- 2018R1D1A1A09082580.

ABSTRACT In this paper, we propose a synchronous communication mechanism that can be used with asynchronous communication in an event-based communication framework (CM). The proposed synchronous communication mechanism supports directly synchronous communication between client and server, indirectly synchronous communication between two clients through server mediation, and indirectly one-to-many synchronous communication with multiple clients. In order to support synchronous communication, the CM uses the synchronization mechanism between the main thread and the processing thread while maintaining the multi-thread structure. The application can utilize both the asynchronous and the synchronous communication service of the CM together according to its requirements. For performance analysis, we compared the CM synchronous and asynchronous communication methods with qualitative and quantitative experiments, respectively. Through qualitative analysis, developing applications using synchronous services can more intuitively design application logic than asynchronous services. From the quantitative experiment, we also verified that the response delay time in the synchronous method is shorter than that in the asynchronous method although the difference in the response delay is not large.

INDEX TERMS Asynchronous communication, client-server system, event-based communication framework, synchronous communication, thread synchronization.

I. INTRODUCTION

Most distributed system applications, such as online games, social networking services, cloud services, and IoT, communicate with each other using a client-server or peer-to-peer model of communication structure. Most applications still follow the client-server model, rather than the peer-to-peer architecture that completely replaces a centralized node, such as a block chain. In the distributed system of the client-server model, the client requests a service from the server, and the interaction occurs between the nodes in the request-reply pattern in which the server responds to the request to the client. The communication method between the client and the server is divided into the synchronous and asynchronous methods [1]. In the synchronous mode, a client requests a service from a server and enters a waiting state without performing the next task until it receives a response from the server. Upon receiving the response from the server, the client proceeds to perform the next task. In the asynchronous mode, after the client requests the service from the server, it continues the execution to the next task without waiting the response.

Communication functions required by the application can be developed by the application developer, but the application development efficiency can be improved by using the service of the communication middleware instead. Existing communication middleware and framework provide various communication services specialized for general purpose or a specific target application. However, they support only the asynchronous method [2]–[6] or do not clearly mention the

The associate editor coordinating the review of this manuscript and approving it for publication was Charith Perara.

support of synchronous and asynchronous communication methods [7]–[9]. Even though there are researches supporting both the synchronous and asynchronous methods, they can be used only in limited situations [10]–[12], or it is not clearly described [13]–[16]. The communication method used by the communication middleware is determined not by the application requirements but by the communication channel management method or the type of the service in the middleware. Therefore, when the application selects middleware service, its communication method is also fixed. For example, as the communication middleware (CM) [2], [3] that is our previous work was developed as an event-based asynchronous communication framework, all communication services are supported asynchronously. However, because all kinds of services can be performed synchronously and asynchronously, they could become more flexible services if applications can choose one according to their requirements.

In this paper, we propose a significantly extended version of CM that supports both synchronous and asynchronous communication methods [17]. We introduce the communication service provided by CM to application and event-based asynchronous communication structure using multiple threads. In particular, we discuss not only the asynchronous communication method that was previously supported in CM, but also the one-to-one directly synchronous communication with the server, the one-to-one indirectly synchronous communication with the client, and the one-to-many indirectly synchronous communication method with multiple clients. To support the asynchronous communication, CM separates the main thread that receives the user's local events and sends them to the server, and the processing thread that processes incoming events from the server. To support the synchronous communication while maintaining this multi-threaded structure, synchronization between the main thread and the processing thread is required. That is, when the client application requests the CM's synchronous communication service, the main thread sends the request event to the target node and suspends its execution. When the processing thread receives the response event for the request from the target node, it wakes up the main thread and forwards the response event. In one-to-one directly synchronous communication, the client waits until the server directly sends a response event. In one-to-one indirectly synchronous communication, the server delivers the response event sent by the target client to the request client. In one-to-many indirectly synchronous communication, since multiple clients send response events, the requesting client can additionally specify the number of response events to wait.

The application can use both the synchronous and asynchronous communication services provided by CM. We compared and analyzed the characteristics of synchronous communication and asynchronous communication methods by qualitatively analyzing each case and quantitatively measuring response delay time. The qualitative analysis shows that application logic design is more intuitive when developing applications using synchronous services.

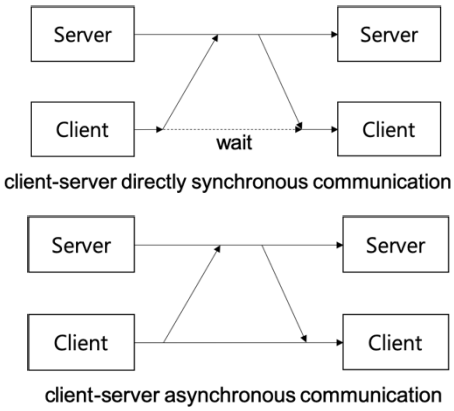


FIGURE 1. Synchronous vs. asynchronous communication.

The quantitative analysis shows that the response delay time in the synchronous method is shorter than the response delay time in the asynchronous method, but that the delay does not show a large difference.

The remainder of this paper is organized as follows. In Section II, we compare the concepts of synchronous and asynchronous communication, and extend the existing synchronous communication to describe the concepts of directly synchronous, indirectly synchronous, and one-to-many indirectly synchronous communication addressed in this paper. In Section III, we introduce our previous CM that provided only asynchronous communication service to the application. In Section IV, we describe how to support various synchronous communication methods in CM. In Section V, we compare and analyze the proposed synchronous communication service and the existing asynchronous communication service from the qualitative and quantitative perspectives. Section VI describes the current state of synchronous and asynchronous communication support in existing communication framework and middleware. Section VII describes future work and concludes this paper.

II. ASYNCHRONOUS vs. SYNCHRONOUS COMMUNICATION

In a client-server application, the communication method between the client and the server usually follows the request-reply pattern where a pair of the client's request message and the server's response message composes a communication service. The communication between the server and the client can be classified as synchronous and asynchronous as shown in Figure 1.

The definition of synchronous and asynchronous communication is different according to its context and communication level [1], [18]–[20]. In this paper, we follow the definition of [1]. In the synchronous communication mode, the client sends a service request message to the server and waits without performing the next task until the server sends a response message. After receiving the response message from the server, the client proceeds to the next execution. In the asynchronous communication method, a client can

send a request message to a server and then immediately perform another task without waiting for a response from the server. For example, it is possible to process the next local input from the user while waiting for a response from the server. Instead, in the asynchronous communication, there is a need for a mechanism that allows the client to asynchronously receive a response message independent of performing other local tasks.

In order to improve the interaction performance, it is effective to selectively determine a suitable communication method depending on the type and characteristics of the service requested by the client. The main criterion for determining the communication method is whether there are other tasks to be performed while the client waits for the result of the service request. It is simple to use the synchronous communication method if the client can perform other tasks only after it receives the response for the request. If there are other tasks to perform while the client is waiting for the result of the request, it is more efficient to use the asynchronous communication method. For example, if a client requests a file transfer service, then the synchronous communication method is simple if the client has to do the following tasks with the file after the file transfer is complete. Conversely, if there are other tasks that the client needs to perform while the file is being transferred, it is desirable to request the file transfer service asynchronously.

In the client-server architecture, the synchronous communication method can be divided into directly synchronous and indirectly synchronous communications. The directly synchronous communication means a synchronous method between a client and a server which are directly connected through a channel, and it is well applied to the general request-reply pattern. The indirectly synchronous communication means a case where clients perform synchronous communication with each other through a server. The indirectly synchronous communication method can be used when a client is necessary to proceed to the next task only after the receipt of confirmation from another client during the execution of a series of tasks. Figure 2 shows how two clients perform indirectly synchronous communication. First, client 1 sends a confirmation request message to client 2. This message is delivered to client 2 through the server. Client 2 sends a response message after processing the request of client 1. The response message is transmitted to the client 1 through the server. The client 1 waits until it receives the response message of the client 2.

As shown in Figure 3, we also define a one-to-many indirectly synchronous communication as a case where a client performs a synchronous communication with multiple clients. In this communication method, client 1 sends a request message to n clients (clients 2 to n + 1) via the server. The client receiving the request message transmits a response message to the request client 1 via the server. Client 1 waits until it receives n response messages, and then executes the next task. An additional consideration in the one-to-many indirectly synchronous communication

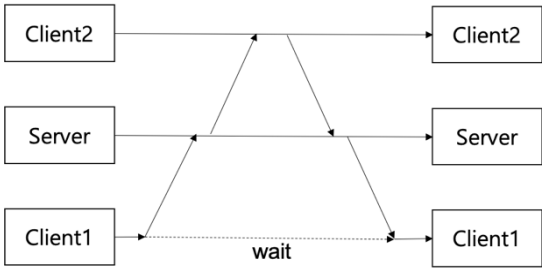


FIGURE 2. One-to-one indirectly synchronous communication.

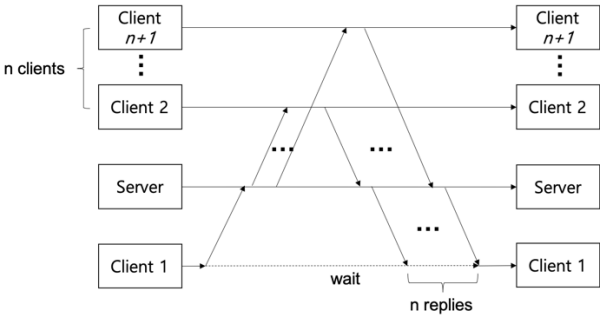


FIGURE 3. One-to-many indirectly synchronous communication.

scheme is that the requesting client can decide how many response messages it should wait for. The requesting client can determine the minimum number of response messages in the range of 1 to n according to the request situation. For example, if a response from all clients is needed, the requesting client waits until it receives n response messages. If the requesting client needs responses only from a majority of clients, it waits until it receives at least n / 2 response messages.

III. COMMUNICATION FRAMEWORK (CM)

CM [2], [3] is a communication framework for developing distributed applications. Application developers can use the Application Programming Interfaces (APIs) and configuration files provided by CM to easily implement various communication functionalities with which CM nodes (applications that use CM's communication services) can interact with each other. A CM node can distinguish its role between a client and a server. The CM client node can call basic communication functions through CM such as registration, connection, log-in, creation and transmission of CM events, receiving and processing CM events. In addition, the client node can use other communication services such as file transfer, content management for social networking service (SNS), network status checking, and communication channel management. All the CM services used by the client consist of the request-reply pattern as shown in Figure 4. When a client requests a service, the client CM creates the request event and sends it to the server. When the server performs the requested service, the server CM creates a response event and sends it back to the client.

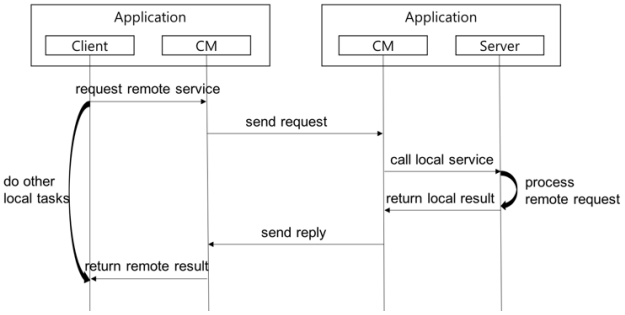


FIGURE 4. Request-reply pattern of CM services.

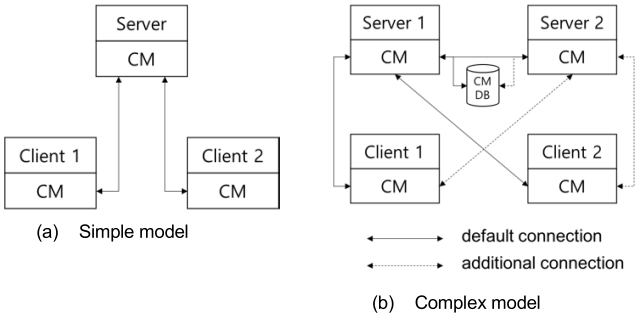


FIGURE 5. CM client-server architectures.

TABLE 1. CM services and options.

Service type	Service name	Options
Communication management	Communication architecture	Client-server, peer-to-peer, hybrid model
Persistency management	Database table management	Database usage on/off
User management	User registration, deregistration, login, logout	Login with or without authentication
Friend management	Friend addition, deletion, search	Different friend types (unilateral or bilateral friends)
SNS management	Content upload, download	File attachment, different level of content disclosure
Chat management	Chatting	Different chatting target (one user, friends, group members, session members, all users)
File transfer management	File transfer	File push or pull modes
Event management	Event composition and transmission	Different transmission modes (one-to-one, one-to-many) with TCP or UDP

A. FLEXIBLE APPLICATION-LEVEL COMMUNICATION SERVICES

In addition to providing a variety of communication services, CM also provides more flexible communication services by providing various options depending on the requirements of the application. Table 1 summarizes the core communication services of CM. By simply changing the CM configuration file, the connection model of the application can be determined by the client-server structure or the peer-to-peer structure. In addition, it is possible to determine whether or not the application uses the DB internally according to the configuration file. In the user management, it is also possible to determine whether the user will go through a user authentication process to join the CM network. In the management of user friend, it is possible to define a different type of a friend: unilateral and bilateral friends. In the SNS content management, an arbitrary file can be attached to the content uploaded by the user, and the disclosure range of the content can be variously specified. A user can choose either push or pull in the file transfer service. One-to-one or one-to-many transmission can be selected by selecting either reliable transmission (TCP) or unreliable transmission (UDP) in the event transmission service.

Utilizing CM's diverse communications services and options, developers can easily build distributed applications from simple to complex structures to meet their requirements. For example, as shown in Figure 5, a developer can call CM APIs to quickly build a simple client-server application that uses a single communication channel, or a complicated client-server system that consists of multiple servers over multiple communication channels, and that uses database (DB). Since CM takes care of most communication functions between applications, application developers can focus more on developing application-specific features, thus improving application development efficiency.

B. EVENT-BASED ASYNCHRONOUS COMMUNICATION FRAMEWORK

Existing CM nodes communicate internally in an event-based asynchronous manner. The advantage of the asynchronous communication method compared to the synchronous method is that the application does not have to wait for the response event for the request. Accordingly, the CM application can perform other tasks after requesting the communication service of the CM. Instead, the application should register a callback function dedicated to receiving an event from CM so that the application can receive a response to the request from CM at any time during the execution of another task. Since CM invokes the registered callback function each time it receives an event, the application can process the response event received at any instant.

To support the event-based asynchronous communication scheme, CM runs multiple threads (main thread, processing thread, sending thread, receiving thread) as shown in Figure 6. First, when the main thread of the application runs the CM, it starts the processing thread, the sending thread, and the receiving thread. The main thread is responsible for handling local events from application users. The processing thread is responsible for receiving and processing CM events received from the receiving thread. In order for an application to process a CM event, an event handler object including an event processing callback function is first registered in CM. If necessary, the processing thread processes the received CM event internally first, and then calls the event processing

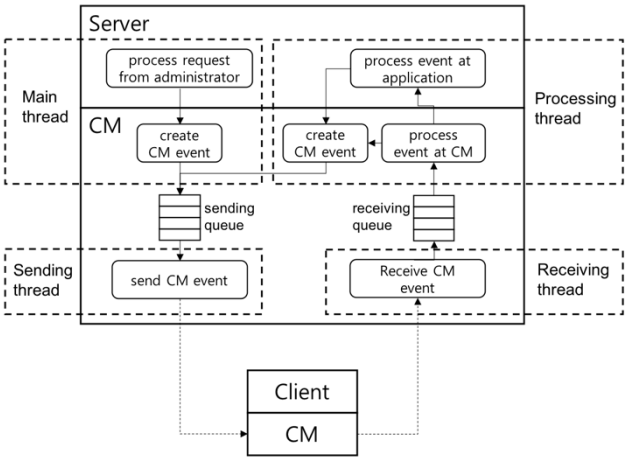


FIGURE 6. CM multi-threaded architecture.

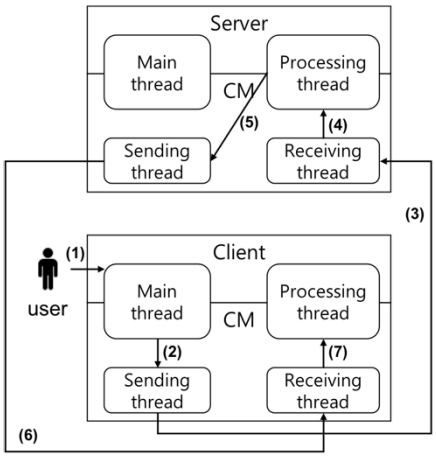


FIGURE 7. Event-based asynchronous communication of CM.

function of the application event handler to enable separate event processing required by the application.

The sending thread is responsible for converting the CM event created by the main thread or the processing thread into a low-level byte message and transmitting it to the network. When the main thread or the processing thread needs to send a message to a remote node, it creates a necessary CM event, puts it in a transmission queue, and forwards it to the sending thread. The receiving thread is responsible for receiving the byte message from the network and converting it into a CM event and passing it to the processing thread. The receiving thread puts the received CM event into the reception queue, and the processing thread receives and processes the received CM event.

Asynchronous communication between the client and the server by the CM's multithreaded architecture is implemented as follows. Figure 7 shows a typical communication procedure between a client and a server using CM. First, the user requests a local or remote service using the client. (1) The user's request is handled by the client's main thread. To process the remote service request, the client's main thread creates a request event and forwards it to the sending thread. (2) The main thread can immediately process the next request of the user without waiting for the result of this request. In the meantime, the server processes the request event. (3) - (4) The server's processing thread creates a response event, if necessary, and sends it back to the client via the sending thread. (5) - (6) The client receives the response event and sends the response event to the processing thread (7), and the processing thread processes the response event. Independent of the processing thread, the client's main thread continues to process multiple requests from the user.

IV. SYNCHRONOUS COMMUNICATION SUPPORT

CM essentially supports event-based asynchronous communication between client and server applications. In addition, it also provides a synchronous communication mechanism depending on the requirements of the application using CM, or depending on the characteristics of the communication service provided by CM. The synchronous communication scheme provided by CM uses a synchronization mechanism between threads and a non-blocking socket channel in a multithreaded environment. The following subsections describe in detail how CM supports synchronous communication.

A. SYNCHRONOUS COMMUNICATION WITH NON-BLOCKING SOCKET CHANNEL

In the existing CM service, since the main thread and the processing thread are separated, the client asynchronously receives the response to a request. Thus, after the client requests the CM service from the main thread, the response from the server to it is received independently in the processing thread. However, in the CM communication service, the client sometimes needs to receive the response from the server synchronously. For example, there may be a situation where a client needs to transmit data through the new socket channel after creating one using the CM channel addition service. In order to do this, the client can use the channel only after confirming that the new socket channel information is registered in the server by receiving the response message from the server. In the conventional asynchronous channel addition service, the client cannot wait to receive a response message from the server after calling the channel addition API function. Instead, it should check whether or not the response message is received in a separate way. If the channel addition service is provided synchronously, the client can directly receive the response message from the server as a return value of the API function after calling the channel addition service. The client is in a waiting state until the function returns a return value. Therefore, if the client receives a return value after calling the service request function, it can use the newly added channel without any confirmation. This synchronous communication is similar to the remote procedure call method. Figure 8 shows the overall procedure for CM to support synchronous communication between the client and the server using the original non-blocking socket channel.

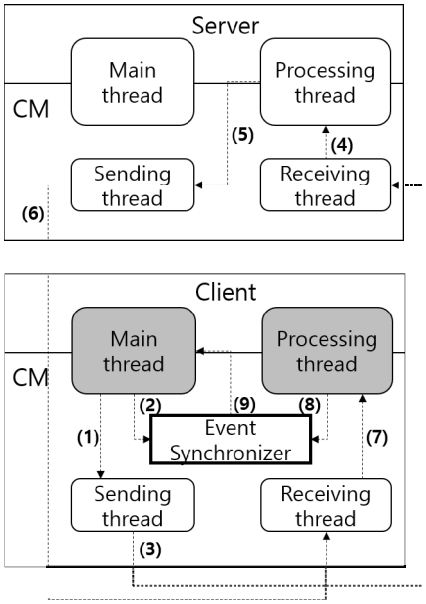


FIGURE 8. Synchronous communication using non-blocking socket channel.

First, the user requests the CM communication service synchronously through the client. (1) The main thread of the client CM creates a CM request event corresponding to the user's request and delivers it to the sending thread through the message transmission queue. (2) The main thread registers the response event type to wait in the event synchronization object. This thread gets a lock on the synchronization object and waits for the response from the server, by calling the wait() function. After this operation, the user cannot make another service request to the client. (3) The sending thread sends a message to the server CM over the default non-blocking socket channel. The receiving thread of the server CM monitors the selector object and receives a service request message. (4) The received message is delivered to the processing thread via the message receiving queue. (5) After processing the request of the client, the processing thread creates a response event and forwards it to the sending thread. (6) The sending thread sends the response event to the client over the default non-blocking socket channel. The receiving thread of the client CM monitors its selector object and receives the response event. (7) The receiving thread forwards the received event to the processing thread. After processing the received event, the processing thread checks whether this event is registered in the event synchronization object or not. (8) If the received event type is the same as the event type registered in the synchronization object, the processing thread registers this response event in the event synchronization object. The processing thread then wakes up the main thread in the wait state by obtaining a lock on the synchronization object and calling the notify() function. (9) The main thread wakes up from the waiting state, checks the response event registered in the event synchronization object, and returns the service result value to the application. The client application receives the result from the return

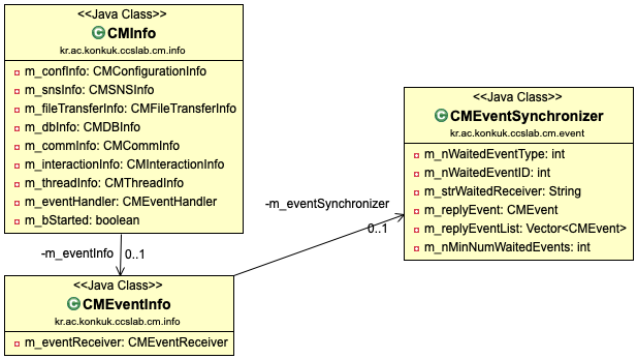


FIGURE 9. Class diagram of event synchronizer and relevant classes.

value of the service request function and performs the next task.

In the synchronous communication using such a non-blocking socket channel, the main thread and the processing thread of the client are synchronized with each other such that the main thread is in a waiting state until it receives the request result. On the other hand, a server receiving a request does not have to wait for a specific thread to process the request from the client, but performs its tasks in the same manner as the conventional asynchronous communication method.

B. EVENT SYNCHRONIZER

The event synchronizer is a core module added to CM to support synchronous communication. Figure 9 is a class diagram showing the relationship between the event synchronizer class and other related CM classes. To simplify the relationship between classes, only member variables are shown for each class, and member functions are omitted.

The CMInfo class contains all the internal state information managed by CM. To this end, the CMInfo class contains various kinds of classes for each purpose. Table 2 shows the role of each information class. Among them, the CMEventInfo class is a class for storing information related to event processing, and includes a CMEventReceiver (CMEventReceiver) responsible for CM event reception and an event synchronizer object (CMEventSynchronizer).

The CM main thread and the processing thread are synchronized through the event synchronizer object as follows. First, the main thread sends a request event with a lock on the synchronizer object, and then calls the wait() function to enter the waiting state. The processing thread receives the response event and check whether the event is the one that the main thread is waiting for or not. If the condition is satisfied, the processing thread also acquire the lock on the synchronizer object, and calls the notify() function to awaken the waiting main thread from the waiting queue. The role of the event synchronizer class is described by member variables as follows. The main thread of the client CM registers the event type (m_nWatiedEventType) and the identifier (m_nWaitedEventID) to the event synchronizer

TABLE 2. CM classes for internal state information.

Class name	Role
CMConfigurationInfo	To store CM configuration information
CMSNSInfo	To store CM information related to the online social network service
CMFileTransferInfo	To store information related to the CM file transfer service
CMDBInfo	To store information related to database used by CM
CMCommInfo	To store socket channel information
CMInteractionInfo	To store current CM server, session, user information
CMThreadInfo	To store thread pool information
CMEventInfo	To store information related to CM events
CMEventHandler	To store the reference to an event handler of application

TABLE 3. CM synchronous APIs.

Function name	Description	Synchronous communication type
CMSessionEvent syncLoginCM(String strUserName, String strPassword)	Login service	One-to-one direct
CMSessionEvent syncRequestSessionInfo()	Retrieval of available session information	One-to-one direct
CMSessionEvent syncJoinSession(String sname)	Join-session service	One-to-one direct
SocketChannel syncAddBlockSocketChannel(int nChKey, String strServer)	Addition of a new blocking socket channel	One-to-one direct
boolean syncRemoveBlockSocketChannel(int nChKey, String strServer)	Deletion of a blocking socket channel	One-to-one direct
CMEvent sendrecv(CMEvent cme, String strReceiver, int nWaitedEventType, int nWaitedEventID, int nTimeout)	To send a CM event to a target node and wait to receive a response event	One-to-one direct/indirect
CMEvent[] castrecv(CMEvent event, String strSessionName, String strGroupName, int nWaitedEventType, int nWaitedEventID, int nMinNumWaitedEvents, int nTimeout)	To send a CM event to a group of target nodes and wait to receive response events	One-to-many indirect

object to wait synchronously after sending the request event to the server. For synchronous communication with another client, the requesting client also registers a target client (m_strWaitedReceiver) that is supposed to send the response event. The processing thread of CM can deliver the received event to the main thread by registering it to the synchronizer object (m_replyEvent). When the client uses one-to-many indirectly synchronous communication, the minimum number of response events (m_nMinNumWaitedEvents) to be waited for by the main thread is registered in the synchronizer object, and the processing thread delivers a list of received response events to the main thread by registering the list to the synchronizer object.

C. SYNCHRONOUS API

In the client application, the main thread takes the user's local input, processes it, and delivers the result to the user. The role of the main thread for synchronous communication is to invoke the synchronous API of CM according to the request of the client user and to wait until it receives a response event for the request event transmitted from this API. The client can request a CM service by calling the API through the CM stub module. Table 3 shows the APIs provided by CM to support synchronous communication.

The APIs using the directly synchronous communication provide specific CM services. The client main thread requests the service, and it waits until the response event from the server is received. CM predefines the type and the identifier (ID) of the response event to each service. When the response event is received, it is returned to the client main thread in the form of a return value of the request function.

In addition to the specific synchronous services provided by CM, if the client needs arbitrary synchronous communication, it can use generalized version of the directly and indirectly synchronous communication via the sendrecv() function. The required parameters in this case are the request event, the receiver of the request, the type and the ID of the response event to wait, and the maximum time to wait for the response event. If the recipient of the request event is a server, it becomes a directly synchronous communication, and if the receiver is another client, it becomes an indirectly synchronous communication. In the case of indirectly synchronous communication, the client's request event is sent to the server and the server CM internally forwards the request event to the final target client. The client's response event is passed back through the server to the requesting client. If the client fails to receive the response event or the timer expires, the function returns the NULL value.

The client can also use one-to-many indirectly synchronous communication via the castrecv() function. In the one-to-many indirectly synchronous communication, the destination of the request event is designated as a client group. The target client group uses the session and group concept defined in CM. Clients participating in the CM network always belong to at least one session and one group. The server can have multiple sessions and a session can have multiple groups. The client can specify the group of recipients of the request event by session ID and group ID. The requesting client can specify the type and ID of the response event to receive, as well as the minimum number of response events to wait. That is, the castrecv() function returns an array of response events when it receives the minimum number of response events. The function returns NULL when the timer expires before receiving the required minimum number of response events.

```
Function: sendrecv
Input: cme (request event), strReceiver (receiver), nWaitedEventType
(reply event type), nWaitedEventID (reply event ID), nTimeout
(maximum waiting time)
Output: replyEvent (reply event)

01: eventSync = CM event synchronizer
02: initialize eventSync
03: set nWaitedEventType, nWaitedEventID, strReceiver to eventSync
04: send cme to strReceiver
05: get lock of eventSync // start critical section
06: eventSync.wait(nTimeout) // wait reply event in the waiting queue
of eventSync
07: release lock of eventSync // end critical section
08: replyEvent = eventSync.getReplyEvent()
09: return replyEvent
```

FIGURE 10. Algorithm of sendrecv() function.

When the client calls the CM synchronous API, the operation algorithm of the main thread is almost the same in different types of synchronous communication. Figure 10 shows the algorithm of the sendrecv() function. The information registered in the event synchronizer object is a request receiver and response event information. In the directly synchronous API, the receiver is the server and service-specific response events are already defined and are not provided as a separate parameter. In the castrecv() function, the recipient information registered in the synchronizer object is changed into a session and a group. The main thread obtains a lock on the synchronizer object, enters the critical section (line 5) first, and unlocks it before entering the waiting state (line 6). When the processing thread wakes up the main thread, it reacquires the lock, then continues execution and releases the lock, leaving the critical section. (line 7) The response event registered by the processing thread can be obtained through the synchronizer object. (line 8)

D. SYNCHRONOUS PROCESS OF RECEIVED EVENTS

The processing thread of CM internally processes the event received through the receiving queue, and if necessary, calls the callback function of the event handler registered in CM to deliver the event to the application. The detailed operation of the processing thread for synchronous communication is as follows. After completing the processing of the received event, the processing thread checks whether the main thread is waiting to receive this event. The checking process is whether the flag value indicating the waiting state of the main thread in the event synchronizer object is TRUE, whether the type and the ID of the received event are the same as those of the waited-for event, and whether the event sender is the same as the receiver of the request event. If this condition is met, the processing thread stores the received event in the synchronizer object and wakes up the main thread that was in the waiting state in the synchronizer object. As with the main thread entering the waiting state, for synchronization, the processing thread must first obtain the lock on the synchronizer object and wake up the main thread in the critical section. Finally, the processing thread releases lock on the synchronizer object.

```
eventSync = CM event synchronizer
while processing thread is not interrupted
Begin
    CMEvent cme = get received event from the receiving queue
    process cme in CM
    if( eventSync.isWaiting() && cme.getSender() ==
eventSync.getWaitedReceiver() && cme.getType() ==
eventSync.getWaitedEventType() && cme.getID() ==
eventSync.getWaitedEventID() )
        Begin
        eventSync.setReplyEvent(cme)
        get lock of eventSync // start critical section
        eventSync.notify() // wake up the main thread
        release lock of eventSync // end critical section
        End
    else if( eventSync.isWaiting() && eventSync.getWaitedEventType()
== cme.getType() && eventSync.getWaitedEventID() == cme.getID()
&& eventSync.getMinNumWaitedEvents() >= 0 )
        Begin
        eventSync.addReplyEvent(cme)
        Get lock of eventSync // start critical section
        if( number of reply events in eventSync ==
eventSync.getMinNumaitedEvents() )
            eventSync.notify() // wake up the main thread
        Release lock of eventSync // end critical section
        End
End
```

FIGURE 11. Algorithm of processing thread.

If the above condition of one-to-one synchronous communication is not satisfied, the processing thread needs to check whether the received event satisfies the condition of one-to-many synchronous communication. The condition is whether the main thread is waiting in the event synchronizer object, whether the received event is the same as the waited-for event, and whether the minimum number (that is greater than one) of response events is registered in the synchronizer object. If this condition is satisfied, the processing thread adds the received event to the synchronizer object and again checks whether the number of events stored in the synchronizer object equals to the minimum number of waited-for events. If the second condition is satisfied, the response condition of the one-to-many synchronous communication is satisfied, and the processing thread can wake up the main thread.

If the synchronous communication condition is satisfied, the processing thread does not forward the event to the client's event handler since the received event will be processed in the main thread. If the synchronous communication condition is not satisfied, the event is handed back to the event handler as necessary according to the internal processing result of the received event so that the client application can process the event asynchronously. Figure 11 shows the overall algorithm of the processing thread for synchronous communication.

V. PERFORMANCE EVALUATIONS

In this chapter, we compare and analyze the performance of synchronous and asynchronous communication methods provided by CM in terms of both quantitative and qualitative methods. The qualitative evaluation compares the usability of communication services that support both synchronous and asynchronous methods among the CM APIs. For quantitative performance analysis, we measured the response time of the

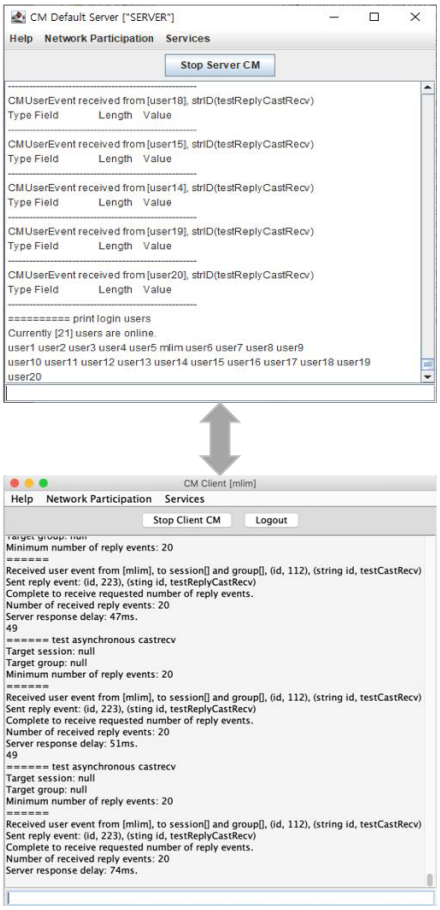


FIGURE 12. CM client and server application.

target node when the synchronous and asynchronous APIs are called for each CM communication service.

A. CM CLIENT AND SERVER APPLICATIONS

For the qualitative and quantitative analysis of CM, we developed a test client and a server application using CM with Eclipse IDE and Java Swing package to compose a simple Graphical User Interface (GUI) environment. The main goal of this application is to test and verify various CM services in the application's perspective. In the client, a user can call all CM services including the synchronous and asynchronous APIs. The server application mainly prints out requests from clients and their results that are delivered by CM. Figure 12 shows a screenshot of the CM client and server applications.

To participate in the CM network, every CM client always should log in to the CM server, and then should join a session within the server. A session is an interaction unit that defines an independent group of clients. That is, a client can interact with the other clients within the same session. To join a session, the client needs to request available sessions from the server. After the login process, the client can establish an additional communication channel, which is dedicated to the file transfer service.

TABLE 4. Synchronous and asynchronous APIs of CM.

Synchronous APIs of client CM	Asynchronous APIs of client CM
CMSessionEvent syncLoginCM(String strUserName, String strPassword)	boolean loginCM(String strUserName, String strPassword)
CMSessionEvent syncRequestSessionInfo()	boolean requestSessionInfo()
CMSessionEvent syncJoinSession(String sname)	boolean joinSession(String sname)
SocketChannel syncAddBlockSocketChannel(int nChKey, String strServer)	boolean addBlockSocketChannel(int nChKey, String strServer)
boolean syncRemoveBlockSocketChannel(int nChKey, String strServer)	boolean removeBlockSocketChannel(int nChKey, String strServer)
CMEvent sendrecv(CMEvent cme, String strReceiver, int nWaitedEventType, int nWaitedEventID, int nTimeout)	boolean send(CMEvent cme, String strReceiver)
CMEvent[] castrecv(CMEvent event, String strSession, String strGroup, int nWaitedEventType, int nWaitedEventID, int nMinNumWaitedEvents, int nTimeout)	boolean cast(CMEvent event, String strSession, String strGroup)

We particularly discuss the aforementioned CM services for the comparison of asynchronous and directly one-to-one synchronous methods, because this service sequence is commonly applied to every CM client.

B. QUALITATIVE COMPARISON OF SYNCHRONOUS AND ASYNCHRONOUS COMMUNICATION

All communication service of CM used in the request-reply pattern between client and server supports both synchronous and asynchronous APIs. Table 4 compares the synchronous APIs and the corresponding asynchronous APIs among the communication services.

The parameters of each API function are the same for both synchronous and asynchronous. The biggest difference between the two methods comes from the return value of the function. First, the return value of the asynchronous API is all boolean values. The boolean return value indicates whether the client CM has successfully sent the client's request to the server or not. If the return value is false, there is a problem with the request processing in the client CM. If the return value is true, the client CM has successfully forwarded the request to the server CM. However, the return value of the asynchronous API does not contain the server's response to the request. Instead, the client can receive the server's response event asynchronously through the event handler registered to CM. That is, when using the asynchronous API, it is necessary to separately check the desired response event among the CM events received in the event handler in order to confirm the response result of the server.

Unlike the asynchronous API, the return value of the synchronous API contains the server's response information to the service request. In addition, the format of the return value may vary depending on the type of service as follows.

Functions such as syncLoginCM (), syncRequestCM(), and syncJoinSession() can receive the server's response event for each request as a return value. The response event includes not only the result of the service request but also the additional information provided by the server. The return value of the syncAddBlockSocketChannel() function is a reference to the newly created blocking socket channel object. The client CM returns a locally generated socket channel only after the server also has successfully added the corresponding socket channel information. If the server side fails to add socket information, this function returns NULL indicating the request failure. This type of return value of synchronous API can be used for the client to obtain the service result directly. The return value of the syncRemoveBlockSocketChannel() function is of the boolean type, just like the asynchronous API. However, the meaning of the return value is different from the asynchronous API. If the return value is true, the corresponding channel on the server side as well as the client side has been successfully deleted. If the return value is false, it means that the deletion of the channel from the client side or the server side has failed. This type of return value of synchronous API can be used when the client needs to confirm whether the server has successfully completed the request or not.

Using synchronous APIs in application development allows for more intuitive development than the asynchronous APIs. This advantage is particularly evident when it is necessary to sequentially request a series of communication services. For example, in order to participate in a CM session, a client must first request the currently available session information from the server. In order to request session information, the client must first log into the server. When these three communication APIs are called synchronously, the client calls the syncLoginCM(), syncRequestSessionInfo(), and syncJoinSession() functions sequentially, by checking each return value. Of course, if any function call fails in the middle, the client stops the session participation procedure. Figure 13 shows the process of calling the synchronous APIs sequentially.

It is also possible to invoke the session participation procedure with the asynchronous APIs, but it is relatively less intuitive because the service request and the reception of server's response are separated into the client application and the event handler, respectively. First, the client calls the loginCM() function to send a login request. When we receive a login response event, we should indicate that the next task should be to request session information. One way is to declare a flag variable to set a specific value. The client's event handler must acknowledge the login response event asynchronously. If the flag variable is set to a specific value when receiving the login response event, the event handler calls the requestSessionInfo() function to request the session information and sets the flag variable to another value. If the event handler receives the available session information from the server and the flag variable is set, the event handler calls the joinSession() function to request the session participation.

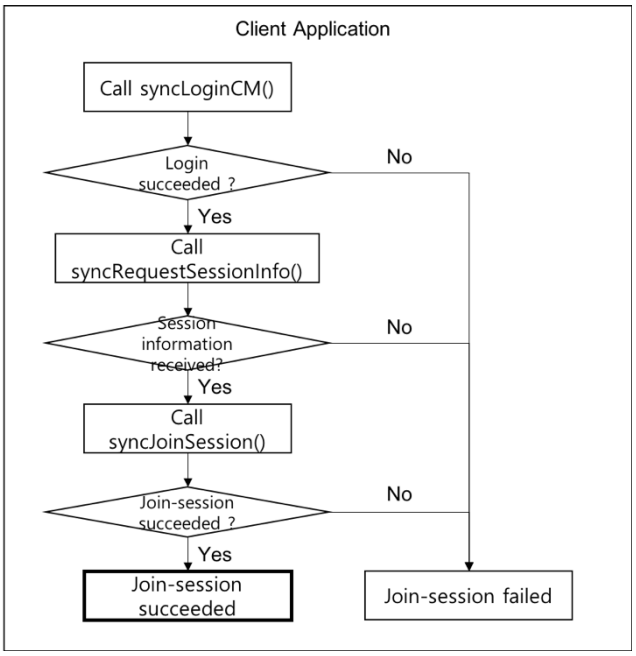


FIGURE 13. Flowchart of sequential request of synchronous APIs.

Figure 14 shows the process of calling the asynchronous APIs sequentially.

C. QUANTITATIVE COMPARISON OF SYNCHRONOUS AND ASYNCHRONOUS COMMUNICATION

Using the synchronous communication service of CM, the client application requests a service to the server and waits for a response through the corresponding function call and receiving the return value. Therefore, it becomes a means to develop applications more intuitively and efficiently as compared with the asynchronous communication service. However, after invoking the synchronous API, the client may experience a delay until it receives a return value. In this section, we quantitatively analyze the response delay time when the client requests the communication service of CM by the synchronous and asynchronous APIs.

The response delay time for a client's request is divided into two cases. The first is the return delay time of the communication API. The return delay time is measured as the elapsed time from when the client calls the API function to when the return value is returned. The second is the response delay time from the target server or client to the requesting client. The response delay is measured as the elapsed time from when the client calls the API function to when it receives the response from the target server or client. In the case of the synchronous API, the return delay time is the same as the response delay time because the client CM returns from the function that the client called only after receiving the response event. In the asynchronous API, the return delay time is the local processing time until the client CM sends a request event to the server. On the other hand, the response delay is the elapsed time from when the client calls the API to when its event handler receives the response event.

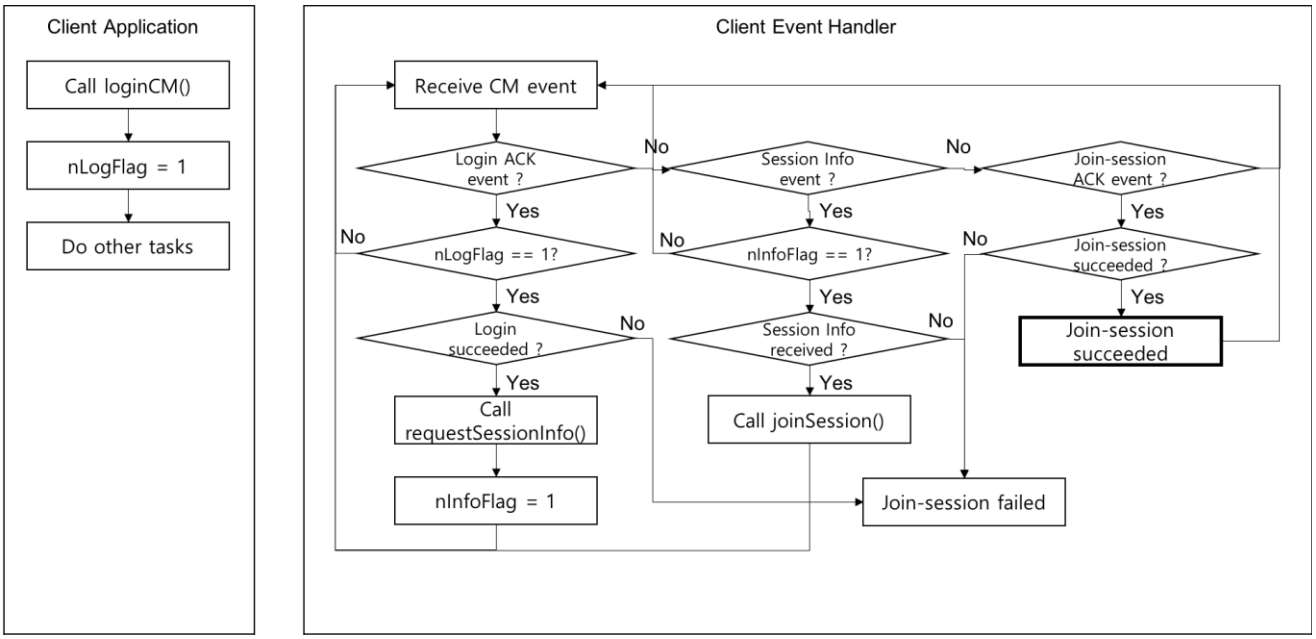


FIGURE 14. Flowchart of sequential request of asynchronous APIs.

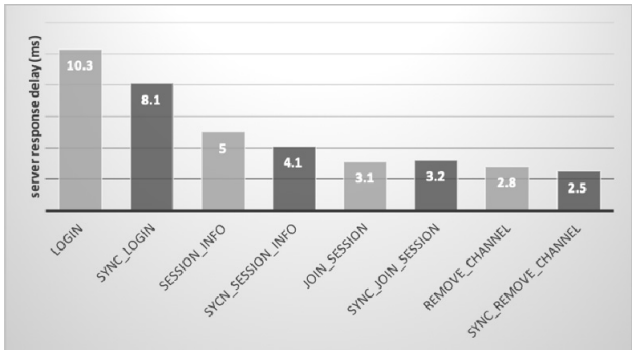
TABLE 5. Return delay time.

Asynchronous APIs	Return delay time (millisecond)
loginDS()	0.1
requestSessionInfo()	0
joinSession()	0
addBlockSocketChannel()	2.7
removeBlockSocketChannel()	0
Synchronous APIs	**Return delay time (millisecond)**
syncLoginDS()	8.1
syncRequestSessionInfo()	4.1
syncJoinSession()	3.2
syncAddBlockSocketChannel()	2381.6
syncRemoveBlockSocketChannel()	2.5

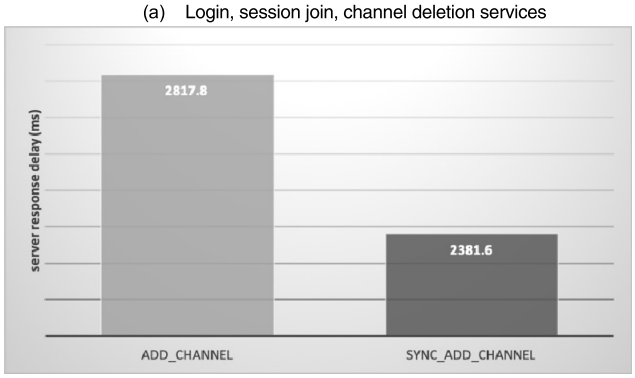
For the experiment, the computer running the server (Windows 10, i3 3.5GHz, 8GB memory) is connected with a 1Gbps wired LAN and the computer running the client (Mac OS, i5 3.8GHz, 16GB memory) is connected to the server via wireless LAN (WiFi 5GHz band). The target communication API for measuring the response delay time is the request for login, session information, session join, socket channel addition and deletion service in the directly synchronous communication, and the sendrecv() and castrecv() functions are used for the indirectly synchronous communication.

The return delay time of the CM service APIs was measured for the synchronous and asynchronous APIs, and the results are shown in Table 5. As can be seen from the measurement results, the asynchronous APIs show negligible return delay time in most function calls. Among the measurement functions, the adding-channel function has a slight delay time because it has a process of adding a communication channel locally. That is, if an asynchronous API is used, the client can immediately return a call to the API function and continue with the next execution. On the other hand, the synchronous APIs show relatively longer return delay time because they can return only after receiving the response events from the server. Like the asynchronous case, the synchronous version of the adding-channel function particularly has long delay time. This is because the connection establishment task with the remote server requires high cost. Therefore, the synchronous communication is not appropriate for the service that takes a long time to complete at the server.

We also measured and compared the response delay of both synchronous and asynchronous CM APIs. Figure 15 shows the result of server response time when the client requests asynchronous CM services and their corresponding directly synchronous services. The socket channel addition function has a higher response delay than the other APIs. This is because the process of creating a new channel and registering related information on the server side is a relatively expensive task. Overall response delay is similar for the rest of the APIs. The comparison of the server response delay of the synchronous APIs to that of the asynchronous APIs shows that the response delay of the synchronous APIs is relatively shorter than that of the asynchronous APIs. In the synchronous method, the server's processing thread receives the server's response event and delivers it to the synchronized main thread, which forwards the response event back to the client application. In the asynchronous method, the processing thread of the client CM receives the response event, and then forwards the response event to the client event handler by invoking the callback function that was registered to CM. That is, since the cost of returning a function after a synchronization process between threads is less than the cost of



(a) Login, session join, channel deletion services



(b) Channel addition service

FIGURE 15. Server response delay of asynchronous and directly synchronous APIs.

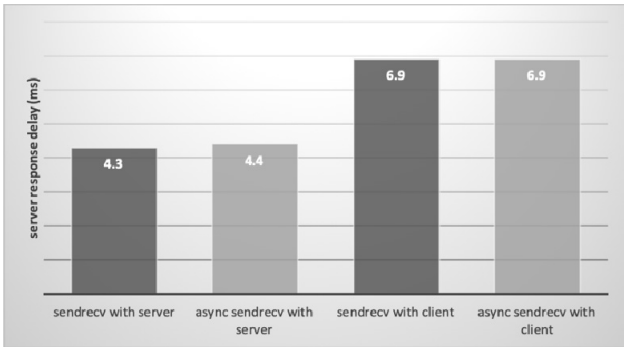


FIGURE 16. Response delay of sendrecv() function and its asynchronous version.

invoking a callback function for event delivery to an application under the assumption that the two cases have the same event transmission time, the synchronous communication has the slightly shorter response delay than the asynchronous communication.

Figure 16 compares the response time between the synchronous communication API, sendrecv() function, and its asynchronous implementation. The left two items of the graph are the result of the directly synchronous communication in which the target node receiving the request event is the server and the result of the asynchronous version thereof. The right two items of the graph are the indirectly synchronous communication method in which the target node is another client and its asynchronous version. In case of the indirect

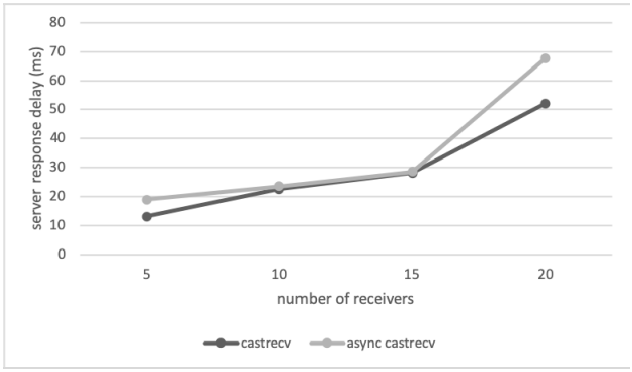


FIGURE 17. Response delay of castrecv() function and its asynchronous version.

communication, response delay is longer than the direct communication because two clients communicate with each other through the server. In both cases, the response delay in synchronous and asynchronous communication methods is almost the same because they use one-to-one communication between two nodes.

Figure 17 compares the response delay between the one-to-many indirectly synchronous communication API, the castrecv() function, and its asynchronous implementation. The number of target clients receiving the client's request event is equal to the minimum number of response events that the requesting client waits. We measured the response delay as the size of target group increases. As in previous experiments, we found that the response delay in the synchronous mode is shorter than that in the asynchronous mode. As the number of minimum response events to wait increases, the response delay time also increases, and the asynchronous method exhibits a relatively higher increase rate of the response delay than the synchronous method.

VI. RELATED WORK

This chapter describes the support for synchronous and asynchronous communication in existing research such as middleware or framework that provides communication services for applications. Relevant studies with a similar purpose to CM are largely targeted at communication middleware for general distributed applications or middleware specialized for specific applications such as SNS, network games, battlefield environment, mobile nodes, and electric vehicles.

Message Passing Interface (MPI) [10] is a message-passing standard among processes for parallel computing architectures. It defines a variety of low-level communication primitives. MPI supports the directly synchronous and the asynchronous communications, but it does not have the indirectly synchronous concept.

Reference [2] is the initial CM model and communication middleware that provides communication services to general distributed applications. This version of CM only supports event-based asynchronous communication. Reference [7] proposed MOCKETS, a message-centric middleware specialized in wireless Internet environments. In order to

compensate the disadvantage of TCP communication, MOCKETS provides continuous communication service even when the communication node moves based on UDP. However, since it focuses on providing socket level communication capability at the application end, it does not describe whether the application supports synchronous or asynchronous communication. Reference [13] is a research on communication middleware that provides communication services mainly to facilitate the development of heterogeneous distributed computing environment. It is described as supporting synchronous and asynchronous message delivery models to meet the needs of a variety of applications, but the details are not available. Reference [14] proposed an Internet Communication Engine as an object-oriented middleware that supports a lightweight communication service similar to CORBA. Because it provides services like CORBA, it is expected to support synchronous and asynchronous communication, but there is no specific explanation or analysis.

Reference [3] is an extended version of CM that adds communication services to facilitate SNS development in the initial CM. It focuses on providing SNS related services and still supports only event based asynchronous communication. Reference [12] has proposed a middleware focused on supporting SNS development on smartphones. It supports synchronous communication that responds to client requests through surrogate objects and is used for specific services such as constantly updating the current position of a friend via publish-subscribe based asynchronous communication between Surrogate objects. Reference [9] proposed a middleware service for pervasive social networking environments. With this middleware service, a user can easily find other users who are socially or physically associated and can share common interests with them. This work does not mention any support of synchronous or asynchronous communication. Reference [4] introduced SNS middleware called MobiClique, which can directly transfer SNS content between devices in an ad hoc peer-to-peer (P2P) environment, rather than a client-server environment. MobiClique provides services that asynchronously deliver SNS-related notification events such as connection status with friends, message reception, and so on. This work does not describe whether the synchronous communication method is supported or not. Reference [15] and [16] have proposed a middleware that provides a mobile SNS development platform called MobiSoC. It provides a centralized service that monitors, manages, and shares various social status of user groups such as profile, user location, and location information. When an application registers a social state event of interest in the middleware, the middleware that detects a change in the social state delivers the event asynchronously. The application also provides an API to call synchronously, but there is no description of the specific communication method.

Reference [11] proposed a message-driven middleware for network games. This research mainly focuses on user message interest (or filtering) management techniques to provide scalability by reducing the number of messages due to a large number of users. This middleware uses CORBA RPC [21] for synchronous communication and CORBA NS [22] for asynchronous communication but does not make a comparative analysis between them. Reference [8]'s work has proposed a communication framework based on a mobile agent, called FlexFeed, which is specialized in the battlefield environment. It focuses primarily on data streaming methods in heterogeneous systems with limited computing and communication resources. Reference [5] proposed a communication middleware named Scalable Distribution Layer (SDDL) to support real-time tracking of mobile nodes. As SDDL provides different communication modes such as unicast, groupcast, and broadcast based on the publish-subscribe model, it supports only the asynchronous communication. Reference [6] proposed a communication framework for notifying on-the-move electric vehicles of charging stations. As it provides push and pull communication modes based on the publish-subscribe model, it supports only the asynchronous communication.

In summary, existing communication middleware and framework support only the asynchronous method or do not clearly mention the support of synchronous and asynchronous communication methods. Even though some researches support both the synchronous and asynchronous methods, they can be used only in limited situations or it is not clearly described. Table 6 summarizes the analysis of communication methods of existing researches.

TABLE 6. Support of communication methods in existing researches.

References	Communication support
[2][3][4][5][6]	Only asynchronous communication
[7][8][9]	Not clearly described
[10]	Directly synchronous and asynchronous communication, but no support for indirectly synchronous communication
[11][12]	Directly synchronous and asynchronous communication for specific service types
[13][14][15][16]	Synchronous and asynchronous communication, but no details described

VII. CONCLUSIONS

In this paper, we proposed a synchronous communication service mechanism in CM, which is an event-based asynchronous communication framework. CM provides both synchronous and asynchronous communication services between a client and a server or multiple clients, thus supporting more flexible communication services according to application requirements. CM has applied a synchronization mechanism between internal threads to support one-to-one directly, one-to-one indirectly, and one-to-many indirectly synchronous communication schemes. Through the support of synchronous and asynchronous communication services of CM, the client can select not only the necessary services among the communication services of CM, but also

the communication method with the server according to the client's current context or requirements. In order to investigate the characteristics of synchronous and asynchronous communication methods, we conducted qualitative analysis and quantitative experiments. As a result of performance analysis, it is more efficient to call synchronous API than to invoke asynchronous API in order to improve programming efficiency and intuitiveness when the client needs to use a set of communication services sequentially. The response delay time of the synchronous APIs is smaller than that of the asynchronous APIs although the difference of the delay is not significant.

In future work, we will conduct practical case studies that can be applied to various types of synchronous communication methods to verify the effectiveness of synchronous communication methods more specifically. In addition, we plan to expand the request-reply pattern, which is a communication method of synchronous and asynchronous CMs, to support the publish-subscribe pattern communication service commonly used in Internet of Things (IoT).

REFERENCES

[1] A. S. Tanenbaum and M. van Steen, "Types of communication," in *Distributed Systems: Principles and Paradigms*, 2nd ed. New York, NJ, USA: Pearson Education, 2007, pp. 124–125.
[2] M. Lim, B. Kevelham, N. Nijdam, and N. Magnenat-Thalmann, "Rapid development of distributed applications using high-level communication support," *J. Netw. Comput. Appl.*, vol. 34, no. 1, pp. 172–182, Jan. 2011. doi: 10.1016/j.jnca.2010.08.003.
[3] M. Lim, "CMSNS: A communication middleware for social networking and networked multimedia systems," *Multimedia Tools Appl.*, vol. 76, no. 17, pp. 18119–18135, Jul. 2017. doi: 10.1007/s11042-016-3839-7.
[4] A.-K. Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, and C. Diot, "MobiClique: Middleware for mobile social networking," in *Proc. 2nd ACM Workshop Online Social Netw.*, Barcelona, Spain, Aug. 2009, pp. 49–54.
[5] L. David, R. Vasconcelos, L. Alves, R. André, G. Baptista, and M. Endler, "A communication middleware for scalable real-time mobile collaboration," in *Proc. IEEE 21st Int. Workshop Enabling Technol., Infrastruct. Collaborative Enterprises*, Hammamet, Tunisia, Jun. 2012, pp. 54–59.
[6] Y. Cao, N. Wang, G. Kamel, and Y.-J. Kim, "An electric vehicle charging management scheme based on publish/subscribe communication framework," *IEEE Syst. J.*, vol. 11, no. 3, pp. 1822–1835, Sep. 2017. doi: 10.1109/JSYST.2015.2449893.
[7] M. Tortonesi, N. Suri, C. Stefanelli, M. Arguedas, and M. Breedy, "MOCKETS: A novel message-oriented communications middleware for the wireless Internet," in *Proc. WINSYS*, Setubal, Portugal, Aug. 2006, pp. 258–267.
[8] M. Carvalho, N. Suri, and M. Arguedas, "A mobile agent-based communications middleware for data streaming in the battlefield," in *Proc. IEEE Mil. Commun. Conf.*, Atlantic, NJ, USA, Oct. 2005, pp. 794–800.
[9] S. B. Mokhtar, L. McNamara, and L. Capra, "A middleware service for pervasive social networking," in *Proc. Int. Workshop Middleware Pervasive Mobile Embedded Comput.*, Urbana, IL, USA, Nov. 2009, p. 2.
[10] *MPI: A Message-Passing Interface Standard*, document Version 3.1, 2015.
[11] G. Morgan, F. Lu, and K. Storey, "Interest management middleware for networked games," in *Proc. Symp. Interact. 3D Graph. Games*, Washington, DC, USA, Apr. 2005, pp. 57–64.
[12] D. Brooker, T. Carey, and I. Warren, "Middleware for social networking on mobile devices," in *Proc. 21st Austral. Softw. Eng. Conf.*, Auckland, New Zealand, Apr. 2010, pp. 202–211.
[13] D. Pakkala, P. Pakkonen, and M. Sihvonen, "A generic communication middleware architecture for distributed application and service messaging," in *Proc. Int. Conf. Autonomic Auton. Syst. Int. Conf. Netw. Services*, Papeete, Tahiti, French Polynesia, Oct. 2005, p. 22.
[14] M. Henning, "A new approach to object-oriented middleware," *IEEE Internet Comput.*, vol. 8, no. 1, pp. 66–75, Jan. 2004. doi: 10.1109/MIC.2004.1260706.
[15] A. Gupta, A. Kalra, D. Boston, and C. Borcea, "MobiSoC: A middleware for mobile social computing applications," *Mobile Netw. Appl.*, vol. 14, no. 1, pp. 35–52, Feb. 2009. doi: 10.1007/s11036-008-0114-9.
[16] C. Borcea, A. Gupta, A. Kalra, Q. Jones, and L. Iftode, "The MobiSoC middleware for mobile social computing: Challenges, design, and early experiences," in *Proc. 1st Int. Conf. MOBILe Wireless MiddleWARE, Operating Syst., Appl.*, Innsbruck, Austria, Feb. 2008, p. 27.
[17] M. Lim, "Supporting synchronous and asynchronous communications in event-based communication framework for client-server applications," *Int. J. Adv. Comp. Res.*, vol. 9, no. 40, pp. 11–19, Jan. 2019. doi: 10.19101/IJACR.COM16004.
[18] F. Cristian, "Synchronous and asynchronous," *Commun. ACM*, vol. 39, no. 4, pp. 88–97, Apr. 1996. doi: 10.1145/227210.227231.
[19] F. P. Lim, "An analysis of synchronous and asynchronous communication tools in e-learning," in *Proc. AST*, Hanoi, Vietnam, 2017, pp. 230–234.
[20] S. Tai and I. Rouvellou, "Strategies for integrating messaging and distributed object transactions," in *Proc. IFIP/ACM Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.*, pp. 308–330, 2000. doi: 10.1007/3-540-45559-0_16.
[21] L. M. Kei and X. Jia, "An efficient RPC scheme in mobile CORBA environment," in *Proc. Int. Workshop Parallel Process.*, Toronto, Ontario, Canada, Aug. 2000, pp. 575–582.
[22] R. E. Gruber, B. Krishnamurthy, and E. Panagos, "CORBA Notification Service: Design challenges and scalable solutions," in *Proc. 17th Int. Conf. Data Eng.*, Berlin, Germany, Apr. 2001, pp. 13–20.

MINGYU LIM received the B.S. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 1998, and the M.S. and Ph.D. degrees in computer science from Information and Communications University (ICU), Daejeon, South Korea, in 2000 and 2006, respectively.

He was a Senior Researcher with MIRALab, University of Geneva, Geneva, Switzerland, from 2006 to 2008. He was an Assistant and Associate Professor with the Department of Internet & Multimedia Engineering, Konkuk University, Seoul, South Korea, from 2009 to 2016. He is currently a Professor with the Department of Smart ICT Convergence, Konkuk University, Seoul. His current research interests include communication middleware and framework, efficient event transmissions, and content distribution in distributed systems.