Received May 8, 2020, accepted May 26, 2020, date of publication June 1, 2020, date of current version June 11, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2998725

C2CFTP: Direct and Indirect File Transfer Protocols Between Clients in Client-Server Architecture

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

This work was supported 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 an efficient direct and indirect file transfer protocol (C2CFTP) that transfers files between clients in a client-server system. Existing file transfer methods use an indirect transfer method through a server to transfer files between sending and receiving clients or a direct transfer method that connects a direct data channel between clients. However, in the case of indirect transmission, unnecessary file input/output (I/O) is required by the server, and in the case of direct transmission, a problem arises in that the file transmission delay time is increased due to channel management cost. The proposed C2CFTP can omit unnecessary file I/O overhead by relaying the file to the receiving client instead of storing the file at the server for indirect transmission. For direct transmission, instead of connecting a data channel every time a file is transmitted, the channel connection overhead is reduced while minimizing the waste of the number of direct channels between clients by maintaining the first connected channel while the file transfer is required within a predetermined time. File transfer experiments show that the proposed file transfer protocol has a reduced file transfer delay time than the existing methods. In addition, it was confirmed that the direct transfer method is suitable for transferring large files, and the indirect transfer method is suitable for transferring multiple small-sized files in a bundle.

INDEX TERMS File transfer protocol, client-server model, communication framework, indirect file transfer, channel management.

I. INTRODUCTION

The file transfer function is an important service commonly used in various contemporary distributed applications such as online games, online social networks, chat applications, cloud storage services, and Internet of Things(IoTs). The File Transfer Protocol (FTP) [1], [2] currently used as a standard and later improved file transfer protocols or systems [3]–[11] have been studied mainly focusing on methods for efficiently transferring large files between a client and a server. In addition, applications that share content between users, such as online social network systems, require a file transfer function, and communication frameworks or middlewares [12]–[18] that provide various communication functions including the file transfer as a service have also been studied. As described above, the existing file transfer function deals with a file transfer environment between directly connected clients and servers in a client-server structure or between directly connected peers in a peer-to-peer(P2P) structure.

However, in a distributed application of a client-server structure, it is also necessary and important to explicitly and implicitly transfer files between clients to share and update digital content in social and business life. For example, in a chat application, a user can send a photo file to a friend to share his/her experience, or a cloud storage service user can share his/her work file with a coworker to collaborate with each other. When it is necessary to transfer files between clients in this way, the existing methods do not suggest a separate file transfer method. That is, existing file transfer methods may use a two-stage transfer in which the transmitting client first transmits the file to the server and the server transmits it to the receiving client. However, this method has a problem in that file transfer performance is deteriorated by redundantly performing file I/O operations during the file

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

transfer process. That is, the server opens a new file to receive the file from the sending client, writes the received blocks to the file, and finally closes the file. The server opens the file again to send this file to the receiving client, reads the file blocks, sends it, and finally closes the file. As such, if the server repeatedly and unnecessarily opens / closes files and inputs / outputs files, the delay in transferring files between clients increases.

Many file transfer methods, including FTP [1], [2], connect a dedicated communication channel for file transfer between transmitting and receiving nodes. However, it does not set a dedicated channel directly between clients. Rather, FTP allows two FTP servers to transfer files directly over a channel connection at the client's request. In the peer-to-peer architecture like BitTorrent [5], the receiving node makes a direct connection with several sending nodes, but in this case, it is not a client-server structure. In addition, these dedicated channels are connected every time a file is transferred and then disconnected after completion. As suggested in our previous study [13], if the file transfer is repeated frequently, the delay time due to the repetition of the channel connection also increases, so a separate channel management method is needed. In the study of [13], a dedicated channel was connected between the client and the server in advance to solve the channel connection delay problem. However, this method is not suitable for file transfer between clients. This is because as the number of clients increases, the number of dedicated channels to be connected between all pairs of clients increases exponentially.

In this paper, we propose a protocol (C2CFTP) that can efficiently transfer files between clients in a client-server architecture by removing the unnecessary file I/O at the server and reducing the overhead of dedicated channel management so that we can reduce the file transfer delay according to different file-transfer requirements of applications. The proposed C2CFTP is an improvement and extension of the previous file transfer method of the communication framework (CM) [13] that supported only file transfer between the client and the server. C2CFTP is composed of two types: direct transfer protocol (direct C2CFTP) and indirect transfer protocol (indirect C2CFTP). The two file transfer services can be used selectively depending on the needs of the application. The direct C2CFTP directly transfers files using a separate data channel and a dedicated thread between the transmitting and receiving clients. The dedicated channel is not released immediately after the file transfer is completed, but is maintained for a keep-alive time. Therefore, if two clients frequently send files, there is no need to reset the channel every time. If there is no file transfer task for a specified threshold period, the channel is considered unnecessary and then it is deleted. The indirect C2CFTP does not use a separate dedicated channel and thread, but transmits the file indirectly through the relay of the server using the default channel connected to the server. Since this default channel is used only for CM event transmission, we add a file block to the CM file event to transmit the file block. The server only serves to deliver file events to the receiving client, and does not perform unnecessary file I/O operations.

As a result of performance analysis through experiments, we found that the proposed C2CFTP reduced the average file transfer delay time compared to the existing file transfer methods. In the experiment transferring one file, the direct C2CFTP showed less latency than the indirect C2CFTP. Instead, in the case of sending a bundle of files, the direct C2CFTP increased overhead because of the delay of managing threads for files. Thus, the indirect C2CFTP showed relatively better performance. In the experiments of transferring a bundle of files, the indirect C2CFTP exhibits a low transmission delay when the file is a small size of less than 1 MB. However, as the file size increases, the direct C2CFTP shows better performance because the effect of thread management overhead of a file bundle becomes smaller. Therefore, an advantage of C2CFTP is that an application can select one of the direct and the indirect C2CFTP methods according to its file transfer requirement.

II. COMMUNICATION FRAMEWORK (CM)

CM [12], [13] is a communication framework for developing distributed applications. By using the CM Application Programming Interfaces (APIs), application developers can easily implement various communication functions that allow applications to interact with other CM nodes (applications using CM's communication service). CM nodes can divide roles into a client and a server. The CM client node can simply use basic communication functions such as registration, connection, login, CM event creation and transmission to the server. The client can also receive and handle CM events from the CM server node. In addition to the basic communication functions, CM nodes can use other services such as file transfer, content management of social networking service (SNS), network status check, and communication channel management.

CM nodes communicate internally in an event-based asynchronous manner. To support the asynchronous communication method, CM runs in multiple threads (main thread, processing thread, sending thread, and receiving thread) as shown in Figure 1. When the main thread of the application runs the CM, the CM 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. The sending thread is responsible for converting CM events created by the main thread or processing thread into low-level byte messages and sending them to the remote target node. When the main thread or the processing thread needs to send a message to a remote node, it creates the necessary CM event, puts it in the sending queue, and delivers it to the sending thread. The receiving thread is responsible for receiving the byte message from the network, converting it into a CM event, and passing it to the processing thread. The receiving thread

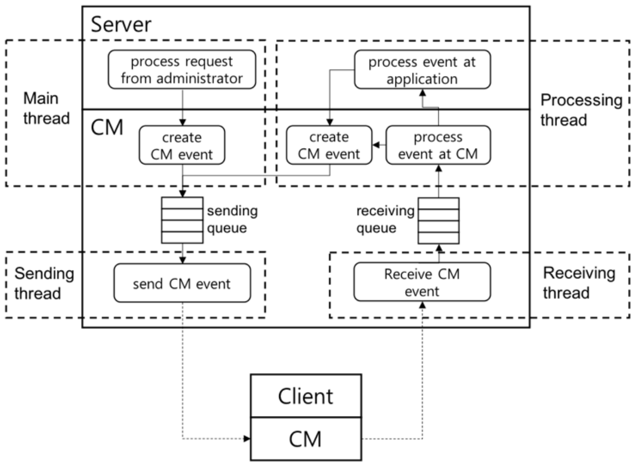


FIGURE 1. CM multi-thread structure.

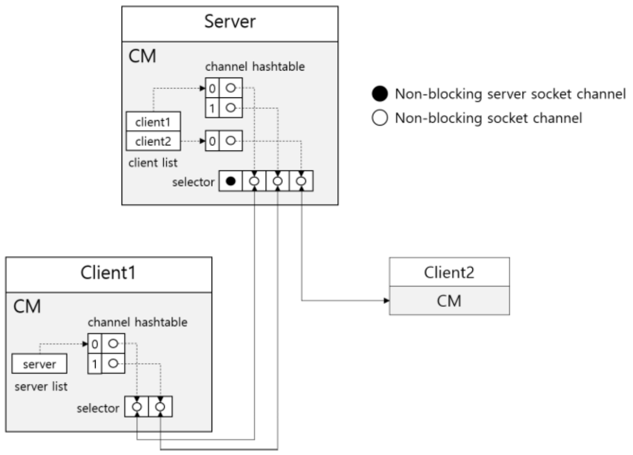


FIGURE 2. CM channel structure in a client and a server.

puts the received CM event into the receiving queue, and the processing thread receives and processes it.

A. CM CHANNEL MANAGEMENT

CM is basically designed to send and receive CM events asynchronously by registering a non-blocking channel in a selector object. The receiving thread of the CM monitors the selector and detects new connection requests coming into the server socket channel or CM events coming into other channels. There are various communication channels that CM handles, such as general socket channels, datagram channels, and multicast channels, but in this paper, we only deal with socket channels necessary for file transfer. CM contains information of other communicating CM nodes (clients or servers). The client CM has the information of the connected server, and the server CM has the logged-in client information. Since one CM node can connect multiple channels with other CM nodes, the channels are managed in the hash table for each connected target node. The ID of the target node becomes the channel name, and the key of the hash table becomes the channel number. The channel number is an integer greater than or equal to 0. The client CM maintains a hash table that stores non-blocking channels connected to the server. At this time, the name of the basic channel connected to the server is the server ID, and the channel number is 0. If the client wants to create another non-blocking channel with the server, a channel number different from the existing number must be assigned and the channel is stored in the hash table using this number as a key. The server CM has a hash table that stores non-blocking channels for a client connected to it. When the server CM searches for a basic non-blocking channel connected to a client, the channel name is the client ID and the channel number is 0. Figure 2 shows the channel structure of client and server in CM. In the figure, the server has the information of two logged in clients (client1, client2) and is connected to them by a non-blocking channel. Client1 is connected to two channels (channel number 0, 1), and client2 is connected to one channel (channel number 0). In addition, the server also has a server socket channel to handle the client's connection request to the selector. Client1 has server information and a channel hash table in it. Client2 is connected to the server by one basic non-blocking channel, and the channel structure diagram of Client2 is omitted in the figure. As such, CM separately manages the channel information of the connected target node with the pair of a channel name and a channel number, and when the CM's upper manager module needs to refer to the channel, it can search for the channel with the pair of information.

B. CM FILE TRANSFER METHOD

The previous file transfer service of CM [13] has two methods: using the default non-blocking communication channel, and applying a separate dedicated blocking communication channel and a thread. The application can select and use the appropriate method from the two methods depending on its requirements. In the file transmission method based on the non-blocking channel (hereinafter referred to as non-blocking file transfer), a file block is transmitted using the default non-blocking channel used by CM for event transmission. Because this method does not require additional channels and threads for file transfer, there is no additional channel management cost. However, there is a burden of converting all transmitted file blocks into CM file events. In the file transfer method based on the blocking channel (hereinafter referred to as blocking file transfer), a dedicated channel and thread for file transfer are used. In this case, CM uses the non-blocking channel only for the file transfer preparation process between the transmitting node and the receiving node, and transmits file blocks using a dedicated blocking channel and a thread separately. To this end, CM manages hash tables for blocking channels separate from the hash tables for non-blocking channels. This method does not affect the default non-blocking channel of CM, but has the disadvantage of adding overhead for managing additional channels and threads.

The limitation of the previous CM is that both file transfer methods only support file transfer between the client and the server. In order for a client to send a file to another client, a two-stage file transfer is necessary in which the sending client first sends the file to the server, and then the server sends it back to the receiving client. This method has a problem in that the server executes unnecessary file I/O, which increase the transmission delay time. To solve the problem, the proposed method for transferring files between clients is described in detail in the next section.

III. CLIENT-TO-CLIENT FILE TRANSFER PROTOCOL (C2CFTP)

When a file transfer is required in the client-server structure, it can be divided into two types: client-server transfer and client-client transfer. In fact, using the client-server file transfer function, the client-client file transfer can be implemented in two simple steps. First, the sending client sends the file to the server. After the transfer is completed, the server sends this file back to the receiving client. However, repeating the same transmission process twice causes unnecessary redundancy in the detailed steps of the protocol. For example, in the process of transmitting a file from the sending client to the server, the server opens a new file, writes the received file blocks to this file, and when the reception is completed, closes the file to finish the transmission process. The server then opens the file again to send this file directly to the receiving client, and this time it needs to read the file blocks. In this way, the server unnecessarily repeats the opening/closing of the same file and reading/writing the same file blocks. It is well known that file I/O repetition is one of the main factors affecting system performance. As another example, even if a separate data channel is used for file transfer, dedicated data channels must be created between the sending/receiving clients and the server every time the file is transferred. This is an additional cost to increase file transfer time. Since this cost is added for each file transfer session, the problem is exacerbated because the file transfer delay time is continuously cumulated, especially in environments where clients frequently send and receive small sized files repeatedly.

The existing CM file transfer service has the same problem as the existing file transfer method, because it only supported file transfer between the client and the server. The client-client file transfer method proposed in this paper is a protocol that eliminates unnecessary extra work due to the previously mentioned two-step file transfer. In the proposed method, the server is only responsible for relaying the file block transfer between two clients, instead of temporarily storing the file in the middle. Since the existing CM file transfer service provides two transmission methods: the blocking and the non-blocking file transfer, this paper extends each method to improve the protocol (which is called C2CFTP) applicable to the client-client file transmission. The improvement of the existing blocking file transfer method is called direct C2CFTP, and the improvement of the existing non-blocking file transfer method is called indirect C2CFTP.

As shown in Figure 3, in the direct C2CFTP, a dedicated blocking channel is created between a sending and a receiving

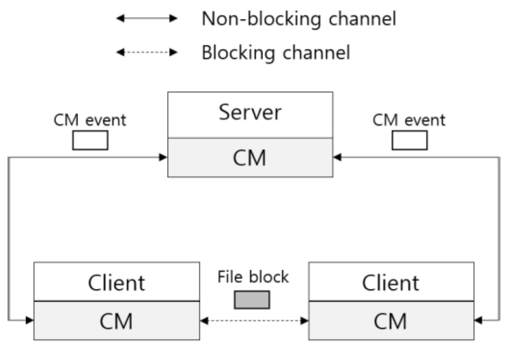


FIGURE 3. Direct C2CFTP concept.

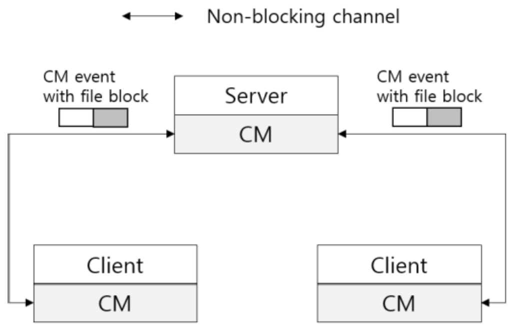


FIGURE 4. Indirect C2CFTP concept.

client to transmit file blocks through this channel. As shown in Figure 4, in the indirect C2CFTP, the client sends and receives file blocks using the default non-blocking channel connected to the server instead of a dedicated channel. That is, the sending client transmits each file block to the server, and the server delivers the block back to the receiving client. This method is similar to the previous two-step file transfer, which was raised as a problem, but the server does not perform file I/O because the file blocks received by the server is transmitted directly to the receiving client rather than stored as a separate file. Both the direct and indirect C2CFTPs send and receive control CM events for synchronization of the sending and receiving clients before and after the file transfer task. Since the clients are not directly connected to each other through a non-blocking channel for CM event transmission, all of these CM file events are transmitted indirectly through the relay of the server [19]. The client uses one of the direct or indirect C2CFTP, which is decided by the server. The server can decide policies of various CM communication services with a CM configuration file that is provided together with the CM library. The file transfer policy can be set in the FILE_TRANSFER_SCHEME field. That is, if the field value is 1, the direct C2CFTP is used, and if 0, the indirect C2CFTP is used. The server decision of the file transfer method is just a matter of policy for keeping a consistent method in an application. Thus, if needed, the client also can easily call the different method instead of following the server decision.

In addition, C2CFTP added a file transfer permission process that was not provided by CM's existing file transfer method. In the existing CM file transfer, if the application calls the CM's file transfer service method, the file transfer proceeds unconditionally regardless of the consent of the other application. C2CFTP added the automatic file transfer permission option to the CM configuration file, so that when an application uses CM, it can decide whether to automatically allow a file transfer request from another application. If the value of the PERMIT_FILE_TRANSFER field of the CM configuration file is set to 1, the request for the file transfer of the other party is allowed unconditionally. On the other hand, if the field value is set to 0, the application must decide whether to allow the file transfer request. The application can inform the CM whether or not to accept the request by calling the replyEvent() method added to the stub module.

A. DIRECT C2CFTP

The direct C2CFTP is used to transfer file blocks by connecting a dedicated blocking channel between the file transmitting node and the receiving node, similar to the existing blocking file transfer method of CM. To this end, the server CM sets the FILE_TRANSFER_SCHEME field value of the configuration file to 1. In the existing file transfer between the client and the server, the client always requested to create a blocking channel to the server because the server already has a server socket channel. However, in direct file transfer between clients, as the file sender and receiver are both clients, we must decide which client first creates the server socket channel. This decision does not significantly affect the direct file transfer protocol. In the proposed protocol, the receiving client creates a server socket channel, and the sending client connects a blocking channel to it. The server plays a role of relaying control CM events necessary for file transfer between the sending and the receiving clients. Figure 5 shows the internal structure of CM when using the direct C2CFTP. The detailed process of the direct C2CFTP is described in detail in the next section.

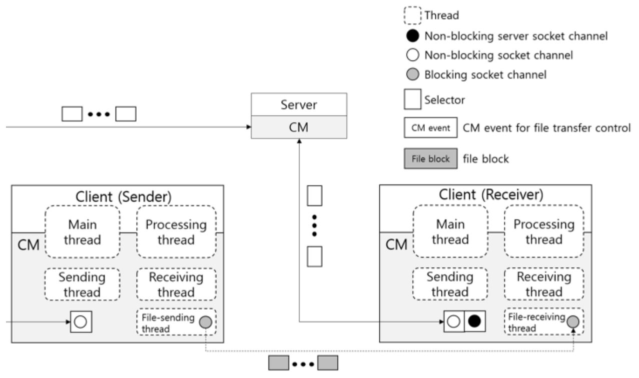


FIGURE 5. Direct C2CFTP structure.

1) FILE PUSH

Table 1 summarizes the CM control events exchanged between the sending and receiving clients at each stage of the file transmission service using the direct C2CFTP and their respective roles. The detailed format of each control event is described in Appendix A. In the table, "sender" is the sending client and "receiver" is the receiving client. Although the server is not shown in the table, all CM events are delivered to the target client via the server. The detailed procedure for the sending client to send the file to the receiving client in the direct C2CFTP is as follows.

TABLE 1. CM events for file-push service in direct C2CFTP.

Step	*Event direction*	*Event ID*
(1)	sender→receiver	REQUEST_PERMIT_PUSH_FILE
(2)	receiver→sender	REPLY_PERMIT_PUSH_FILE
(3)	sender→receiver	START_FILE_TRANSFER_CHAN
(4)	receiver→sender	START_FILE_TRANSFER_CHAN_ACK
(5)	sender→receiver	No CM event
(6)	sender→receiver	END_FILE_TRANSFER_CHAN
(7)	receiver→sender	END_FILE_TRANSFER_CHAN_ACK

(1) The sending client calls the pushFile() method of the CM stub module. The CM main thread sends a file transmission permission request event (REQUEST_PERMIT_PUSH_FILE) to the receiving client through the sending thread. In the receiving client, the receiving thread dispatches the request event to the processing thread, and the processing thread processes the request. If the request from the sending client is allowed, the return code of the response event (REPLY_PERMIT_PUSH_FILE) is set to 1, and if the request is rejected, the return code is set to 0. If both the sender and the receiver are clients, the receiving client creates a server socket channel to wait for the creation of a dedicated channel for data transmission with the sending client. The port number of this server socket channel is randomly assigned so as not to conflict with other port numbers, and this port number is set as the value of the port number field of the response event.

(2) The receiving client sends the response event (REPLY_PERMIT_PUSH_FILE) set in (1) to the sending client. When the sending client refers to the return code of the response event and the value is 0, the receiving client has rejected the request, and the file transmission process ends here. If the return code is 1, the sending client adds new file information to the outgoing file list and continues the file transmission procedure. If both the sender and the receiver are clients, the sending client stores the port number of the server socket in the receiver information. Since the CM client maintains information of other clients belonging to the same group, the sending client has the receiving client information. When a blocking channel connected to the receiving client exists, the transmitting client sets the channel information in the transmission file information and proceeds to the next step (3). If there is no blocking channel, the sending client

first connects the blocking channel with the receiving client address and port number, sets this channel in the transmission file information, and then proceeds to the next step.

(3) The sending client sends a transfer start event (START_FILE_TRANSFER_CHAN) to the receiving client. The receiving client sets a directory to store the received file and searches for a blocking channel to the sending client. The receiving client adds the blocking channel and file information to the incoming file list. When the CM receiving thread starts the file receiving thread, the file receiving thread waits to read file blocks being sent to the blocking channel.

(4) After starting the file receiving thread, the receiving client sends a response event (START_FILE_TRANSFER_CHAN_ACK) to the sending client. The sending client also starts a file sending thread, which starts sending file blocks over the blocking channel.

(5) The file sending thread transmits the file blocks to the end through the blocking channel, and the file receiving thread reads the file blocks from the blocking channel and writes them to the received file.

(6) When the file sending thread transmits all file blocks, it sends a completion event (END_FILE_TRANSFER_CHAN) to the receiving client and ends. The file sending thread directly transmitted the file block to the blocking channel, but since the transmission completion event is a CM control event, it is transmitted to the receiving client through the server with the non-blocking channel. When the receiving client receives the transmission completion event, it waits until its file receiving thread receives all file blocks and terminates.

(7) When the file receiving thread completes the task, the receiving client sends a response event (END_FILE_TRANSFER_CHAN_ACK) to the sending client, and deletes the file information. When the sending client also receives the response event, the corresponding file information is deleted. At this point, the receiving client searches for another incoming file information that the sending client has not yet started transmitting. When such file information is retrieved, it means that there are more files to be sent by the sending client, and the receiving client returns to step (4) to continue the next file receiving operation. When there is no more file information to be received from the sending client, the file push service ends.

2) BLOCKING CHANNEL MANAGEMENT

In the direct C2CFTP, the sending client and the receiving client use a blocking channel only for file block transfer. In the existing CM, the blocking file transfer method used a blocking channel between the client and the server, but did not use any direct connection between the clients. In the direct C2CFTP, in order to create a blocking channel between transmitting and receiving clients, the file receiving client first creates a server socket channel. This server socket channel is a non-blocking channel that is registered with the selector and waits for connection requests from other sending clients. When the sending client establishes a connection with the receiving client, the newly created channel in the receiving client is registered with the selector as a non-blocking channel. When the sending client sends a channel add event (ADD_BLOCK_SOCKET_CHANNEL) through the new channel, the receiving client's processing thread receives and processes this request event. That is, after unregistering the new non-blocking channel from the selector and changing to the blocking mode, it is added to the blocking channel hash table of the sending client. At this time, the channel name is the sending client ID and the channel number is 0. The receiving client sends the channel registration result as a response event to the sending client. The response event (ADD_BLOCK_SOCKET_CHANNEL_ACK) is delivered to the sending client through the server with the default non-blocking channel. When the sending client confirms that the receiving client has successfully added the blocking channel, it continues the file sending procedure. Figure 6 shows an example of the channel structure of the client and server in the direct C2CFTP. Client1 is a sending client and Client2 is a receiving client.

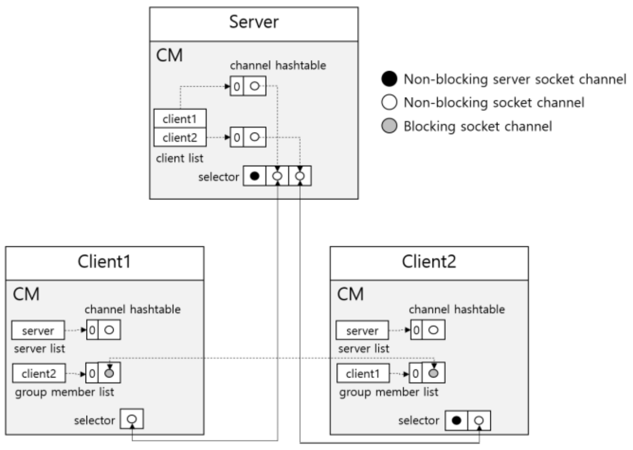


FIGURE 6. Channel structure of direct C2CFTP.

One of the considerations in the direct C2CFTP is to decide when to create and delete blocking channels between clients. If a channel is created whenever a file transfer is necessary, the creation time of each channel affects the file transfer delay time. The channel creation time varies greatly depending on the situation of the network and host at that time. Figure 7 shows the change in the total file transfer time when a blocking channel is created for each file transfer and a small file transfer of 1 KB in size is repeated. Therefore, if the application uses the file transfer function quite frequently, creating and deleting channels whenever necessary can be a significant performance burden. On the other hand, creating a blocking channel among all clients in advance causes a waste of unnecessary channel resources. Figure 8 compares the number of blocking channels between a server and a client

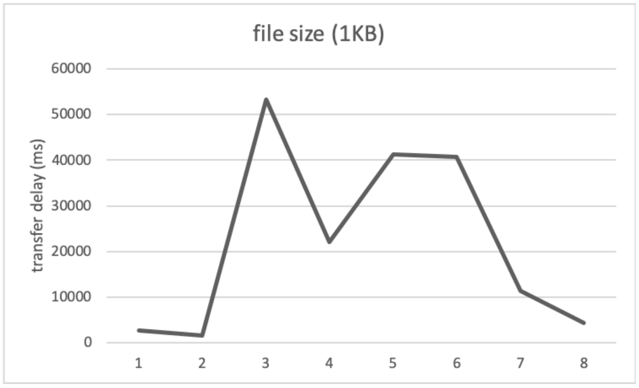


FIGURE 7. Change of file transfer delay including channel creation time.

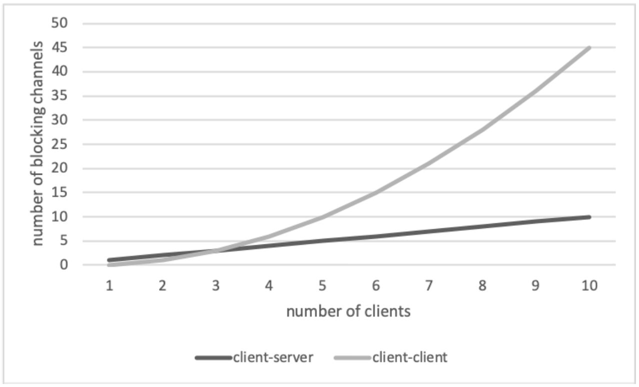


FIGURE 8. Number of blocking channels between all pairs of a server and clients vs. number of blocking channels between all pairs of clients.

and the number of blocking channels between two clients as the number of clients increases. In the former case, a client has only one blocking channel with the server regardless of the number of clients. In the latter case, however, a client should have blocking channels with all the other clients. Therefore, as the number of clients increases, so does the number of blocking channels. In this paper, the above two extreme methods are mixed to manage the blocking channel between clients. First, a blocking channel is created when the first file transfer is needed between two clients. This channel remains for a while even after the file transfer is complete. If this channel is no longer used for file transfer during the threshold time that can be set in the CM configuration file, CM deletes this channel.

The procedure for deleting blocking channel information between clients is as follows. When the receiving client confirms that its blocking channel has not been used for more than the threshold period, it sends a channel-deletion request event (REMOVE_BLOCK_SOCKET_CHANNEL) to the sending client. When the sending client receives this delete request event, it closes the blocking channel in the receiving client information and deletes it from the hash table. The sending client sends a response event (REMOVE_BLOCK_SOCKET_CHANNEL_ACK) including the deletion result to the receiving client. The receiving client also deletes the blocking channel information, and finally the server socket channel is also closed and deleted at the selector. CM event formats related to blocking channel addition and deletion are described in detail in Appendix B.

3) FILE PULL

While the file push is a service in which a file sending node requests file transfer, the file pull is a service in which a file receiving node requests file transfer. Table 2 shows CM control events exchanged between a receiving and a sending client when using a file receiving service with the direct C2CFTP. The detailed format of each control event is described in Appendix A. As shown in the table, the remaining steps (3) to (7) except the first two steps are the same as for the file push service. The detailed procedure of the first two steps is as follows.

TABLE 2. CM events for file-pull service in direct C2CFTP.

Step	*Event direction*	*Event ID*
(1)	receiver→sender	REQUEST_PERMIT_PULL_FILE
(2)	sender→receiver	REPLY_PERMIT_PULL_FILE
(3)~(7)	same as the file-push task in direct C2CFTP	

(1) The receiving client calls the requestFile() method of the CM stub module. If both the sender and receiver are clients, the CM main thread of the receiving client first creates a server socket channel to receive file blocks. The port number of the server socket channel is set as a field in the request event (REQUEST_PERMIT_PULL_FILE) and transmitted to the sending client.

(2) Upon receiving the request event, the sending client checks whether the requested file exists and whether the request is permitted. The sending client sets the return code field of the response event (REPLY_PERMIT_PULL_FILE) according to the permission of the request and transmits it to the receiving client. The return code is set to 1 for accepting the request, 0 for rejecting, or -1 if no such requested file exists. If the request is granted, the sending client adds the port number to the receiving client information. If there is no blocking channel to the receiving client, the sending client first completes the blocking channel connection and then performs the next step. If the receiving client receives the response event (REPLY_PERMIT_PULL_FILE) and the return code is 0 or -1 (i.e., the request is not allowed), the server socket channel is closed and deleted, and the file pull service is aborted.

The steps (3) to (7) are the same as for the file push service.

B. INDIRECT C2CFTP

To use the indirect C2CFTP, the server CM sets the FILE_TRANSFER_SCHEME field value of the configuration file to 0. The indirect C2CFTP uses the default non-blocking channel connected to the server for file block transfer, similar to CM's previous non-blocking file transfer method. Since all non-blocking channels managed by CM are used for CM event transmission, a CM event including file blocks are required. Among the file events in Appendix A, the CONTINUE_FILE_TRANSFER event is used for this purpose. In addition, the server has been improved so that it can only play the role of relaying file events in client-client file transfer. To this end, the file sender and file receiver fields are included in all file events, and when the file event is received, the server first checks whether it is a file sender or a receiver. If the server itself is not both the sender and receiver, the server determines that this event is for client-client file transfer and the server CM delivers it to the file sender or receiver client using the CM's internal forwarding technology [19]. Figure 9 shows the internal structure of CM when using the indirect C2CFTP. The detailed process of the indirect C2CFTP is described in the next section.

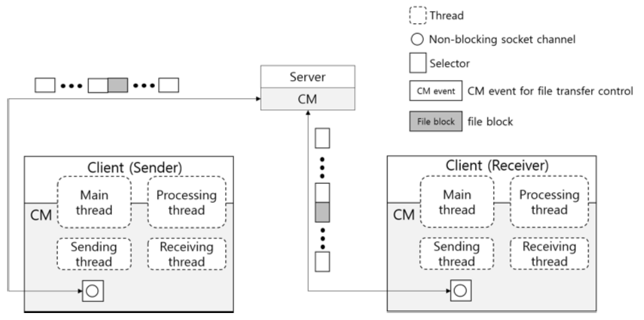


FIGURE 9. **Indirect C2CFTP structure.**

TABLE 3. **CM Events for file-push service in indirect C2CFTP.**

Step	*Event direction*	*Event ID*
(1)	sender→receiver	REQUEST_PERMIT_PUSH_FILE
(2)	receiver→sender	REPLY_PERMIT_PUSH_FILE
(3)	sender→receiver	START_FILE_TRANSFER
(4)	receiver→sender	START_FILE_TRANSFER_ACK
(5)	sender→receiver	CONTINUE_FILE_TRANSFER
(6)	sender→receiver	END_FILE_TRANSFER
(7)	receiver→sender	END_FILE_TRANSFER_ACK

1) FILE PUSH

Table 3 summarizes CM control events used in the indirect C2CFTP and their respective roles. The detailed format of each control event is described in Appenddix A. The main difference from the direct C2CFTP is that when transferring a file block in step (5), it uses the default non-blocking channel with the server, not a dedicated blocking channel. That is, both control CM events and file blocks are sent between the two clients via the server. The detailed procedure is as follows.

(1) The sending client calls the pushFile() method of the CM stub module. The CM main thread sends a transfer-permission request event (REQUEST_PERMIT_PUSH_FILE) to the receiving client through the sending thread. The receiving client puts the return code value to a response event (REPLY_PERMIT_PUSH_FILE) depending on whether the request is allowed or not, and sends it to the sending client.

(2) The sending client receives the response event from the receiving client, and if the return code of the event is 0 (rejected), the file transfer process is aborted. If the return code is 1 (permit), the sending client adds new file information to the outgoing file list and continues to the next step.

(3) The sending client sends a start event (START_FILE_TRANSFER) to the receiving client. The receiving client sets a directory to store the received file, opens a new file to receive, and adds newly opened file information to the incoming file list.

(4) The receiving client sends a response event (START_FILE_TRANSFER_ACK) to the file-sending client after the file reception preparation is completed.

(5) When the sending client receives the response event (START_FILE_TRANSFER_ACK), the file to be transmitted is opened and each file block is included in the block event (CONTINUE_FILE_TRANSFER) and transmitted to the receiving client. Whenever the receiving client receives a file block event (CONTINUE_FILE_TRANSFER), it writes the file block to the received file.

(6) The sending client closes the file after sending the last file block event and sends the transfer-completion event (END_FILE_TRANSFER). When the receiving client receives this event, it closes the received file and deletes the file information from the incoming file list.

(7) After the file receiving operation is completed, the receiving client sends a response event (END_FILE_TRANSFER_ACK) to the sending client. When the sending client receives this event, it deletes the file information from the outgoing file list and completes the file push task.

TABLE 4. **CM events for file-pull service in indirect C2CFTP.**

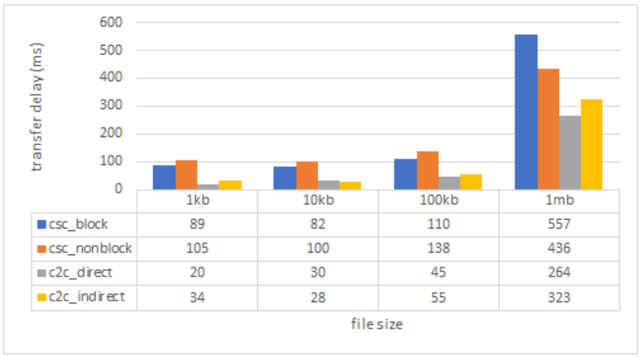
Step	*Event direction*	*Event ID*
(1)	receiver→sender	REQUEST_PERMIT_PULL_FILE
(2)	sender→receiver	REPLY_PERMIT_PULL_FILE
(3)~(7)	same as the file-push task in indirect C2CFTP	

2) FILE PULL

Table 4 shows CM control events exchanged between the receiving and sending clients when using the file receiving service with the indirect C2CFTP. The first two events are the same as those of the file-pull service of the direct C2CFTP. The processing part of these two events is the same as the processing in the direct C2CFTP, except that there is no blocking channel setup process between the sending and receiving clients. The rest of the steps (3) to (7) are the same as those in the file-push service of the indirect C2CFTP.

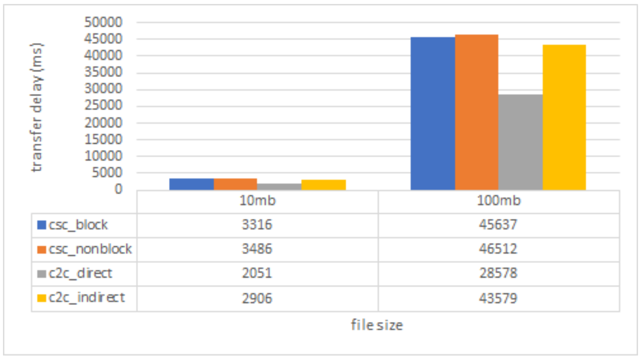
IV. PERFORMANCE EVALUATIONS

In order to compare and analyze the performance of the direct and indirect C2CFTPs proposed in this paper, we measured the average file transfer time for the proposed and existing methods. The existing method is the two-step file transfer between clients through the server. In the experiment, the server machine (Windows 10, Intel Core i3 3.5GHz CPU, 8GB RAM) connected to a 1Gbps WAN, and two clients (MacOS Catalina, Intel Core i5 3.8GHz CPU, 16GB RAM) connected to the server via Wi-Fi (2.4GHz band) of a wired/wireless router. The clients are in charge of sending and receiving files, respectively, and the sending client measured elapsed time until the file transfer is completed by calling the CM's file-push service. In addition, in the experiment, we also compared the change of the transmission time according to the different file size and the number of files. All measured file transfer times are obtained by transferring the files 100 times to obtain the average value.



	1kb	10kb	100kb	1mb
■ csc_block	89	82	110	557
■ csc_nonblock	105	100	138	436
■ c2c_direct	20	30	45	264
■ c2c_indirect	34	28	55	323

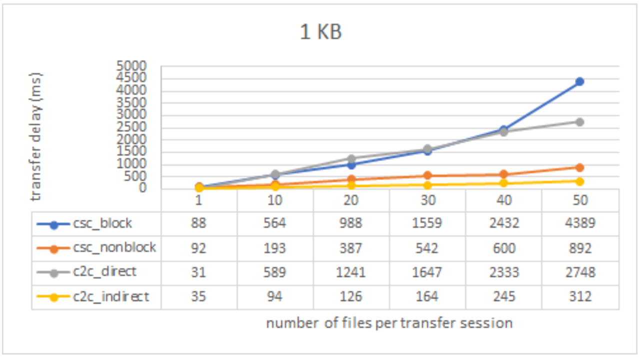
FIGURE 10. Transfer delay according to file size (1KB ∼ 1MB).



	10mb	100mb
■ csc_block	3316	45637
■ csc_nonblock	3486	46512
■ c2c_direct	2051	28578
■ c2c_indirect	2906	43579

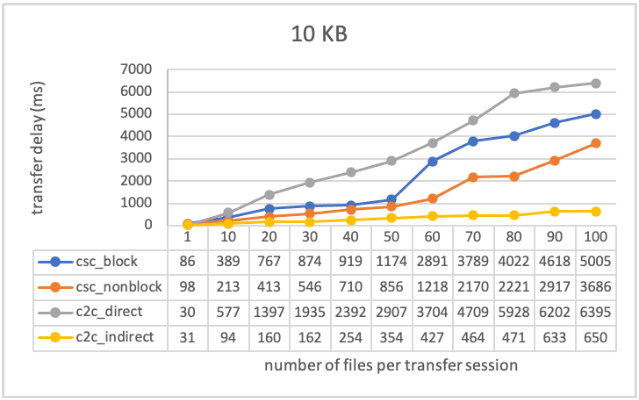
FIGURE 11. Transfer delay according to file size (10MB ∼ 100MB).

Figures 10 and 11 show the results of measuring the file transfer time for each method according to the file size. In the conventional method (csc_block, csc_nonblock), the sending client first performs file transfer to the server, and the server transmits the received file back to the receiving client. The existing method is further divided into the blocking transfer (csc_block) and the non-blocking transfer (csc_nonblock). As a result of the experiment, the proposed method (c2c_direct, c2c_indirect) has shorter file transfer time than the existing method regardless of the file size. Specifically, the direct C2CFTP (c2c_direct) reduced the transmission time by an average of 54.7% compared to the existing blocking transfer method (csc_block), and the indirect C2CFTP (c2c_indirect) decreased by an average of 41.4% compared to the existing non-blocking transfer method (csc_nonblock). This is because the proposed method omits redundant and unnecessary file I/O of the server. Among the proposed file transfer methods, the average transfer time of the direct C2CFTP is reduced by 22.4% compared to the indirect C2CFTP. In the case of transferring one file, the direct C2CFTP that uses the dedicated channel and thread is better than the indirect C2CFTP that uses the relay of the server.



	1	10	20	30	40	50
csc_block	88	564	988	1559	2432	4389
csc_nonblock	92	193	387	542	600	892
c2c_direct	31	589	1241	1647	2333	2748
c2c_indirect	35	94	126	164	245	312

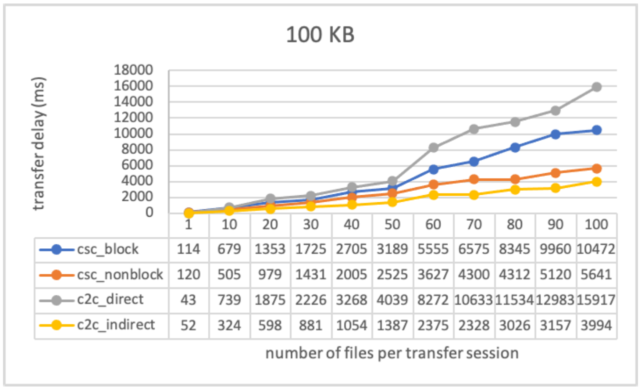
FIGURE 12. Transfer time according to the number of files per transfer session (1KB).



	1	10	20	30	40	50	60	70	80	90	100
csc_block	86	389	767	874	919	1174	2891	3789	4022	4618	5005
csc_nonblock	98	213	413	546	710	856	1218	2170	2221	2917	3686
c2c_direct	30	577	1397	1935	2392	2907	3704	4709	5928	6202	6395
c2c_indirect	31	94	160	162	254	354	427	464	471	633	650

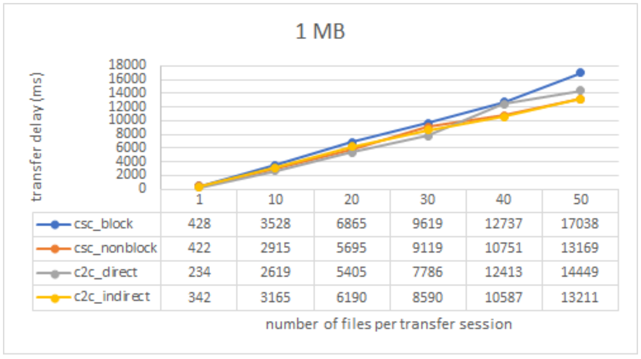
FIGURE 13. Transfer time according to the number of files per transfer session (10KB).

Figures 12 to 15 show the results of measuring the transmission time according to the number of files transmitted at one time. For this experiment, we prepared a group of files with the specific size by copying the original file. In one transfer session of a file group, we transferred the group of files continuously from the first to the last file with no transfer interval. Interestingly, the transmission time



	1	10	20	30	40	50	60	70	80	90	100
csc_block	114	679	1353	1725	2705	3189	5555	6575	8345	9960	10472
csc_nonblock	120	505	979	1431	2005	2525	3627	4300	4312	5120	5641
c2c_direct	43	739	1875	2226	3268	4039	8272	10633	11534	12983	15917
c2c_indirect	52	324	598	881	1054	1387	2375	2328	3026	3157	3994

FIGURE 14. **Transfer time according to the number of files per transfer session (100KB).**



	1	10	20	30	40	50
csc_block	428	3528	6865	9619	12737	17038
csc_nonblock	422	2915	5695	9119	10751	13169
c2c_direct	234	2619	5405	7786	12413	14449
c2c_indirect	342	3165	6190	8590	10587	13211

FIGURE 15. **Transfer time according to the number of files per transfer session (1MB).**

is largely divided into two categories. As the number of files transmitted at one time increases, the group using only the default non-blocking channel and thread (csc_nonblock, c2c_indirect) shows lower transmission time than the group using the dedicated blocking channel and thread (csc_block, c2c_direct). For example, in the case of transferring 50 files of 1KB (Figure 12), the indirect C2CFTP (c2c_indirect) compared to the direct C2CFTP (c2c_direct) reduced the file transfer time by 88.6%. Although the blocking methods (csc_block, c2c_direct) reduce the overhead of creating and destroying a separate thread as they get and release an available thread from a thread pool, they still need the process of starting the separate thread dedicated to file transfer on a dedicated channel for each file and terminating the dedicated thread when file transfer is completed. If the number of files being transmitted at one time is large, the number of thread start and stop also increases, and accordingly, the thread management overhead increases. As a result, we analyzed that this thread management overhead has an effect on the file transfer delay. When we measured the transfer time even more until the number of files was 100, the tendency of the delay difference among methods kept same as shown in Figure 13 and 14.

In another aspect, as the file size increased, the difference in the transmission time reduction rate between these two groups became smaller. For example, if 50 files of 1MB are transferred (Figure 15), the indirect C2CFTP (c2c_indrect) still has a shorter file transfer time than the direct C2CFTP (c2c_direct), but the difference is 8.6% decrease compared to the 50 files of 1KB case. This is because the effect of the total transmission delay due to the thread management overhead is relatively small as the transmission time of the file itself increases as the file size increases. The direct C2CFTP (c2c_direct) showed better performance until the number of transferred files was 30. We estimated that this is also because the accumulated thread management overhead has less effect on the whole transfer delay until the number of files became 30. When comparing the existing method (csc_nonblock) with the proposed method (c2c_indirect), the proposed method showed a shorter transmission time when transferring files smaller than 1 MB. When transferring a 1MB file, the performance of either side was not significantly superior.

Through the results of the experiment, we confirmed that the file transfer performance can be improved by selecting and using one of the proposed direct and indirect C2CFTPs depending on the situation when transferring a bundle of files. The indirect C2CFTP is suitable when a client sends files smaller than 1 MB in size. The direct C2CFTP is suitable for up to 30 files when each file size is over 1MB, and the indirect C2CFTP shows better performance when transferring a larger number. There is another case where the indirect C2CFTP is suitable. If the receiving client is located in a virtual private network different from the sending client, and the sending client cannot directly access the receiving client, we can use the indirect C2CFTP using relay of the server, although the direct C2CFTP is not available.

V. RELATED WORK

File transfer protocol (FTP) [1], [2] is a representative standard protocol related to file transfer. Most commercial file transfer clients and servers are implemented according to this standard specification for compatibility. Looking at the file transfer procedure, it is similar to the CM blocking file transfer method in that it separately manages channels for the control message transfer and the file data transfer. GridFTP [3], [4], which is an extension of the existing FTP, was developed to adjust the TCP buffer size and use multiple channels for secure, reliable, and high-performance data transmission. In other words, it is designed as a useful protocol when transferring large files or small files in bulk. In order to efficiently transfer large-scale files, the BitTorrent protocol [5] is a file transfer method that dramatically reduces transmission time by receiving file blocks from multiple source nodes in the peer-to-peer architecture. It is mainly used for transferring large files such as video files. Dual-Direction FTP (DDFTP) [6] technology was proposed as a method to quickly transfer large files by utilizing multiple replicated FTP servers. In order to transfer file blocks from multiple replicated FTP servers in parallel, DDFTP assigns file blocks to different servers.

A recent study [7] proposed a file transfer method that can perform synchronization of multiple files between various heterogeneous devices such as N-screens in a short time. In traditional FTP, the data channel had to be connected repeatedly every time the file was transferred. To solve this problem, the method of maintaining the data channel is used. Fan's research [8] proposed FTP-NDN technology, which is a file transfer method based on peer-to-peer encryption in a Named Data Network (NDN) environment. The main purpose of this technique is not to receive a file only from a fixed source node, but to find a node with the same file close to the receiver node's surroundings and receive the file in encrypted form. Bormann and Shelby [9] devised a block-wise transfer of files in CoAP (Constrained Application Protocol) [20]. While the other existing methods above use TCP as the transport layer protocol, but CoAP uses UDP for simplicity. However, the transmitting node has to wait for an acknowledgment (ack) message from the receiving node for each block transmitted and retransmits if the previous block transmission fails. There were also other recent researches on file transfer for cloud computing environment [10] and IoTs [11]. Lin's research [10] proposed algorithms to solve a single-file transfer scheduling (SFTS) and a multi-file transfer scheduling (MFTS) problems when one or more big files are transferred over multiple paths. Liu's research [11] proposed a camera file transfer system, but it mainly focuses on security vulnerabilities of photo file transfer in device-to-device networks.

Communication frameworks or middlewares for online social networks [14]–[18] provide various communication services to applications, among which image file transfer is required as a basic function. However, since these studies focus on efficient interaction services between users, they do not specifically mention how they developed file transfer services.

In summary, existing methods related to file transfer protocols mainly focus on methods for quickly transferring a large number of large files. In addition, the existing file transfer methods deal only with the case where the client and the server directly connected in the client-server structure, but the file transfer between the clients is not considered separately. In other words, in order to transfer files between clients in the conventional method, the two-step transmission (transmission from the sending client to the server, and transmission from the server to the receiving client) is required.

TABLE 5. CM event header.

Number of bytes	Field data type	Description
4	int	Event length
4	int	Event type (CMInfo.CM_FILE_EVENT)
4	int	Event ID
2	short	Length of sender field (n)
n	String	Sender name of this event
2	short	Length of receiver field (n)
n	String	Receiver name of this event
2	short	Length of session handler field (n)
n	String	Session name for handling this event
2	short	Length of group handler field (n)
n	String	Group name for handling this event
2	short	Length of distribution session field (n)
n	String	Session name for distribution of this event
2	short	Length of distribution group field (n)
n	String	Group name for distribution of this event

TABLE 6. REQUEST_PERMIT_PULL_FILE.

Number of bytes	Field data type	Description	Field type
4	int	Event length	Event header
4	int	Event type (CMInfo.CM_FILE_EVENT)	
4	int	Event ID (CMFileEvent.REQUEST_PERMIT_PULL_FILE)	
...	...	Other CM event header fields	
2	short	Length of file sender field (n)	Event body
n	String	File sender name	
2	short	Length of file receiver field (n)	
n	String	File receiver name	
2	short	Length of file name (n)	
n	String	File name	
4	int	ID of content attaching this file	
1	byte	File append mode	
4	int	Port number of server socket channel	

TABLE 7. REPLY_PERMIT_PULL_FILE.

Number of bytes	Field data type	Description	Field type
4	int	Event length	Event header
4	int	Event type (CMInfo.CM_FILE_EVENT)	
4	int	Event ID (CMFileEvent.REPLY_PERMIT_PULL_FILE)	
...	...	Other CM event header fields	
2	short	Length of file sender field (n)	Event body
n	String	File sender name	
2	short	Length of file receiver field (n)	
n	String	File receiver name	
2	short	Length of file name (n)	
n	String	File name	
4	int	Return code	
4	int	ID of content attaching this file	

VI. CONCLUSION

In this paper, we proposed a file-transfer protocol (C2CFTP) that efficiently performs file transfer between clients in a client-server architecture. The proposed C2CFTP is included as a file transfer service of CM, a communication framework, and provides two methods: the direct transmission (direct C2CFTP) and the indirect transmission (indirect C2CFTP). As a result of the experiment, C2CFTP was able to reduce the file transfer delay time between clients by omitting unnecessary intermediate file I/O from existing file transfer methods and improving dedicated channel management. The direct C2CFTP is suitable for the transmission of large files, and the indirect C2CFTP is effective when sending several small files in a bundle or when direct connection between clients is not possible.

When CM transfers a bunch of files, the direct C2CFTP showed a worse performance comparing to the indirect C2CFTP due to the repetition of thread start and stop per file. For our future work, we plan to improve the performance

TABLE 8. REQUEST_PERMIT_PUSH_FILE.

Number of bytes	Field data type	Description	Field type
4	int	Event length	Event header
4	int	Event type (CMInfo.CM_FILE_EVENT)	
4	int	Event ID (CMFileEvent.REQUEST_PERMIT_PUSH_FILE)	
...	...	Other CM event header fields	
2	short	Length of file sender field (n)	Event body
n	String	File sender name	
2	short	Length of file receiver field (n)	
n	String	File receiver name	
2	short	Length of file path (n)	
n	String	File path	
8	long	File size	
1	byte	File append mode	
4	int	ID of content attaching this file	

TABLE 9. REPLY_PERMIT_PUSH_FILE.

Number of bytes	Field data type	Description	Field type
4	int	Event length	Event header
4	int	Event type (CMInfo.CM_FILE_EVENT)	
4	int	Event ID (CMFileEvent.REPLY_PERMIT_PUSH_FILE)	
...	...	Other CM event header fields	
2	short	Length of file sender field (n)	Event body
n	String	File sender name	
2	short	Length of file receiver field (n)	
n	String	File receiver name	
2	short	Length of file path (n)	
n	String	File path	
8	long	File size	
1	byte	File append mode	
4	int	ID of content attaching this file	
4	int	Port number of server socket channel	
4	int	Return code	

TABLE 10. START_FILE_TRANSFER, START_FILE_TRANSFER_CHAN.

Number of bytes	Field data type	Description	Field type
4	int	Event length	Event header
4	int	Event type (CMInfo.CM_FILE_EVENT)	
4	int	Event ID (CMFileEvent.START_FILE_TRANSFER or CMFileEvent.START_FILE_TRANSFER_CHAN)	
...	...	Other CM event header fields	
2	short	Length of file sender field (n)	Event body
n	String	File sender name	
2	short	Length of file receiver field (n)	
n	String	File receiver name	
2	short	Length of file name (n)	
n	String	File name	
8	long	File size	
4	int	ID of content attaching this file	
1	byte	File append mode	

of the direct C2CFTP by reducing the thread management overhead.

We also plan to devise a method for CM to dynamically determine the appropriate file transfer method according to the current situation of the application. The current CM allows a server application to select a file transfer method between the direct and indirect C2CFTP in the CM configuration file and client applications follow the server decision to keep a consistent file transfer policy. By adding an option

TABLE 11. START_FILE_TRANSFER_ACK, START_FILE_TRANSFER CHAN ACK.

Number of bytes	Field data type	Description	Field type
4	int	Event length	Event header
4	int	Event type (CMInfo.CM_FILE_EVENT)	
4	int	Event ID (CMFileEvent.START_FILE_TRANSFER_ACK or CMFileEvent.START_FILE_TRANSFER_CHAN_ACK)	
...	...	Other CM event header fields	
2	short	Length of file sender field (n)	Event body
n	String	File sender name	
2	short	Length of file receiver field (n)	
n	String	File receiver name	
2	short	Length of file name (n)	
n	String	File name	
4	int	ID of content attaching this file	
8	long	Received file size	

TABLE 12. CONTINUE_FILE_TRANSFER.

Number of bytes	Field data type	Description	Field type
4	int	Event length	Event header
4	int	Event type (CMInfo.CM_FILE_EVENT)	
4	int	Event ID (CMFileEvent.CONTINUE_FILE_TRANSFER)	
...	...	Other CM event header fields	
2	short	Length of file sender field (n)	Event body
n	String	File sender name	
2	short	Length of file receiver field (n)	
n	String	File receiver name	
2	short	Length of file name (n)	
n	String	File name	
4	int	ID of content attaching this file	
4	int	Length of file block (n)	
n	byte array	File block	

TABLE 13. END_FILE_TRANSFER, END_FILE_TRANSFER_CHAN.

Number of bytes	Field data type	Description	Field type
4	int	Event length	Event header
4	int	Event type (CMInfo.CM_FILE_EVENT)	
4	int	Event ID (CMFileEvent.END_FILE_TRANSFER or CMFileEvent.END_FILE_TRANSFER_CHAN)	
...	...	Other CM event header fields	
2	short	Length of file sender field (n)	Event body
n	String	File sender name	
2	short	Length of file receiver field (n)	
n	String	File receiver name	
2	short	Length of file name (n)	
n	String	File name	
8	long	File size	
4	int	ID of content attaching this file	

that the file transfer service actively adapts according to the application situation, CM will be able to provide a desired service while minimizing the manual configuration of the application.

Our last future research plan is to analyze the performance of the proposed file transfer protocol in an extended environment. For example, we plan to increase the number of clients and perform performance comparison of file transfer protocols through experiments in different situations, such as one-to-many file transfer rather than one-to-one transfer.

TABLE 14. END_FILE_TRANSFER_ACK, END_FILE_TRANSFER_CHAN_ACK.

Number of bytes	Field data type	Description	Field type
4	int	Event length	Event header
4	int	Event type (CMInfo.CM_FILE_EVENT)	
4	int	Event ID (CMFileEvent.END_FILE_TRANSFER_ACK or CMFileEvent.END_FILE_TRANSFER_CHAN_ACK)	
...	...	Other CM event header fields	
2	short	Length of file sender field (n)	Event body
n	String	File sender name	
2	short	Length of file receiver field (n)	
n	String	File receiver name	
2	short	Length of file name (n)	
n	String	File name	
8	long	File size	
4	int	Return code	
4	int	ID of content attaching this file	

TABLE 15. ADD_BLOCK_SOCKET_CHANNEL, REMOVE_BLOCK_SOCKET_CHANNEL.

Number of bytes	Field data type	Description	Field type
4	int	Event length	Event header
4	int	Event type (CMInfo.CM_SESSION_EVENT)	
4	int	Event ID (CMSessionEvent.ADD_BLOCK_SOCKET_CHANNEL or CMSessionEvent.REMOVE_BLOCK_SOCKET_CHANNEL)	
...	...	Other CM event header fields	
2	short	Length of channel name (n)	Event body
n	String	Channel name (requester name)	
4	int	Channel number (>= 0)	

TABLE 16. ADD_BLOCK_SOCKET_CHANNEL_ACK, REMOVE_BLOCK_SOCKET_CHANNEL_ACK.

Number of bytes	Field data type	Description	Field type
4	int	Event length	Event header
4	int	Event type (CMInfo.CM_SESSION_EVENT)	
4	int	Event ID (CMSessionEvent.ADD_BLOCK_SOCKET_CHANNEL_ACK or CMSessionEvent.REMOVE_BLOCK_SOCKET_CHANNEL_ACK)	
...	...	Other CM event header fields	
2	short	Length of channel name (n)	Event body
n	String	Channel name (respondent name)	
4	int	Channel number (>= 0)	
4	int	Return code (1: success, 0: failure)	

APPENDIX A
CM FILE EVENT FORMAT

See Tables 5–14.

APPENDIX B
CM SESSION EVENT FORMAT

See Tables 15 and 16.

REFERENCES

[1] J. Postel and J. K. Reynolds, *File Transfer Protocol*, document RFC 765, 1980. [Online]. Available: https://tools.ietf.org/html/rfc765

[2] P. Hethmon, *Extensions to FTP*, document RFC 3659, 2007. [Online]. Available: https://tools.ietf.org/html/rfc3659

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link, "The globus striped GridFTP framework and server," in *Proc. ACM/IEEE SC Conf. (SC)*, Seattle, WA, USA, 2005, p. 54, doi: 10.1109/sc.2005.72.

[4] W. Allcock. (2003). GridFTP: Protocol extensions to FTP for the Grid. Global Grid Forum Draft. [Online]. Available: https://www.ogf.org/documents/GFD.20.pdf

[5] D. Qiu and R. Srikant, "Modeling and performance analysis of bittorrent-like peer-to-peer networks," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, vol. 34, Portland, OR, USA, 2004, pp. 367–378, doi: 10.1145/1015467.1015508.

[6] J. Al-Jaroodi and N. Mohamed, "DDFTP: Dual-direction FTP," in *Proc. 11th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, vol. 1, Newport Beach, CA, USA, May 2011, pp. 504–513, doi: 10.1109/ccgrid.2011.32.

[7] D. Kim, J. Baek, P. S. Fisher, and S. Kim, "FFTP: A fast file transfer protocol for home N-screen platform," *Pers. Ubiquitous Comput.*, vol. 22, no. 1, pp. 143–152, Feb. 2018, doi: 10.1007/s00779-017-1082-5.

[8] C.-I. Fan, I.-T. Chen, C.-K. Cheng, J.-J. Huang, and W.-T. Chen, "FTP-NDN: File transfer protocol based on re-encryption for named data network supporting nondesignated receivers," *IEEE Syst. J.*, vol. 12, no. 1, pp. 473–484, Mar. 2018.

[9] C. Bormann and Z. Shelby, *Block-Wise Transfers in the Constrained Application Protocol (CoAP)*, document RFC 7959, 2016. [Online]. Available: https://tools.ietf.org/html/rfc7959

[10] C. Lin, Y. Bi, G. Han, J. Yang, H. Zhao, and Z. Liu, "Scheduling for time-constrained big-file transfer over multiple paths in cloud computing," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 2, no. 1, pp. 25–40, Feb. 2018, doi: 10.1109/tetci.2017.2755692.

[11] T. Liu, K. Liu, Y. Cheng, and L. X. Cai, "Security analysis of camera file transfer over Wi-Fi," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Shanghai, China, May 2019, pp. 1–6, doi: 10.1109/icc.2019.8761891.

[12] M. Lim, "CMSNS: A communication middleware for social networking and networked multimedia systems," *Multimedia Tools Appl.*, vol. 76, no. 17, pp. 18119–18135, Sep. 2017, doi: 10.1007/s11042-016-3839-7.

[13] Y. Moon and M. Lim, "An enhanced file transfer mechanism using an additional blocking communication channel and thread for IoT environments," *Sensors*, vol. 19, no. 6, p. 1271, Mar. 2019, doi: 10.3390/s19061271.

[14] D. Brooker, T. Carey, and I. Warren, "Middleware for social networking on mobile devices," in *Proc. 21st Austral. Softw. Eng. Conf.*, Auckland, New Zealand, 2010, pp. 202–211.

[15] 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. (WOSN)*, Barcelona, Spain, 2009, pp. 49–54.

[16] S. B. Mokhtar, L. McNamara, and L. Capra, "A middleware service for pervasive social networking," in *Proc. Int. Workshop Middleware Pervas. Mobile Embedded Comput. (M-PAC)*, Urbana Champaign, IL, USA, 2009, Art. no. 2.

[17] 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.

[18] 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. ICST Conf. Mobile Wireless Middleware, Operating Syst. Appl.*, Innsbruck, Austria, 2008, Art. no. 27.

[19] M. Lim, "Internal forwarding scheme for consistent event transmission in distributed multi-user systems," in *Proc. NCM*, Seoul, South Korea, 2010, pp. 563–568.

[20] Z. Shelby, K. Hartke, and C. Bormann, *The Constrained Application Protocol (CoAP)*, document RFC 7252, 2014. [Online]. Available: https://tools.ietf.org/html/rfc7252

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, in 2000 and 2006, respectively. He was a Senior Researcher with the MIRALab, University of Geneva, Geneva, Switzerland, from 2006 to 2008. He was an Assistant and an Associate Professor with the Department of Internet and Multimedia Engineering, Konkuk University, Seoul, South Korea, from 2009 to 2016, where he is currently a Professor with the Department of Smart ICT Convergence. His current research interests include communication middleware and framework, efficient event transmissions, and content distribution in distributed systems.