

Distributed System Project Report

Distributed Chat Application



efrei

PARIS PANTHÉON - ASSAS UNIVERSITÉ

Table des matières

I. Project description and requirements	3
II. Architecture Design/Theory	4
A) Authentication.....	4
B) Messaging.....	7
1. Private Message.....	7
2. Broadcast Message.....	7
C) Fault tolerance	9
III. Implementation	12
A) GitHub of the project	12
B) Send Broadcast Message.....	12
C) Receive Broadcast Message	14
D) Send Private Message.....	14
E) Receive Private Message.....	15
F) Fault tolerance	15
IV. Test	17

I. Project description and requirements

The application to implement is a peer-to-peer distributed chat application. With this application, the user should be able to send broadcast messages up to 9 different chatters (connected users) and he should be able to send private messages to a specific chatter.

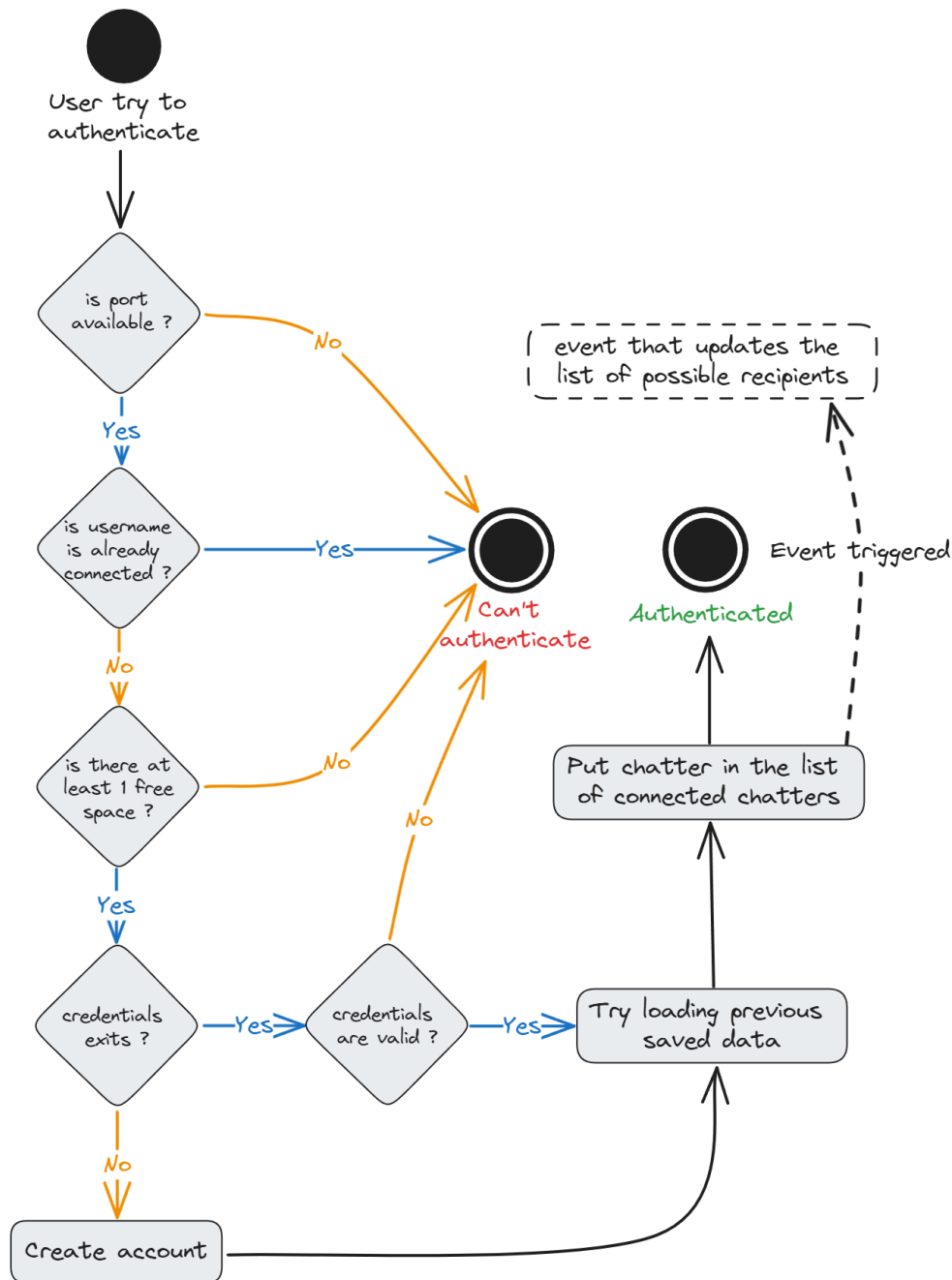
There is few constraints and assumptions made for this project. The maximum of connected chatters at the same time is 10. Communication between chatters should use UDP which means that **messages are unreliable and unordered** when sent (we must find solutions to address these problems). Broadcast messages should be received in a logical order (find the best algorithm to manage that). Private messages should be received in order (we need define which order).

In the whole report, chatters are defined as clients or users.

II. Architecture Design/Theory

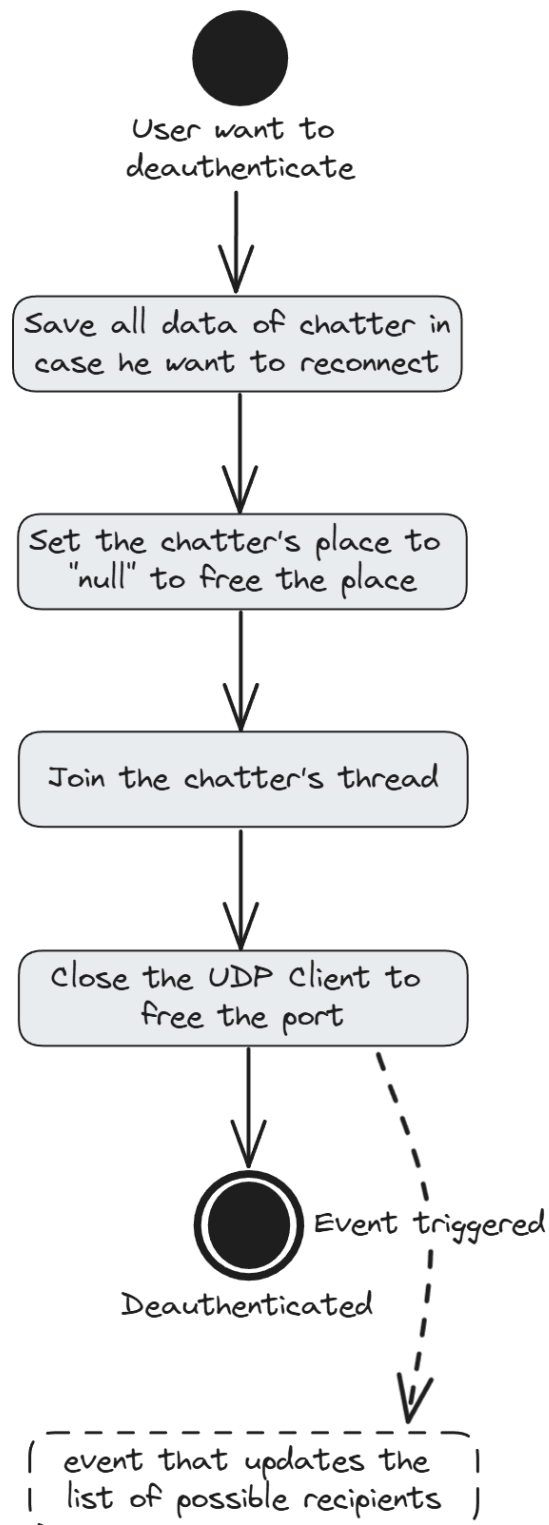
A) Authentication

The first step is to authenticate users who wish to use the application. If the user has never logged in, an account must be created. If an account already exists, you need to check the user's login details to make sure they match. However, these are not the only checks to be carried out. Below is an activity diagram showing the algorithmic logic to be applied for server authentication.

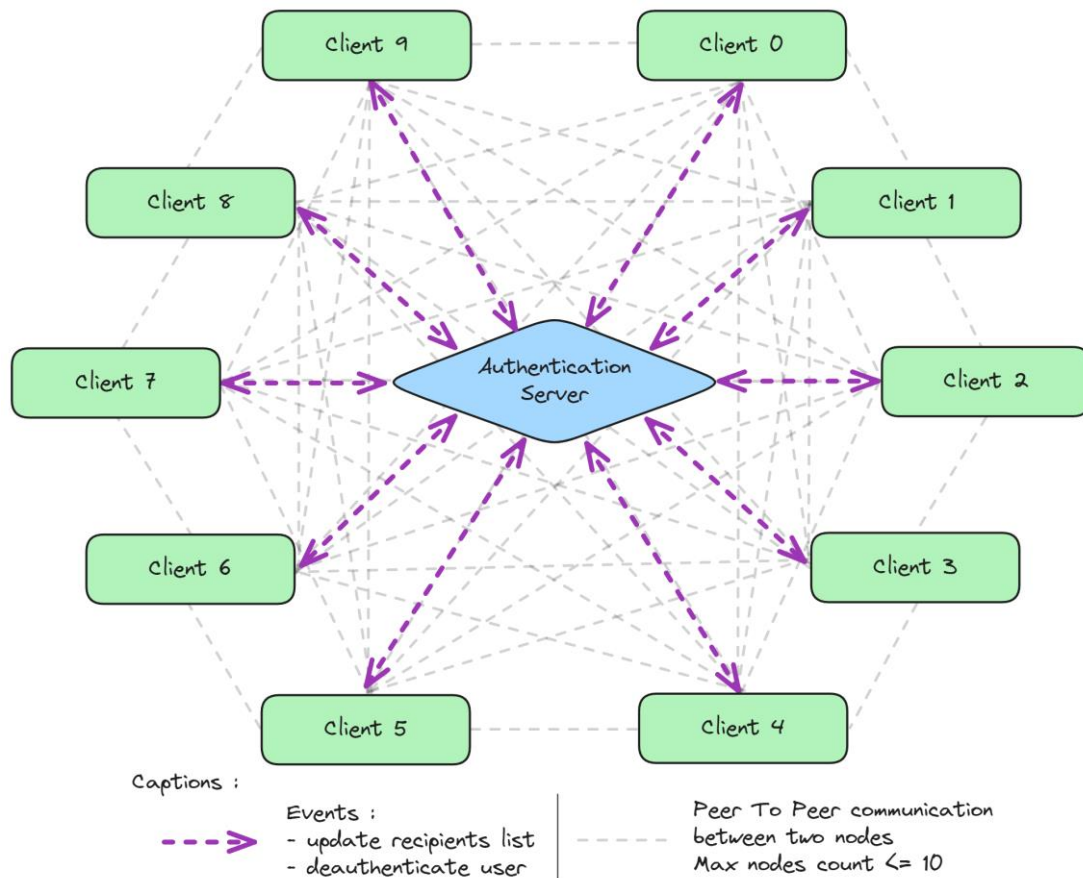


Scheme 1 : Activity Diagram for chatter's authentication on the application

In fact, if authentication is the most important part of the Authentication Server, the user might also want to log out without causing data loss and/or errors for other users and future users. Below is a diagram of the user logout activity.



Scheme 2 : Activity Diagram for chatter's deauthentication on the application



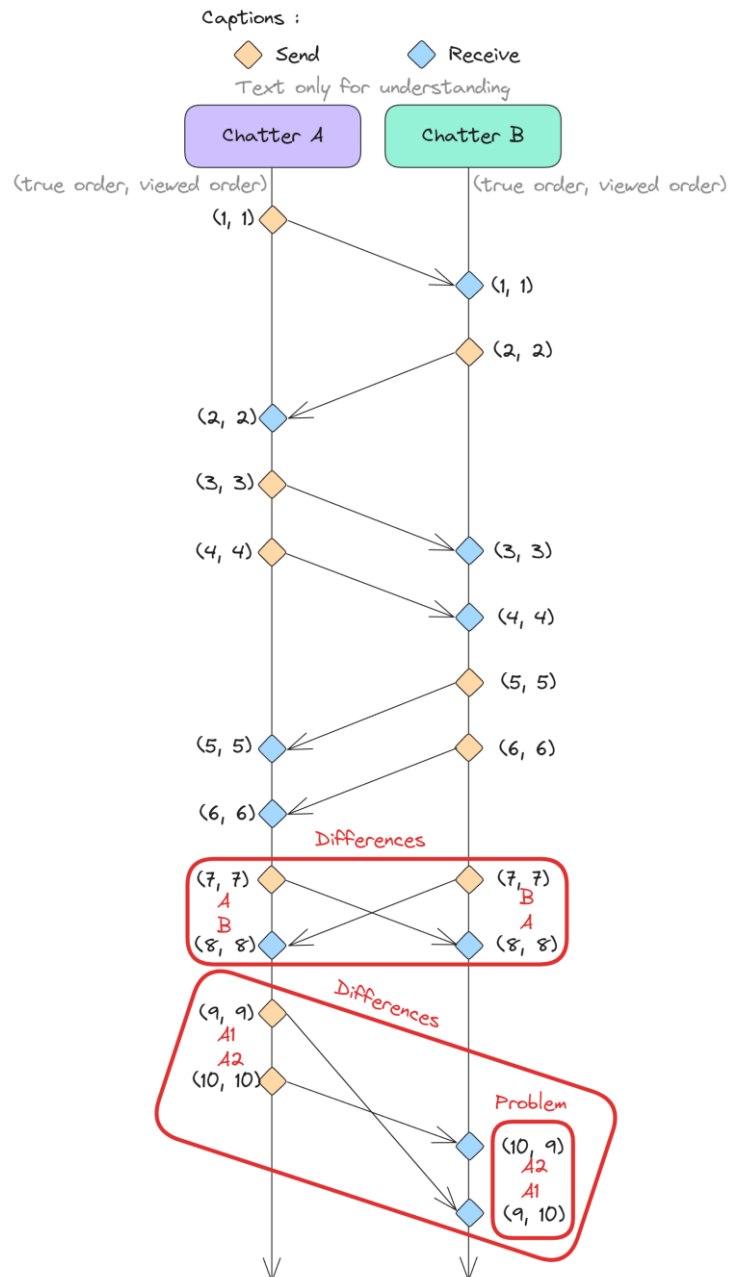
Scheme 3 : Diagram focusing on peer to peer with Authentication Server

The events that update the list of possible recipients on clients' UIs (User Interfaces) allow to each client to select the communication channel he wants to use. All chatters must use the Authentication to instantiate an UDP client that allows for peer-to-peer communication using UDP. Closing the client must automatically call the deauthentication function of the Authentication Server. Since the scope of the project is focusing on private messaging and broadcast messaging, the Authentication is a shared static class that will not communicate with clients using UDP.

B) Messaging

Once a Chatter has logged in through the authentication service, the user then has two options. They can either send private messages to another connected Chatter or send a broadcast message.

1. Private Message

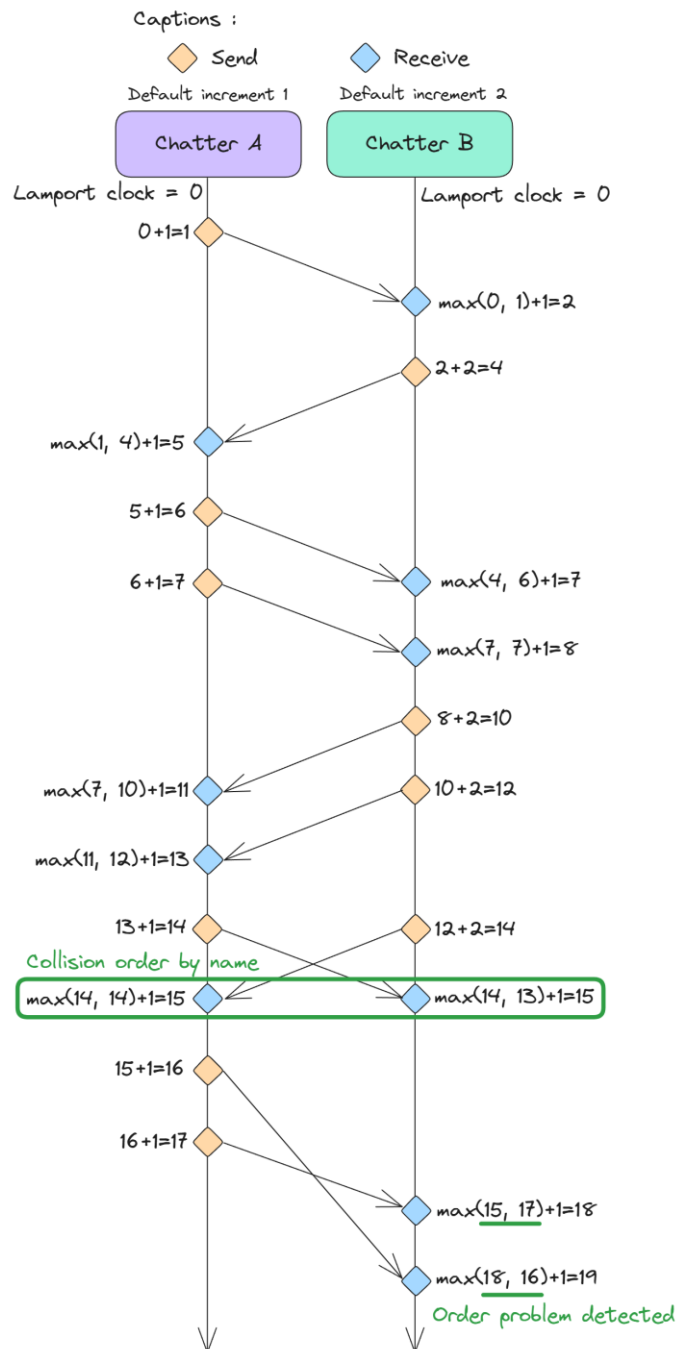


Scheme 4 : Naïve message sending using UDP (no order)

In the diagram above, the UDP messages are naively sent without any system to ensure their ordering. The displayed numbers are there to highlight the various issues that can arise. Two possible problems can be observed in this approach. If the messages are sent at the same time, the order of message display will be different

between Chatter A and Chatter B. Additionally, a lost message will not be detected. Finally, if the delay between two messages causes the first one to arrive second, it will be impossible to retrieve their correct order.

That is why I chose to implement the Lamport Clock Algorithm. It allows for managing situations where two messages are sent simultaneously. We can manage their display based on the chatter's username in alphabetical order. Additionally, if a message is not received, this can be detected thanks to the counters which will show a large offset.

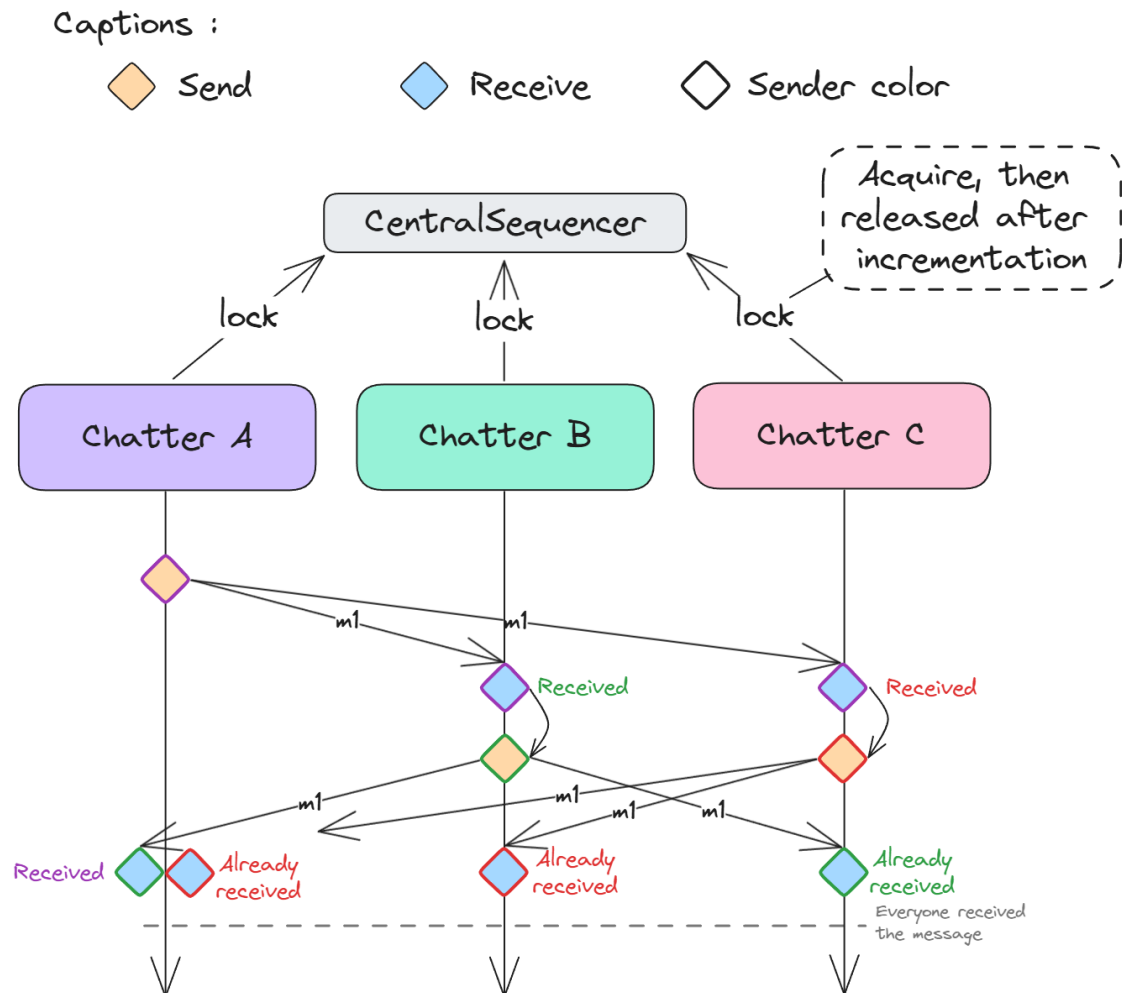


Scheme 5 : Lamport Clock System

2. Broadcast Message

For broadcasting messages, I had several algorithms to choose from. While some of my peers decided to implement complex systems, I kept in mind that the maximum number of nodes (chatters) is 10. The eager reliable broadcast is a resource-intensive algorithm with a complexity of $O(n^2)$. Although this could be problematic for an application with a large number of chatters, it is important to remember that we have a maximum of 10 users. Additionally, the second advantage of this system is that message loss is less likely as more users are connected.

However, this algorithm alone does not allow for ordering messages. That is why I coupled it with the Total Order Broadcast, which I slightly modified. Since all users must go through the authentication server, I decided that the "leader" would be the authentication server. Instead of serving as a relay to the other chatters, we use a shared variable acting as a central sequencer. This would ensure message order is always maintained by adding a lock on the sequencer whenever a new message is to be sent.



Scheme 6 : Eager Reliable Broadcast

Message order is kept thanks to the central sequencer. This allows to know if a message has been lost or if the message received is a duplicate.

Post presentation addition

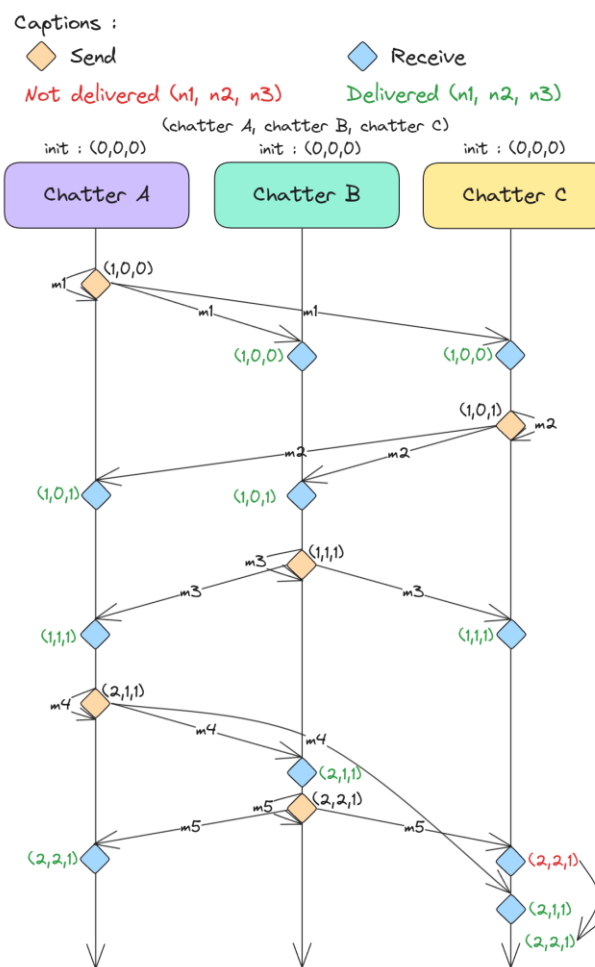
The use of a central sequencer and a "leader" node presents a real problem because it loses sight of the goal of having a distributed peer-to-peer chat due to the need for synchronization of all chatters by a "shared variable." Moreover, the use of FIFO broadcast does not guarantee causal order.

To address this problem, we can use the causal broadcast algorithm. This involves using vector clocks to ensure the correct delivery order of messages.

Each chatter has a vector of n clocks, with n being the number of chatters connected to the application. Before sending a message, the chatter increments his clock by 1 in the vector clock. It attaches the vector clock to the sent message.

Upon receiving a message, the message is redirected to a buffer to be processed. The idea is to compare the received message with its internal vector clock. The comparison to determine if the message can be delivered is as follows:

- The sender's clock in the vector clock of the received message must be less than or equal to the sender's clock+1 in the receiver's internal vector clock.
- For each other clock in the vector clock of the message, it must be less than or equal to its corresponding clock in the receiver's internal vector clock.



Scheme 7 : Causal Broadcast

C) Fault tolerance

As mentioned in the course and in the project topic, the simplest way to reduce the likelihood that a chatter does not receive the message intended for them is to send the same message multiple times.

To maintain good performance, I differentiated between private messages and broadcast messages in how the same message is sent multiple times. For private messages, I broadly estimated a message loss rate of 10%. Therefore, to ensure that the user's message is transmitted in the vast majority of cases, I send the user's message 10 times.

With a simple probability calculation, we find that the probability of the message being received at least once is 99,99999999% ($100 - 1 * \exp(-8)$). This leaves little room for bad luck.

Nevertheless, if this same logic is applied to broadcasting messages, we will seek to correlate this repetition of sending based on the number of connected chatters. Indeed, if only 2 chatters are connected, the probability that message delivery will be problematic is high when 10% of the time the message does not reach the recipient. In the situation where 10 chatters are connected, message loss is less of an issue because there are 10 messages sending that will be rebroadcast by our Eager Reliable Broadcast algorithm. Thus, I arrived at the following conclusion:

Number of repetitions = $10 - (\text{Number of connected chatters} - 1)$

This creates a reasonable balance and ensures that the message is delivered regardless of the number of users, while keeping the total number of messages sending reasonable.

III. Implementation

A) GitHub of the project

The complete source code as been published on a repository git. Here is the link to find the complete solution I developed in C# and using Winforms:

[NathanDelorme/DistributedChat \(github.com\)](https://github.com/NathanDelorme/DistributedChat)

B) Authentication and Deauthentication

```
public static string AuthenticationValidity(Chatter chatterToAuthenticate)
{
    string username = chatterToAuthenticate.GetUsername();
    string password = chatterToAuthenticate.GetPassword();
    int port = chatterToAuthenticate.GetPort();
    bool hasAnEmptySlot = false;

    foreach (Chatter? chatter in Chatters)
    {
        if (chatter == null)
        {
            hasAnEmptySlot = true;
            continue;
        }

        if (chatter.GetPort() == port)
            return $"Port {port} already used";

        if (chatter.GetUsername().ToLower() == username.ToLower() || chatter.GetUsername().ToLower() == "broadcast")
            return $"Username {username} already used";
    }

    if (!hasAnEmptySlot)
        return "Chat is full";

    if (ChattersAccounts.ContainsKey(username))
    {
        if (ChattersAccounts[username] != password)
            return "Wrong password";
    }

    return "AuthServer - OK";
}
```

```
public static string AuthenticateChatter(Chatter chatter)
{
    string errorMessage = AuthenticationValidity(chatter);
    if (errorMessage != "AuthServer - OK")
        return errorMessage;

    if (!ChattersAccounts.ContainsKey(chatter.GetUsername()))
    {
        ChattersAccounts.Add(chatter.GetUsername(), chatter.GetPassword());
    }

    AddChatter(chatter);
    return "AuthServer - OK";
}
```

```
private static void AddChatter(Chatter client)
{
    if (_savedLamportClocks.ContainsKey(client.GetUsername()))
    {
        client.SetMessageBuffers(_savedMessageBuffers[client.GetUsername()]);
        client.SetMessageHistory(_savedMessageHistory[client.GetUsername()]);

        client.SetLamportClocks(_savedLamportClocks[client.GetUsername()]);

        client.SetPrivateMessageHistory(_savedPrivateMessageHistory[client.GetUsername()]);
        client.SetRawPrivateReceivedMessage(_savedRawPrivateReceivedMessage[client.GetUsername()]);
    }

    for (int i = 0; i < Chatters.Length; i++)
    {
        if (Chatters[i] == null)
        {
            Chatters[i] = client;
            ChattersChanged?.Invoke();
            break;
        }
    }
}
```

```
private static void RemoveChatter(Chatter client)
{
    _savedMessageBuffers[client.GetUsername()] = client.GetMessageBuffers();
    _savedMessageHistory[client.GetUsername()] = client.GetMessageHistory();

    _savedLamportClocks[client.GetUsername()] = client.GetLamportClocks();

    _savedPrivateMessageHistory[client.GetUsername()] = client.GetPrivateMessageHistory();
    _savedRawPrivateReceivedMessage[client.GetUsername()] = client.GetRawPrivateReceivedMessage();

    for (int i = 0; i < Chatters.Length; i++)
    {
        if (Chatters[i] == client)
        {
            Chatters[i] = null;
            ChattersChanged?.Invoke();
            client.Close();
            break;
        }
    }
}
```

C) Send Broadcast Message

```
public async void SendBroadcastMessage(string messageContent)
{
    if (_udpClient == null)
        throw new Exception("Chatter is not started.");

    _sequencerSemaphore.WaitOne();
    int sequenceNumber = ++_centralSequencer;
    _sequencerSemaphore.Release();

    for (int i = 0; i < 10 - (AuthenticationServer.GetChatters().Count - 1); i++)
    {
        foreach (Chatter chatter in AuthenticationServer.GetChatters())
        {
            if (chatter.GetUsername() == _username)
                continue;

            Message message = new Message(true, sequenceNumber, this.GetUsername(), chatter.GetUsername(), messageContent);
            byte[] data = Encoding.UTF8.GetBytes(message.MsgToString());
            IPEndPoint targetEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), AuthenticationServer.GetChatterPort(chatter.GetUsername()));
            _udpClient.Send(data, data.Length, targetEndPoint);
            DataInOut?.Invoke(true, message);
            Debug.WriteLine($"Broadcast Message Sent: from {this.GetUsername()} to {chatter.GetUsername()} with sequence number {sequenceNumber}");

            _messageBuffers[message.GetSequenceNumber()] = message;
        }
    }
}
```

D) Receive Broadcast Message

```
if (message.IsBroadcast())
{
    if (!_messageBuffers.ContainsKey(message.GetSequenceNumber()))
    {
        _messageBuffers[message.GetSequenceNumber()] = message;

        foreach (Chatter chatter in AuthenticationServer.GetChatters())
        {
            if (chatter.GetUsername() == _username)
                continue;

            IPEndPoint targetEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), AuthenticationServer.GetChatterPort(chatter.GetUsername()));
            _udpClient.Send(data, data.Length, targetEndPoint);
            DataInOut?.Invoke(true, message);
        }
    }

    TryDisplayBroadcastMessages();
}
```

```
private void TryDisplayBroadcastMessages()
{
    while (_messageBuffers.ContainsKey(_nextSequenceNumber))
    {
        Message message = _messageBuffers[_nextSequenceNumber];
        _messageBuffers.Remove(_nextSequenceNumber);

        _messageHistory[_nextSequenceNumber] = message;
        MessageReceived?.Invoke(message);

        _nextSequenceNumber++;
    }
}
```

E) Send Private Message

```
public async void SendMessage(string recipient, string messageContent)
{
    if (_udpClient == null)
        throw new Exception("Chatter is not started.");

    if (!_lamportClocks.ContainsKey(recipient))
        _lamportClocks[recipient] = 0;

    if (string.Compare(this.GetUsername(), recipient) > 0)
        _lamportClocks[recipient] += 2;
    else
        _lamportClocks[recipient] += 1;

    int sequenceNumber = _lamportClocks[recipient];

    Message message = new Message(false, sequenceNumber, this.GetUsername(), recipient, messageContent);
    byte[] data = Encoding.UTF8.GetBytes(message.MsgToString());
    IPEndPoint targetEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), AuthenticationServer.GetChatterPort(recipient));
    for (int i = 0; i < 10; i++)
    {
        _udpClient.Send(data, data.Length, targetEndPoint);
        DataInOut?.Invoke(true, message);
    }

    Debug.WriteLine($"Private Message Sent: from {this.GetUsername()} to {recipient} with sequence number {sequenceNumber}");

    if (!_privateMessageHistory.ContainsKey(recipient))
        _privateMessageHistory[recipient] = new Dictionary<Tuple<int, string>, Message>();
    _privateMessageHistory[recipient][new Tuple<int, string>(sequenceNumber, this.GetUsername())] = message;

    TryDisplayMessages(recipient);
    RewriteChat?.Invoke();
}
```

F) Receive Private Message

```
else
{
    string sender = message.GetSender();

    if (!_lamportClocks.ContainsKey(sender))
        _lamportClocks[sender] = 0;

    if (!_rawPrivateReceivedMessage.ContainsKey(sender))
        _rawPrivateReceivedMessage[sender] = new Dictionary<int, Message>();

    if (_rawPrivateReceivedMessage[sender].ContainsKey(message.GetSequenceNumber()))
        continue;

    _lamportClocks[sender] = Math.Max(_lamportClocks[sender], message.GetSequenceNumber()) + 1;

    _rawPrivateReceivedMessage[sender][message.GetSequenceNumber()] = message;
    TryDisplayMessages(sender);
}

private void TryDisplayMessages(string conversationKey)
{
    if (!_rawPrivateReceivedMessage.ContainsKey(conversationKey))
        return;

    Dictionary<int, Message> messageQueue = _rawPrivateReceivedMessage[conversationKey];
    if (!_privateMessageHistory.ContainsKey(conversationKey))
    {
        _privateMessageHistory[conversationKey] = new Dictionary<Tuple<int, string>, Message>();
    }

    foreach (var message in messageQueue.Values)
    {
        if (!_privateMessageHistory[conversationKey].ContainsKey(new Tuple<int, string>(message.GetSequenceNumber(), message.GetSender())))
            _privateMessageHistory[conversationKey][new Tuple<int, string>(message.GetSequenceNumber(), message.GetSender())] = message;
    }

    RewriteChat?.Invoke();
}
```

G) Fault tolerance

Simulate random loss when receiving a message (private or broadcast):

```
// random loss
int random = new Random().Next(1, 100);

if (random <= 10)
{
    Debug.WriteLine($"Message Loss: from {message.GetSender()} to {message.GetRecipient()} with sequence number {message.GetSequenceNumber()}");
    continue;
}
```

Simulate sending delay:

```
// add random delay
int rndDelay = new Random().Next(500, 2000);
Debug.WriteLine($"Broadcast Delay : {rndDelay}");
await Task.Delay(rndDelay);
```

Send 10 times the same message for private messaging:

```
for (int i = 0; i < 10; i++)
{
    _udpClient.Send(data, data.Length, targetEndPoint);
    DataInOut?.Invoke(true, message);
}
```

Send (10 – (number connected user - 1)) times the same message for each chatter:

```
for (int i = 0; i < 10 - (AuthenticationServer.GetChatters().Count - 1); i++)
{
    foreach (Chatter chatter in AuthenticationServer.GetChatters())
    {
        if (chatter.GetUsername() == _username)
            continue;

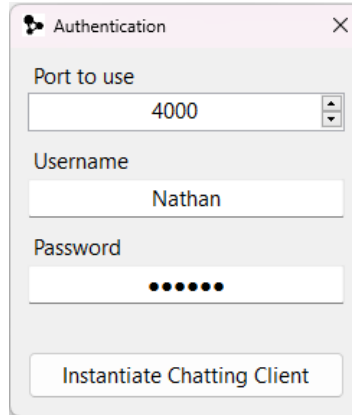
        Message message = new Message(true, sequenceNumber, this.GetUsername(), chatter.GetUsername(), messageContent);
        byte[] data = Encoding.UTF8.GetBytes(message.MsgToString());
        IPEndPoint targetEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), AuthenticationServer.GetChatterPort(chatter.GetUsername()));
        _udpClient.Send(data, data.Length, targetEndPoint);
        DataInOut?.Invoke(true, message);
        Debug.WriteLine($"Broadcast Message Sent: from {this.GetUsername()} to {chatter.GetUsername()} with sequence number {sequenceNumber}");

        _messageBuffers[message.GetSequenceNumber()] = message;
    }
}
```

There is a lot to show, but this was the most important parts of the program.

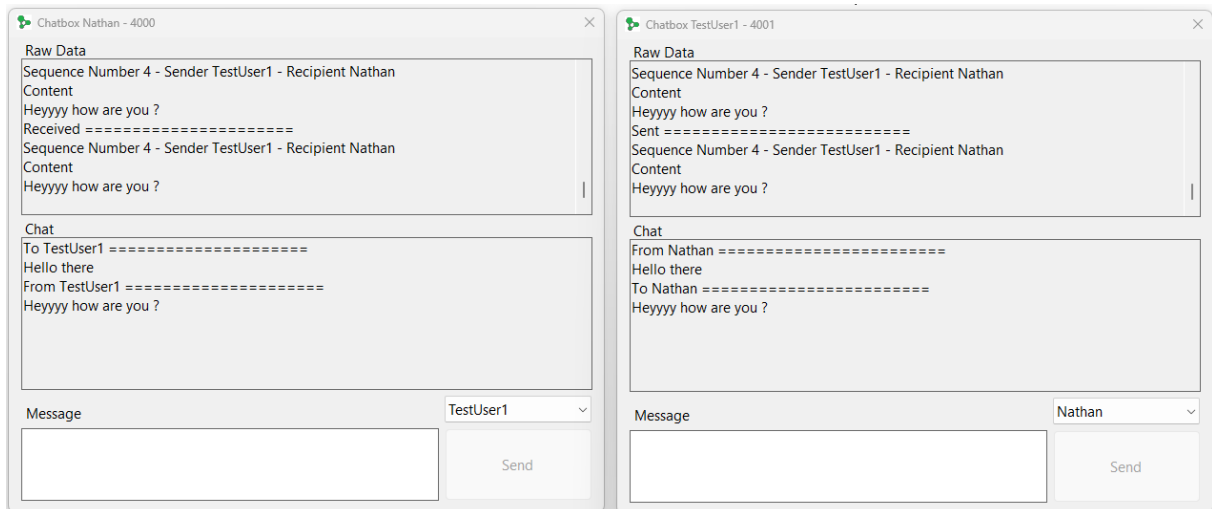
IV. Test

All test has been manually done through the interfaces:



An authentication dialog box titled "Authentication" with a close button (X) in the top right corner. It contains three input fields: "Port to use" with the value "4000", "Username" with the value "Nathan", and "Password" with masked characters "•••••". Below these fields is a button labeled "Instantiate Chatting Client".

Sending private message



Two side-by-side chat application windows. The left window is titled "Chatbox Nathan - 4000" and the right window is titled "Chatbox TestUser1 - 4001". Both windows have a "Raw Data" section at the top showing a log of messages. Below this is a "Chat" section with a text area for messages. At the bottom, there is a "Message" input field and a "Send" button. In the left window, the "Message" field is empty and the recipient dropdown is set to "TestUser1". In the right window, the "Message" field is empty and the recipient dropdown is set to "Nathan".

Chatbox Nathan - 4000

Raw Data

Sequence Number 4 - Sender TestUser1 - Recipient Nathan
Content
Heyyyy how are you ?
Received =====
Sequence Number 4 - Sender TestUser1 - Recipient Nathan
Content
Heyyyy how are you ?

Chat

To TestUser1 =====
Hello there
From TestUser1 =====
Heyyyy how are you ?

Message TestUser1

Send

Chatbox TestUser1 - 4001

Raw Data

Sequence Number 4 - Sender TestUser1 - Recipient Nathan
Content
Heyyyy how are you ?
Sent =====
Sequence Number 4 - Sender TestUser1 - Recipient Nathan
Content
Heyyyy how are you ?

Chat

From Nathan =====
Hello there
To Nathan =====
Heyyyy how are you ?

Message Nathan

Send

Sending broadcast messages

Chatbox Nathan - 4000

Raw Data

Broadcast - Sequence Number 2 - Sender TestUser1 - Recipient TestUser4
Content
B
Received =====
Broadcast - Sequence Number 2 - Sender TestUser1 - Recipient Nathan
Content
B

Chat

Broadcast from Nathan =====
A
Broadcast from TestUser1 =====
B

Message

broadcast

Send

Chatbox TestUser1 - 4001

Raw Data

Broadcast - Sequence Number 2 - Sender TestUser1 - Recipient TestUser4
Content
B
Received =====
Broadcast - Sequence Number 2 - Sender TestUser1 - Recipient TestUser4
Content
B

Chat

Broadcast from Nathan =====
A
Broadcast from TestUser1 =====
B

Message

broadcast

Send

Chatbox TestUser2 - 4002

Raw Data

Broadcast - Sequence Number 2 - Sender TestUser1 - Recipient Nathan
Content
B
Received =====
Broadcast - Sequence Number 2 - Sender TestUser1 - Recipient TestUser4
Content
B

Chat

Broadcast from Nathan =====
A
Broadcast from TestUser1 =====
B

Message

broadcast

Send

Chatbox TestUser4 - 4003

Raw Data

Broadcast - Sequence Number 2 - Sender TestUser1 - Recipient TestUser2
Content
B
Received =====
Broadcast - Sequence Number 2 - Sender TestUser1 - Recipient Nathan
Content
B

Chat

Broadcast from Nathan =====
A
Broadcast from TestUser1 =====
B

Message

broadcast

Send

Broadcast 3 messages at the same time (order kept !)

Chatbox Nathan - 4000

Raw Data

Received =====
Broadcast - Sequence Number 3 - Sender User4002 - Recipient Nathan
Content
c
Sent =====
Broadcast - Sequence Number 3 - Sender User4002 - Recipient Nathan
Content
c

Chat

Broadcast from Nathan =====
a
Broadcast from User4001 =====
b
Broadcast from User4002 =====
c

Message

broadcast

Send

Chatbox User4001 - 4001

Raw Data

Received =====
Broadcast - Sequence Number 3 - Sender User4002 - Recipient User4001
Content
c
Sent =====
Broadcast - Sequence Number 3 - Sender User4002 - Recipient User4001
Content
c

Chat

Broadcast from Nathan =====
a
Broadcast from User4001 =====
b
Broadcast from User4002 =====
c

Message

broadcast

Send

Chatbox User4002 - 4002

Raw Data

Sent =====
Broadcast - Sequence Number 3 - Sender User4002 - Recipient Nathan
Content
c
Broadcast - Sequence Number 3 - Sender User4002 - Recipient User4001
Content
c

Chat

Broadcast from Nathan =====
a
Broadcast from User4001 =====
b
Broadcast from User4002 =====
c

Message

broadcast

Send