CinsApMaSys

Fudayl Ikbal Cavus – Mahmut Sacit Meydanal

Abstract —**This technical report presents the design and implementation of CinsApMaSys, a system for managing communication and information sharing among the residents of CinsApartment. The system consists of a server located in the IT room of the apartment, which collects weather and currency exchange information from external sources, as well as information from a card reader that controls the outer gate of the apartment. The system also includes a GUI-based interface for each computer and server, divided into four main sections: weather information, currency exchange information, card reader information, and a chat section for communication among the residents.**

The system is built using asynchronous sockets and is designed to allow easy communication and information sharing among the residents of the apartment

Index Terms — AsyncSocket, Async Programming, ApartmentManagement

I. Introduction

The efficient management of communication and information sharing among residents in an apartment complex is essential for maintaining a harmonious living environment. In this technical report, we present the design and implementation of CinsApMaSys, a system for managing communication and information sharing among the residents of CinsApartment. The system is built using asynchronous sockets, allowing for easy communication and information sharing among the residents.

The system consists of a server, that will be located in the IT room of the apartment, which collects weather and currency exchange information from external sources, as well as information from a card reader that controls the outer gate of the apartment. This information is then made available to all residents through a GUI-based interface that is accessible on each computer and server in the apartment.

The interface is divided into four main sections: weather information, currency exchange information, card reader information, and a chat section for communication among the residents.

The weather information section is updated in real-time from the server, providing residents with the current weather conditions and forecast for the area. The currency exchange information section is also updated in real-time, providing residents with the current exchange rates for various currencies. The card reader information section displays information about the use of the outer gate, such as the date and time when it was unlocked and by whom. The chat section allows residents to communicate with each other in real-time, promoting a sense of community among the residents.

In this technical report, we will describe the design and implementation of CinsApMaSys in detail, including the use of asynchronous sockets for communication, the GUI-based interface, and the various features of the system.

II. Design of The Application

Design of the application consists of 2 sections, which are:

a. Login Screen Design
b. Client Panel Screen Design

Before starting to design, we thought about the application features, what will user do, what will we need to create system that focuses on user-experience (UX) and provides layout to use all the required features.

After this analysis, we come up with a few musts for Login Screen. Family Name and the password that will provided by IT Room Manager. As we developed further, our ideas have changed too. In development process server was sometimes crashing so we decided to add `Toolbar Status Indicator` for connection status. Which also makes it possible to reconnect by clicking the bar.

As a result, login screen should have, Family Name input, password input and connection status indicator. Additionally, we added placeholders to input boxes, since user may need it. (Example user/pass combination is used in placeholder: Erdem, erdem) These were the things, that is required to use application and for user-experience.

Final design of Login Screen:

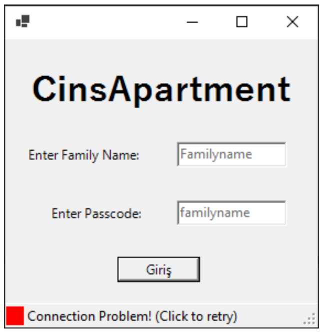


Figure 1 Final Product with `Connection Problem Status Indicator`

Figure 2 `Connecting…` Status Indicator

Figure 3 `Connected` Status Indicator

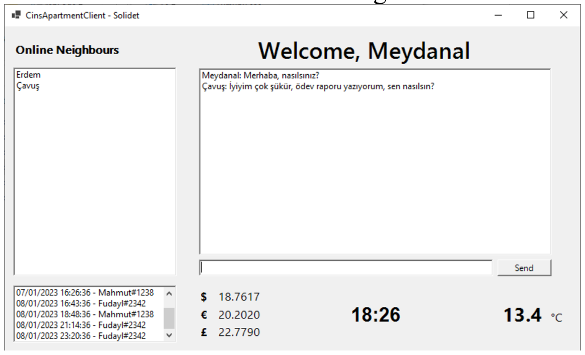
b. Client Panel Screen Design

Every requirement of the application is clearly given to us. So, we used to design applications and put every section together, then we made all the alignments inside of designing tool, since it's way faster to visualize product. And after each revision we checked whether final product seems fine or provides good user-experience or not.

Crucial part of the application was missing. If we'll have Chat feature in our application, we should know who is online, since that would be too boring to send messages to the empty space, without knowing if there's a receiver.

Client panel screen will be the most used screen of the application, so it has to make user feel like an application, which user uses for its entire computer experience. For instance, we're so familiar to see date-time in Windows' toolbar, which is at bottom of the screen, or we tend to see user list in chat applications side by side (Whatsapp Web, Facebook Messenger, usually user-list on the left), even simple exchange and weather information's position is important. We created our layout with the help of all those thoughts.

Result of Client Panel Screen Design:



Erdem
Çavuş

Meydanal: Merhaba, nasılsınız?
Çavuş: İyiyim çok şükür, ödev raporu yazıyorum, sen nasılsın?

```
07/01/2023 16:26:36 - Mahmut#1238
08/01/2023 16:43:36 - Fudayl#2342
08/01/2023 18:48:36 - Mahmut#1238
08/01/2023 21:14:36 - Fudayl#2342
08/01/2023 23:20:36 - Fudayl#2342
```

$	18.7617
€	20.2020
£	22.7790

18:26

13.4 °C

Figure 4 `Client Panel Screen` with 3 total online members

III. Implementation of the Application

As we have tried to follow the core programming principals, we believe we made it as clear as possible.

We can split our implementations to those key concepts:

a. Login System Implementation

We believed that we had to have a login system in order to preserve the 8-family limit and increase our app's believability. To achieve this, we had to make our server and client work together in cooperation.

Our system sent its credentials as package to server and wait our server's answer before deciding to whether allow user to our application's main section or not. IT Manager can increase the number of family count that can use the application. We provided constant `Credentials.json` file in our project. There's dictionary, "Familyname" as key and "Password" as value.

We store different password combinations for each family, the reason of that even though family limit is 8 we don't want to limit it to 8 clients. Users in same family should be able to use and connect with their neighbours when needed.

After all, if a pass is granted via server, which is nothing but another package sent by server, our user logs in and let other sockets know he is online with distributing this info through our server.

b. API Calls Implementation

To make information on our clients overlay we had to make frequent calls, but it comes with a downside, it could make server act slower upon more needed areas, so we had to hit a sweet spot between the most up to date information and performance. After testing between various numbers, we decided to do API calls every 10 seconds and send new data to the connected clients.

Since API Calls aren't fast services, and our server should be async, we implemented our API calling function as `Task`, with the help of that API requests aren't blocking our main thread.

There was an edge problem, in this solution. That is the problem of sending data to new joined client. If the user is joined at the beginning of the interval, it will stay without weather or exchange data for approx. 10 seconds. So, we started to store last called API result in memory. Whenever there's a new user, we're sending `lastWeatherInformationPacket` and `lastExchangeInformationPacket` to the client.

(Note: The way we send packets will be described in later sections)

c. Chat System Implementation

Chat section was one of the first systems we have implemented right after having the prototype of GUI. As we have developed quite projects together, we had a experience with developing chat systems for our applications so we are quite comfortable and happy with the current state of our chat implementation. We are also listening for 'Enter' keypress to send the message on event to make UX better. As users are quite used to have it already implemented It, if we wouldn't have it implemented their UX would take a drastic hit as our program wouldn't behave the way as they would normally expect.

As logical part of this implementation, one line of code does the job with the help of our Client.cs and SocketHelper.cs. Later in the document `Send` and `Broadcast` methods of the client will be mentioned.

Whenever server receives ChatMessagePacket, it's broadcasting it over our sender client, so everyone gets the message except sender. Since the sender should be able to see sent message even with no internet or server connection, we keep it outside of our broadcast. Which might help us in our future implementations to add message status as `Pending`.

If client messages would send to everyone including sender itself, our sender might feel delay on its screen which might be frustrating. So, we preferred to implement this way.

d. Online Neighbours Implementation

To see which users are available, we need to let our servers know & remember who is online and have it stored until client says it is going to log off. So built on those principles we made clients let server acknowledge that he is logging in, assuming he provided true credentials, and we stored it on server. If someone logs in at any time, he will receive online users list from server in order to synchronize its data and store it in its own local data. Afterwards If someone is logged off, server would send its name and clients will remove it its own data and likewise if someone logs in, he gets the name of the person who logged in and he would add it to its own local data.

This is done by the help of "Dictionary<string, Client> clients". We track every client and their information on server-side memory. Then notify new user with the help of that data.

e. Card Log Reader

We used mock data to pretend as if some families entered using cards. We could have implemented fake log generator, but we didn't want to put effort on that as it is not one of the main concepts of our homework. We receive and send logs through sockets without problem, and we believed that is good enough.

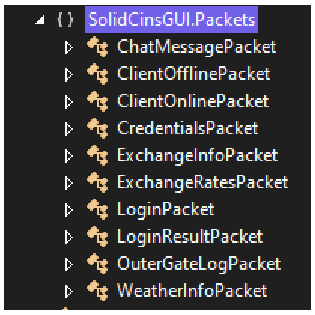
IV. Helper Classes

We have two strong helpers as foundations throughout our application.

a. Socket Helper

This part doesn't exactly affect end user, yet this is the most crucial part for our development process. We had to do a lot of unnecessary and repetitive work for processing data to make it suitable for receiving and sending. We streamlined this process as much as possible, so we almost were like using our own socket library as these helpers did the jobs every time, we call them.

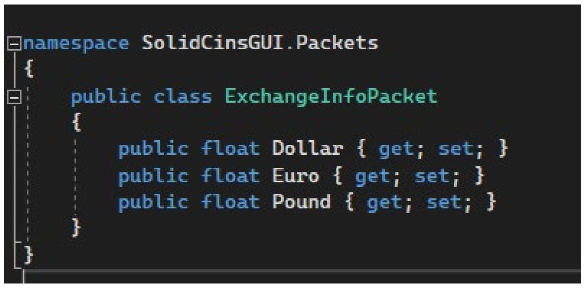
To create consistent data transfer, we thought that our data packets should pre-defined class templates. Since if we would use random string manipulation that would be hard for any other developer than us. Of course, to use those packets they should be defined both server and client side with same namespace.



```
◢ { } SolidCinsGUI.Packets
    ▷ ChatMessagePacket
    ▷ ClientOfflinePacket
    ▷ ClientOnlinePacket
    ▷ CredentialsPacket
    ▷ ExchangeInfoPacket
    ▷ ExchangeRatesPacket
    ▷ LoginPacket
    ▷ LoginResultPacket
    ▷ OuterGateLogPacket
    ▷ WeatherInfoPacket
```

Figure 5 All used packets

Whenever we need additional information it's easy to create a packet. And usage (send/receive) of it.



```
namespace SolidCinsGUI.Packets
{
    public class ExchangeInfoPacket
    {
        public float Dollar { get; set; }
        public float Euro { get; set; }
        public float Pound { get; set; }
    }
}
```

Figure 6 Example packet definition

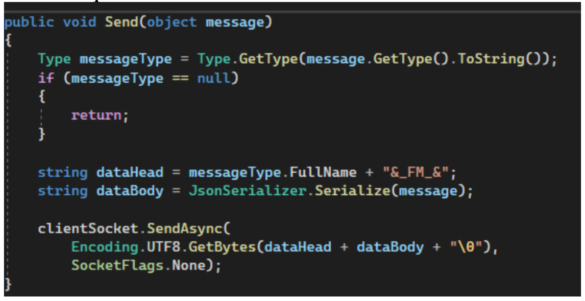
i) Send Method Implementation

We are using types to determine which event a message belongs to with the help of pre-defined packets.

In code part, Send is method that takes `object message` as parameter. We used built-in `Type` class to get type of provided "object" and used it in data-head section. And the data-body is value of object fields but it's in serialized version with the help of "JsonSerializer". After the concatenation data-body and data-head with our special delimiter "&_FM_&". Message is ready to send with "Socket.SendAsync" method.

Additionally, to understand where the data ends when receiving, we used "\0" line ending character at the end of our encoded message.

Final implementation for send method:



```
public void Send(object message)
{
    Type messageType = Type.GetType(message.GetType().ToString());
    if (messageType == null)
    {
        return;
    }

    string dataHead = messageType.FullName + "&_FM_&";
    string dataBody = JsonSerializer.Serialize(message);

    clientSocket.SendAsync(
        Encoding.UTF8.GetBytes(dataHead + dataBody + "\0"),
        SocketFlags.None);
}
```

Figure 7 Generic Send Method implementation

Whenever we need to use this method only thing we do is: "`client.Send(new XYZPacket())`"
It handles everything itself.

ii) Broadcast Method Implementation

Our Send method is mainly used in here only difference is, it's looping through all online clients and sends the packet to everyone except the object itself.

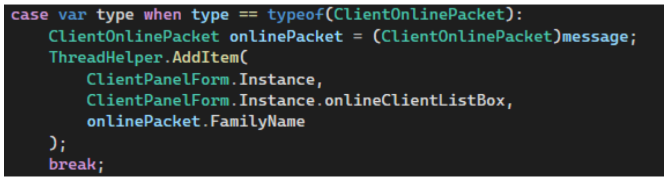
iii) Receive Method Implementation

This part also automates receiving part without losing any data along the way. 'ReceiveAsync' is used while getting data and then, since we know the data sections, 'Receive' method separates the parts and send them to appropriate methods. To identify type, data-head is sent to built-in method "Type.GetType", and to consume data, deserialized data-body is sent to "HandlePacket" method.

iv) Handle Packet Method Implementation

This one is equivalent of the 'socket.on('event')` function of most used socket library which is socket.io. We are listening to 'packages' we receive and act upon according to functions we define on our switch cases here. This is lifeline of our program as we receive and act upon datas here.

Since GUI Application has packets as well, we know what's the type of packet after using "Type.GetType" then simple switch-case with type comparison helps us to do an action based on the event.

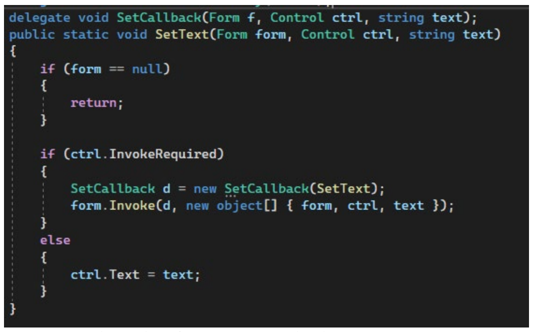


```
case var type when type == typeof(ClientOnlinePacket):
    ClientOnlinePacket onlinePacket = (ClientOnlinePacket)message;
    ThreadHelper.AddItem(
        ClientPanelForm.Instance,
        ClientPanelForm.Instance.onlineClientListBox,
        onlinePacket.FamilyName
    );
    break;
```

Figure 8 Example case (event) handler

b. Thread-Helper

In Form Application .NET didn't allow us to manipulate the content directly in cross-threads. This class helped us a lot along the way, because our main vision in this application is to create Async Application. Basically, the thing that this class does is whenever we want to set a value or add an item to list, it's controlling whether control has true value for "InvokeRequired" or not. Which is crucial feature to use with Threads. If it's required, we're invoking it to appropriately update the value. But if not, we're setting the value as we usually do.



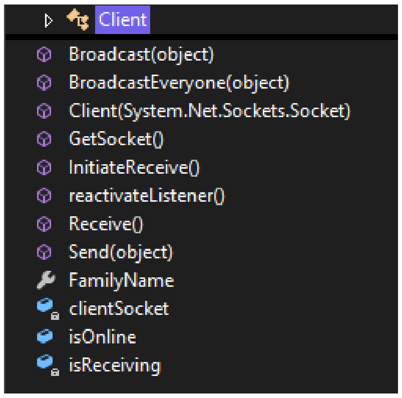
```
delegate void SetCallback(Form f, Control ctrl, string text);
public static void SetText(Form form, Control ctrl, string text)
{
    if (form == null)
    {
        return;
    }

    if (ctrl.InvokeRequired)
    {
        SetCallback d = new SetCallback(SetText);
        form.Invoke(d, new object[] { form, ctrl, text });
    }
    else
    {
        ctrl.Text = text;
    }
}
```

Figure 9 Thread-safe SetText method

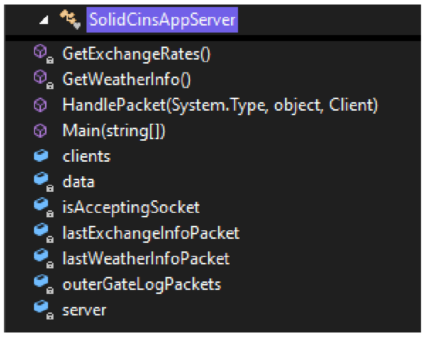
V. Object Models

We prepared our object models as carefully as possible as if we are building a large project to avoid having confusions and have clear & swift development process. We prepared our object models before everything else, so we continue building other things on top of our object models.



```
Client
  Broadcast(object)
  BroadcastEveryone(object)
  Client(System.Net.Sockets.Socket)
  GetSocket()
  InitiateReceive()
  reactivateListener()
  Receive()
  Send(object)
  FamilyName
  clientSocket
  isOnline
  isReceiving
```

Figure 10 Object Model of Our Client



```
SolidCinsAppServer
  GetExchangeRates()
  GetWeatherInfo()
  HandlePacket(System.Type, object, Client)
  Main(string[])
  clients
  data
  isAcceptingSocket
  lastExchangeInfoPacket
  lastWeatherInfoPacket
  outerGateLogPackets
  server
```

Figure 11 Object Model of Our Server

VI. Conclusion

In this technical report, we have presented the design and implementation of CinsApMaSys, a system for managing communication and information sharing among the residents of CinsApartment. The system is built using asynchronous sockets, allowing for easy communication and information sharing among the residents. The system consists of a server located in the IT room of the apartment, which collects weather and currency exchange information from external sources, as well as information from a card reader that controls the outer gate of the apartment.

The system also includes a GUI-based interface for each computer and server, divided into four main sections: weather information, currency exchange information, card reader information, and a chat section for communication among the residents. These features provide residents with real-time information about the environment and the ability to communicate with each other in a friendly and efficient way.

Overall, CinsApMaSys has been designed to improve the living experience for residents by providing them with easy access to important information and facilitating communication among them. It can serve as a model for similar systems in other apartment complexes. The use of asynchronous sockets and a GUI-based interface makes the system user-friendly and easy to use. With the help of this system the residents of CinsApartment can keep themselves updated with the latest information, communicate with each other and overall, it will enhance their living experience in the apartment.

REFERENCES:

[1] https://learn.microsoft.com/tr-tr/dotnet/api/system.net.http.httpclient.sendasync?view=net-7.0

[2] https://learn.microsoft.com/en-us/dotnet/api/system.net.sockets.socket.receiveasync?view=net-7.0

API SOURCES:

[1] https://api.open-meteo.com/v1/forecast?latitude=38.41&longitude=27.13&current_weather=true&hourly=temperature_2m,relativehumidity_2m,windspeed_10m

[2] https://api.exchangerate-api.com/v4/latest/TRY

Fudayl Çavuş was born in Konak, İzmir, Turkey in 2002. He went to primary and secondary school in Izmir, Turkey. He had C1 level knowledge of German and at bachelor's degree, he focuses on programming and computer science.

Email: fudaylcavus@gmail.com
Student No: 200316042

Mahmut Sacit Meydanal was born in Gaziosmanpaşa, Istanbul in 2001. Currently studies Computer Engineering in Celal Bayar University.

Email:mahmutsacitmeydanal@gmail.com

Student No: 200316057