TCP/IP Communication System for Multiple Client Connections

Atsushi Matsubara

Production Division

1 Introduction

Transmission Control Protocol/Internet Protocol (TCP/IP) communication, which the historical background of its development is derived from a military communication technology and leads to the rapid spread of the internet communication, has been used worldwide as the most excellent communication protocol from the aspect of usefulness and convenience, while the network communication technologies have been diversified.

The links between nodes in TCP/IP communication realize using IP address and Port number. The former address (dotted-quad notation) is uniquely assigned to a node as the identification number and the latter number (16-bit unsigned binary one) as the input-output interface for data stream. Note that a node means a terminal or a personal computer.

Two nodes forms server-client architecture in TCP/IP communication. The client requests data or service to the server, whereas the latter returns the response or the processing result to the former. The formations of the architecture are “one to one (a server to a client)” and “one to many (a server to clients)”. Moreover, TCP/IP communication adopts an asynchronous communication whose method does not synchronize transmission and reception for data on the architecture.

The author reports on TCP/IP communication system for multiple clients. The system has the following features: 1) Operating the class instances (e.g., button, textbox) on the developed application with the pointing devices (mouse or touch pad) leads to successful communication, 2) The server creates the client-customized Socket and Window class instances when connecting with a client, 3) The transmission of text or image file is possible on the server-client line, 4) The server allows clients to connect whenever it is running its own application.

The system is programmed in Visual Basic language and is provided as Windows Presentation Foundation (WPF) application. The development and the execution environments for it are “Microsoft Visual Studio Community 2022” and “Microsoft .Net Framework ver.4.8”, respectively.

2 System design

2.1 Graphical User Interface (GUI)

Figure 1 shows the customized Window class instance for client. The top of the Window sets TextBox and Button class instances: the two former instances are respectively to display IP address and port number, and the four latter instances to transmit data (“Submit”), establish a server-client line (“Connection”), terminate the communication line (“Disconnected”) and close the Window (“End”).

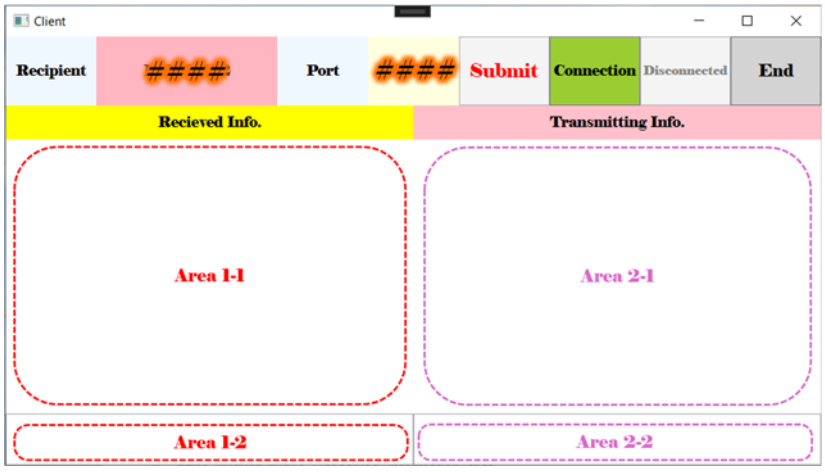


Figure 1. Client-customized Window

The middle and bottom of it sets

TextBlock and TextBox class instances. The former (“Area 1-1”, “Area 2-1”) and the latter (“Area 1-2”, “Area 2-2”) instances are respectively to display the messages and the file names. In addition, the left (“Area1-1” or “Area1-2”) / the right side (“Area 2-1” or “Area 2-2”) presents the received / transmitting information from / to the server.

Transmission procedure of file follows that a client clicks the Button class instance “Submit” after dragging and dropping a file on the TextBox class instance “Area 2-2”. In contrast, showing a file name in the instance “Area 1-2” means successfully data reception. Raising the double-click event on the “Area 1-2” or “Area 2-2” leads to open the file.

It is noted that the showed IP address and Port number are masked using sequences of number sign (‘#’) characters because their notations are privacy-related information.

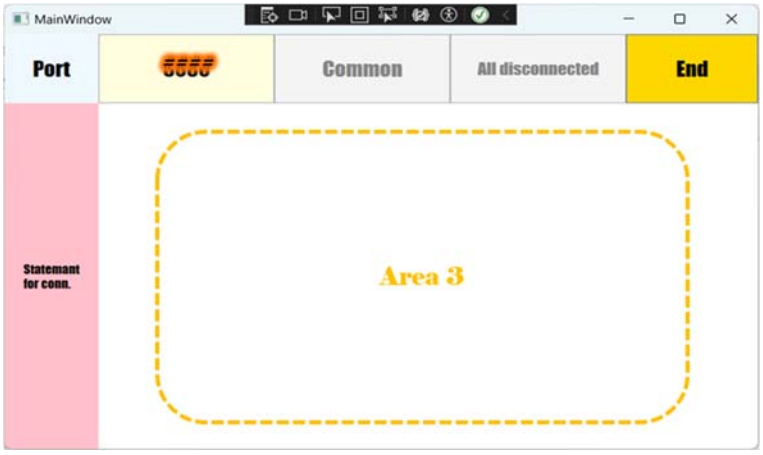


Figure 2. Server-customized Window

Figure 2 shows the server-customized Window class instance. The Window has three Button class instances (“Common”, “All disconnected”, and “End”) , whose role are respectively to create (instantiate) Window for transmitting a shared file to multiple clients (see Figure 4), to terminate all the communication lines and to self-close the Window. The TextBlock class instance “Area 3” displays the communication-related messages.

Window in Figure 3 is a class instance generated depending on each client-access for establishing communication. The Window (customized for each client) has the presentable TextBox class instance for the client’s IP address, although its layout is similar to a client’s one (see Figure 1). The server has a mechanism for transferring or distributing client information to the associated Window when receiving it (e.g., client A’s information to Window A, B’s it to B, ⋯). Therefore, the server’s customized Window in Figure 2 takes on a role of hub aggregating all client’s information.

The server allows the Button instance “Common” on server-customized Window to create Window for transmitting a shared file as shown in Figure 4. This Window contains class instances of ListBox, Button, TextBlock “Area 6-1” and TextBox “Area 6-2”. The ListBox allows the server to select the destinations (IP addresses) for transmitting a file. The four Button class instances (“All”,” Setting”,” Submit” and “End”) have the capability to select immediately all destinations, decide them, transmit a file in “Area 6-2” and self-close the Window, respectively. The server automatically closes the Window using a timer after transmitting (that is, clicking the instance “Submit”). TextBlock “Area 6-1” displays this transmission-related messages.

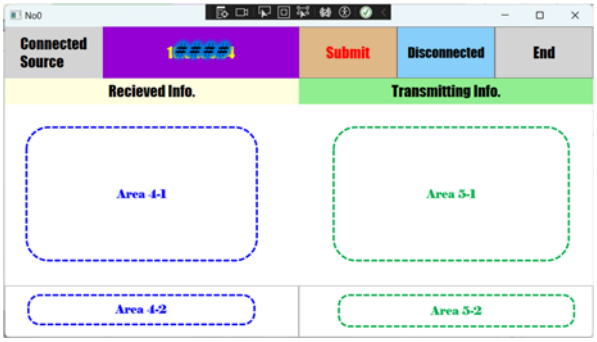


Figure 3. Customized Window for each client

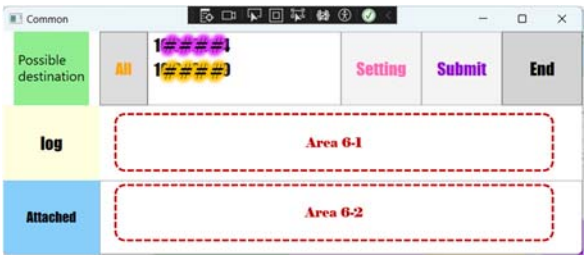


Figure 4. Window for transmitting a shared file

2.2 Data Configuration

The achievements of the communication-related events (e.g., establishing, maintaining, terminating, and data transmission / reception) depend on the messages representing in Figure 5. The system has the seven messages each transmitted from client and server: the notation for former is "**mC*n***"and that of the latter is "**mS*n***", where message number $\boldsymbol{n} \in \{0, \cdots, 6\}$.

The message consists of a sequence of characters as follows:

$$\text{"ClientIP"} + \text{":"} + \boldsymbol{n} + \text{":"} + \text{a message} \quad (1)$$

"+" character is the operator to concatenate two strings, and the colon ":" is a delimiter to split the received messages. You will see that the string "ClientIP" (or "ServerIP" in Figure 5) means the client's (or server's) IP address.

In addition, file information trails behind the above message configuration (1):

$$\text{message configration}(1) + \text{"\_\_\_\_\_"} + \text{a file name} + \text{"???"} + \text{contents of the file} \quad (2)$$

The sequences of the seven underscore "_" and three question mark "?" characters are also splitting delimiters.

The system always receives one of the above messages from a destination regardless of an active or a passive event. The active event, "establishing", "transmitting", "self-closing", etc., displays the message in "Area 2-1 (or Area 5-1)" as the transmitting information. The passive one, "accessed", "received", "terminated", etc., does it in "Area 1-1 (or Area 4-1)" as the received information (see Figure 1 and 3).

Client

type	message
mC0	"ClientIP:0:Accessed from the Client (ClientIP)"
mC1	"ClientIP:1:Data-Received from the Client (ClientIP)"
mC2	"ClientIP:2:Submitted Updated Data to the Client (ClientIP)"
mC3	"ClientIP:3:Submitted Novelty Data to the Client (ClientIP)"
mC4	"ClientIP:4:Disconnected from the Client (ClientIP)"
mC5	"ClientIP:5:The Server (ServerIP) disconnected"
mC6	"ClientIP:6:The Server (ServerIP) disconnected All Clients"

Server

type	message
mS0	"ClientIP:0:Established Connection with the Server (ServerIP)"
mS1	"ClientIP:1:Submitted Data to Server (ServerIP)"
mS2	"ClientIP:2:Receiverd-Updated Data from the Server (ServerIP)"
mS3	"ClientIP:3:Receiverd-Novelty Data from the Server (ServerIP)"
mS4	"ClientIP:4:Disconnected from my Client (ServerIP)"
mS5	"ClientIP:5:Disconnected from the Server (ServerIP)"
mS6	"ClientIP:6:All Clients disconnected from the Server (ServerIP)"

Figure 5. Message configuration list

2.3 Communication protocol

The protocol of communication sets as follows based on the above messages:

1. [Establishing communication] Sever receives the message "**mC0**", whereas a client does the "**mS0**". The sever generates the customized Socket and Window class instances for the client and waits again others to access.
2. [Transmitting a file] The sever receives the message configuration (2) including the "**mC1**" and the client does the "**mS1**". The former conversely gets the "**mC2**" (or "**mC3**"), whereas the latter does the message configuration (2) including the "**mS2**" (or "**mS3**").
3. [Terminating communication] The server closes the Socket and Window class instances for the client after receiving the "**mC4**" (or "**mC5**"," **mC6**"). The client can close the application after receiving the "**mS4**" (or "**mS5**"," **mS6**").
4. [Re-establish] The server will re-establish communication with the client if its application running.

Therefore, Figure 6 represents a stream diagram for the above messages.

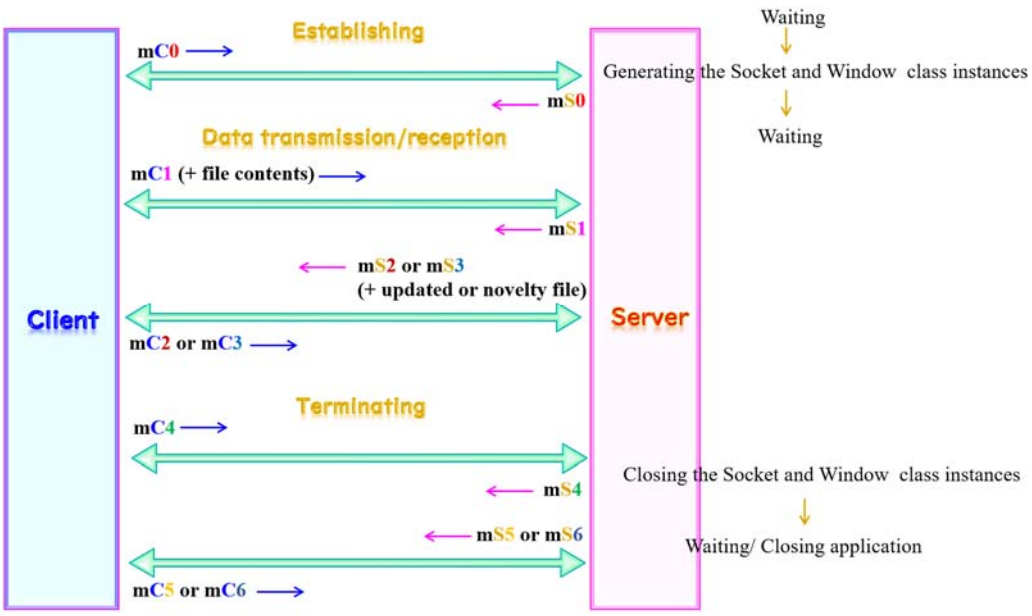


Figure 6. Stream diagram for the messages

3 Algorithm and implementation [1), 2)]

3.1 Client's application

The system assigned an endpoint to a node (client or server) using Socket class belonging to "System.Net.Sockets" namespace. Code 1 shows pseudo code which means the processing flow for connecting to the server.

A server-client line for communication establishes using Socket.Connect() method with an instantiated EndPoint class belonging to "System.Net" namespace. The EndPoint class instance has information of an IP address and a Port number. Socket.BeginReceive() method for receiving data has six arguments, of which the four noticeable ones are the first one that specifies the received data as a one-dimensional byte array (*Buffer* in Code 1), the third that does size of the array, the fifth that an instantiated AsyncCallback delegate including in "System" namespace and the sixth, a client-defined class instance ("the *Client*'s info.") to store a received data.

The AsyncCallback delegate refers to function "*Function* 2" called whenever receiving data; that is, the function is wrapped with the delegate. The called function "*Function* 2" as Callback one has the following tasks: (1) extracting the received information using the AsyncState property of the instantiated IAsyncResult class; (2) accomplishing the reception-related procedures; (3) splitting data, saving it to a file or displaying it in TextBlock or TextBox instance; (4) calling recursively "*Function* 2" to continue receiving data reusing Socket.BeginReceive() method.

Socket.available property in "*Function* 2" gets the readable size of the received data per a time (or calling "*Function* 2"). The property returning non-zero means that the client's application reuses Socket.BeginReceive() method (that is, calls repeatedly "*Function* 2"); otherwise that it accomplishes receiving all data from the server. The maximum value of the Socket.available property is the size of the array specified by Socket.BeginReceive() method's argument specified in the third. Therefore, the client's application once receives all data if it knows the whole size of data prior to the server's transmitting, it dividedly does them otherwise.

Now setting the whole size of the receiving data as A, the value of Socket.available property (=a size of the array) as B, the times X of calling "*Function* 2" is as follows:

$$X = \begin{cases} \mathrm{int}\left[\frac{A}{B}\right] & (C = 0) \\ \mathrm{int}\left[\frac{A}{B}\right] + 1 & (C \neq 0) \end{cases}$$

where, remainder $C = A \bmod B$. The int[·] casts a value in [] to a round-down integer.

The AsyncCallback delegate generates a novelty thread for the data reception, which ensures that the receive processing avoids interfering with the operation of the Window's instances (Button, TextBox and so on) on the main thread.

The system thus can access the Window's instances from the novelty thread for receiving using Application.Current.Dispatcher. Invoke() method belonging to "System.Windows" namespace.

Data Processing uses the following methods:

(1) MemoryStream.Write(), the MemoryStream class method included in "System.IO" namespace, stores the received data to client-defined memory (*ReceivedData*); (2) Default.GetString() / Default.GetBytes(), the Encoding class methods included in "System.Text" namespace, can cast a byte array to a string / vice versa; (3) System.FileStream.Read() / Write() method reads/writes from/to a file.

```
----Global (or Public) Declaration----
Dimension Client as Socket class variable
Dimension ReceivedData as MemoryStream class

----Function1 area----
Creating the class instances for Client
Creating the Endpoint class instances using IP address and Port number for Server
Client.Connect(EndPoint class instance)
Client.BeginReceive(Buffer,…, AsyncCallback delegate to call the Function 2, the Client's info.)

----Function 2 area (argument: IAsyncResult instance including the Client's info.) ----
Client extracted from IAsyncResult instance (using AsyncState property)
len the received byte number returned from Client.EndReceive() method
Creating the class instance for ReceivedData
ReceivedData.Write(Buffur, Buffer.length,etc)
Casting ReceivedData to data as String using Encoding.default.GetString() method
if Client.available=0
   ReceivedData.Split(delimit char)
      ---Processing depended on the message number
   ReceivedData.Close()
end if
Client.BeginReceive(Buffer,…, AsyncCallback delegate to call the Function 2, the Client's info.)
```

Code 1. Pseudo code for a client

Omitting the details, the system uses Socket.Send() method to transmit data, whose first argument specifies the transmitting data as a byte array and second one does size of the array.

3.2 Server's application

Pseudo-code for the server is introduced in Code 2. The server facilitates access requests from multiple clients through the following procedure: (1) associating the server with the endpoint for the communication system using Socket.Bind () method which has the self-IP address and the port information; (2) preparing the queue and keeping the waiting state for a client using Socket.Listen () method, and A queue means that clients sequentially wait for access to the server; (3) providing an asynchronous communication system for Socket class instance using Socket.BeginAccept() method. The first and second arguments of the method specifies an instantiated AsyncCallback delegate and the Socket class instance for the server (*Server* in Code 2), respectively.

The function "*Function* A" wrapped with the AsyncCallback delegate is the callback one called for each client access. The function has the following tasks: (1) extracting the *Server* instance through IAsyncResult.AsyncState property; (2) returning the accepted client's Socket instance(*client*) using EndAccept() method; (3) storing the *client* in the memory area (*Client* [*i*]) pointed to by a unique index number (setting it as variable *i*) assigned to each accepted client in the server-

defined array, which allows the system to operate or manage smoothly the communication for each client; (4) preparing to call the Callback function "*Function* B" using Socket.BeginReceive() method; (5) calling recursively "*Function* A" to connect to a novelty client reusing Socket.BeginAccept() method.

Function B, implements the receive processing per client, is compliant with "*Function* 2" in Code 1.

The number of threads in the server process is four on a client-server line: the details are for operating Window (see Figure 2), for accepting the client's request, for operating Window for the client (see Figure 3) and for receiving the client's data. Total $(2N + 2)$ threads thus exist independently and simultaneously if the number of the accepted clients is N.

```
----Global (or Public) Declaration----
Dimension Server as Socket class variable
Dimension Client[] as array of Socket class
Dimension ReceivedData as MemoryStream class

----Window loaded area----
Creating the class instances for Server and Client []
Creating the Endpoint class instances using IP address and Port number for Server
Server.Bind(EndPoint class instance)
Server.Listen()
Server.BeginAccept(AsyncCallback delegate to call the Function A, Server)

----Function A area (argument: IAsyncResult instance including Server) ----
Server extracted from IAsyncResult instance (using AsyncState property)
client instance for Socket class returned from Server.EndAccept() method
   Processing---extraction the client's information from the client instance
                and store them to the variables
i ← new or specified index number
Client[i] ← the client instance
Buffer[] byte array for the received data
Client[i].BeginReceive(Buffer,···, AsyncCallback delegate to call the Function B, Client[i]'s info.)
Server.BeginAccept(AsyncCallback delegate to call the Function A, Server)

----Function B area (argument: IAsyncResult instance including Client[i]'s info.) ----
   Processing --- conformable to Function 2 in Code1.
Client[i].BeginReceive(Buffer,···, AsyncCallback delegate to call the Function B, Client[i]'s info.)
```

Code 2. Pseudo code for server

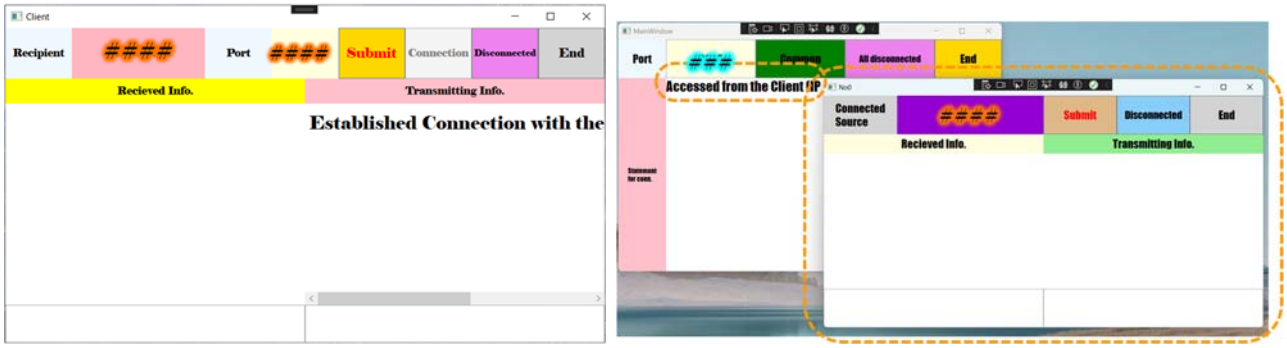
4 Communication example

The chapter shows the case of two clients (from here, three nodes are called "Server", "Client1" and "Client2"). The procedure of the example is set as follows:

1. Client 1 establishes communication line with Server.
2. Client 1 transmits the text file A (the extension: "txt", the size: 4kbyte) to Server.
3. Server transmits the text file B ("csv", 203kbyte) to Client 1.
4. Client 2 establishes communication line with Server.
5. Server transmits the portable document format file C ("pdf", 357kbyte) to all clients.
6. Client 1 terminates the communication with Server.
7. Server terminates the communication with Client 2, which disconnects all communication lines.

Figure 7 shows states of Window class instances for Client1 and Server after establishing the communication line. The achievement of establishing is based on the fact that Server displays the received message "**mC0**" in "Area 3" and generates the Window class instance for Client 1 whereas Client1 does the received "**mS0**" in "Area 2-1".

Figure 8 illustrates the situation after transmitting file A to the server. Server / Client 1 displays the received message "**mC1**" / "**mS1**" in "Area 4-1" / "Area 2-1" when Client 1 clicks the button class instance "Submit" after dragging and dropping the file A on "Area2-2". In addition, it is confirmed from the Server side of the figure that the received file A (in

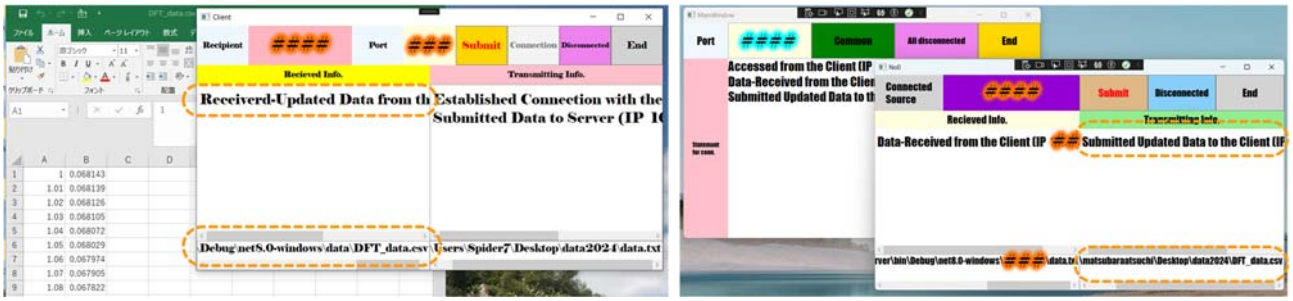


(a) Client 1's side (b) Server's side

Figure 7. Establishing the Client 1- Server line

(a) Client 1 (b) Server

Figure 8. Transmitting the file A to Server



(a) Client 1 (b) Server

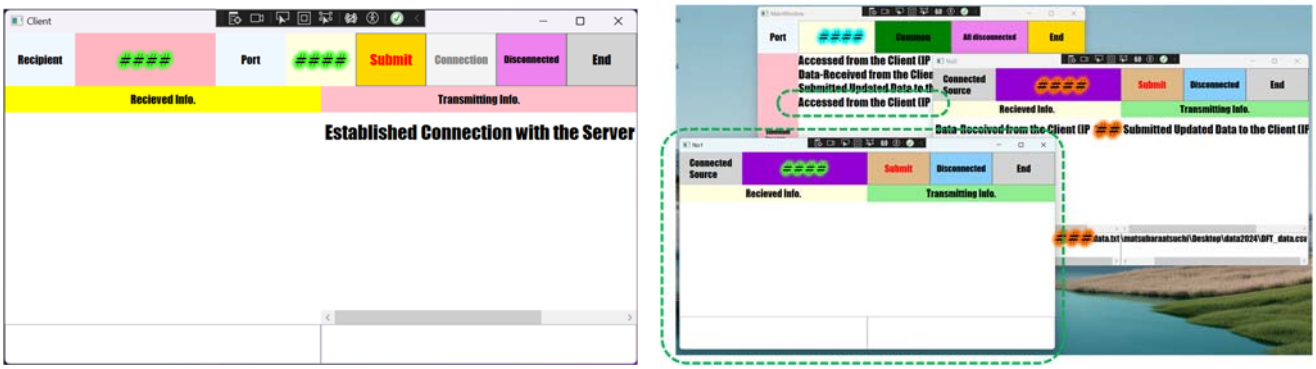
Figure 9. Transmitting the file B to Client 1

"Area4-2") can safely open.

Oppositely, the result of transmitting the file B to Client 1 is shown in Figure 9. Client 1 / Server displays the received message "**mS2**" / "**mC2**" in "Area 1-1" / "Area 5-1". As shown in the figure (a), opening safely the received file B (in "Area 1-2") on Client 1 side leads to the success of the transmission.

In the above situation, the system obtains the result shown in Figure 10 when Server forms the novelty communication line with Client 2. The figure shows the same achievement of establishing as Client 1: Server / Client2 displays the received "**mC0**" / "**mS0**" in "Area 3" / "Area 2-1", and the former generates the novelty Window class instance for the latter.

As the next task, Server transmits the file C to Client 1 and 2. The transmission protocol is as follows (refer to Figure 4): first, Server creates the Window for transmitting a shared file using the Button class instance "Common" on Server's

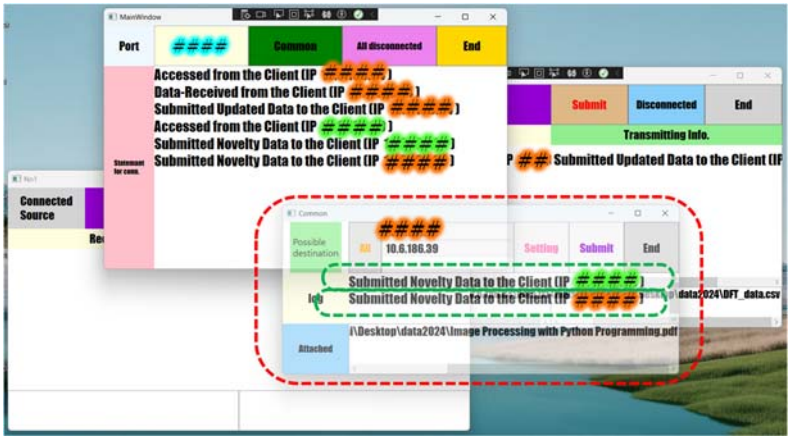


(a) Client 2 (b) Server

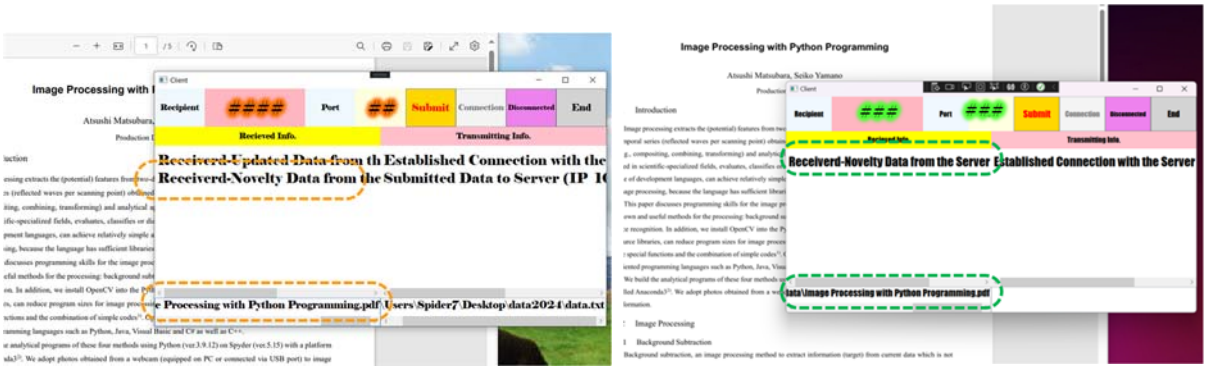
Figure 10. Establishing the Client 2- Server line

Window; second, the clicked event of the Button class instance “All” allows to select all clients as transmitting destination and that of the Button “Setting” does to decide them; third, the file C is set in the “Area 6-2” by drag and drop operations; final, Server transmits the file using the Button “Submit”.

The successful result is shown in Figure 11. It shows that All clients receive the message “**mS3**” and the file whereas the server does the “**mC3**”. Furthermore, the figure (a) catches the shot just before terminating the Window class instance for transmitting. It is due to (the fact) that the instance closes automatically using a timer (DispatcherTimer class instance) after transmitting a file.

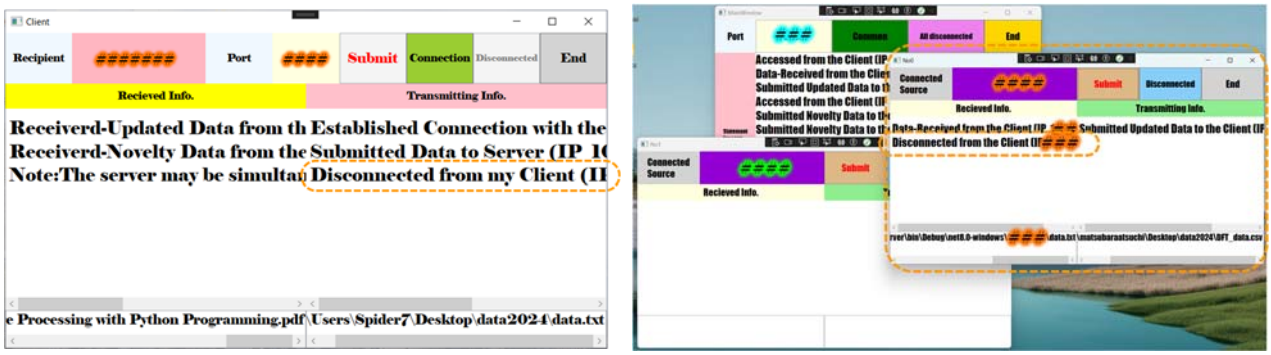


(a) Server



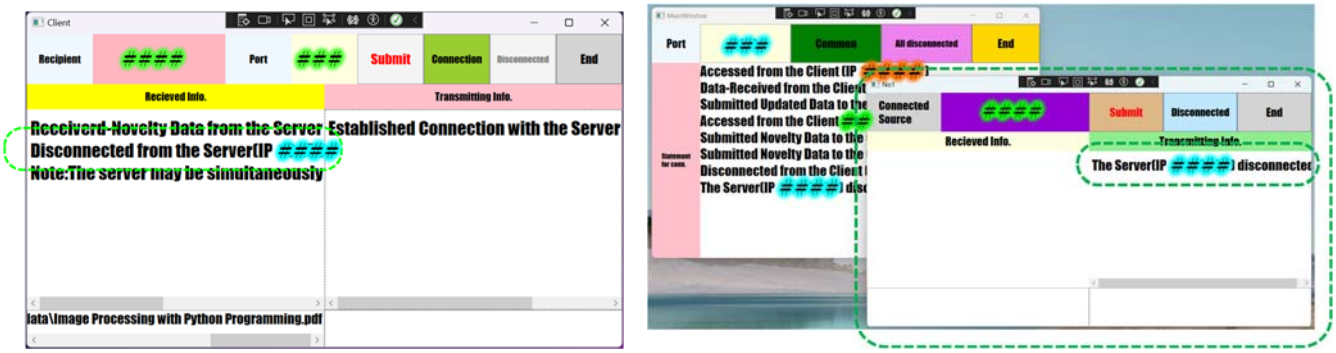
(b) Client 1 (c) Client 2

Figure 11. Transmitting the file C to Client 1 and 2



(a) Client 1 (b) Server

Figure 12. Terminating Server-Client 1 line from Client1



(a) Client 2 (b) Server

Figure 13. Terminating Server-Client 2 line from Server

Finally, the termination of the two communication lines is described below. First, Figure 12 shows terminating the Server-Client 1 line from Client1. Server / Client 1 receives the message "**mC4**" / "**mS4**". Server will soon close the Window class instance for Client1 because the DispatcherTimer class instance is running. Next, Figure 13 shows terminating the Server-Client 2 line from Server. Server / Client 2 receives the message "**mC5**" / "**mS5**". Server will similarly close the Window for Client2. Server thus displays the message that does not have the connected clients (Figure 14).

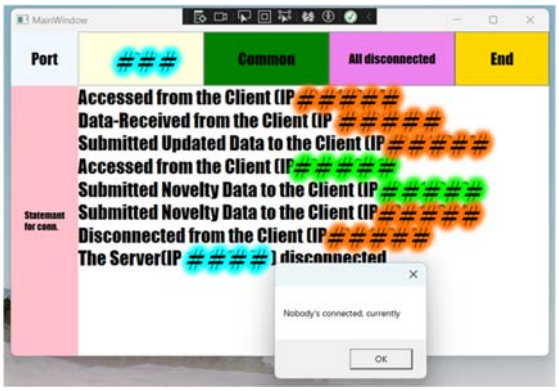


Figure 14. No clients connected

As supplementary information, the system appropriately switches "IsEnabled" property of each Button class instance to "True" or "False" depending on the events. This means to avoid occurring system errors associated with operation of Button class instances (see Tables 1 and 2)

	Submit	Connection	Disconnected	End
After starting the application	**False**	**True**	**False**	**True**
After establishing tne communication	**True**	**False**	**True**	**True**
After terminating the communication line	**False**	**True**	**False**	**True**

Table 1. Enabled of each Button class instance (client's side)

	Server-customized Window			Window for client		
	Common	All Disconnected	End	Submit	Disconnected	End
After starting the application	**False**	**False**	**True**			
After establishing tne communication	**True**	**True**	**True**	**True**	**True**	**True**
After terminating the communication line	**False**	**False**	**True**			

	The common or novelty file transmission Window			
	All	Setteing	Submit	End
After starting this Window	**True**	**False**	**False**	**True**
After selecting the clients	**True**	**True**	**False**	**True**
After deciding the clients	**True**	**False**	**True**	**True**
After transmitting data	**True**	**False**	**False**	**True**

Table 2. Enabled of each Button class instance (server's side)

5 Conclusion

The reported system could be suitable for small-scale communication one that a client provides or shares research and educational resources such as programming codes, analysis results, image or text data and administrative documents with server on communication line. The system actually controls the number of clients and the size of a transmittable file: the former number depends on members of a lab, a project group and a department of a few dozen people; the latter size is about a few Mbytes. Thus, the system will be useful not only as simple chat application but also, for instance, a server analysis system for the reason that it is possible to centralize management and operation of analysis resources including programs by a server.

In contrast, the system, notably, the server does not limit the number of incoming clients because it uses normal type of Socket.Listen() method without the maximum number (of clients) specified as the argument. The more threads the server has, the more likely it is to lead to significant consumption of the system resources (that is, overloading the operating system). It thus would be a useful idea to improve the system to reuse threads such as thread pool.

Additionally, the system has a lot of unicast architectures establishing independently a one to one communication line although it can connect to multiple clients. Transmitting a shared information on these architectures could lead to increase network load (that is, the processing size or the complexity of data communication). Therefore, it seems that it would be beneficial to adopt the User Datagram Protocol (UDP) communication method, which does not need to establish communication line prior to transferring data between two nodes, such as multicast and broadcast architectures.

The author hopes to improve smartly the system to facilitate communication processing and to enrich its functionality as one of chat applications in the future working. This paper has been polished with the advice and the comments of Dr. Seiji Nishifuji and Dr. Shota Nakashima, who are respectively the associate professor and the lecturer in Graduate School of Sciences and Technology for Innovation, Yamaguchi University. The author is deeply grateful to them.

References

1) System.Net Namespace, Microsoft Ignite (2024)
https://learn.microsoft.com/en-us/dotnet/api/system.net?view=net-9.0

2) D. B. Makofske, M. J. Donahoo, K. L. Calvert: TCP/IP Sockets in C#: Practical Guide for Programmers, Morgan Kaufmann, pp.37-57, 117- 131(2004)