University College of Northern Denmark

Technology and Business

Computer Science Academy Profession (AP) Degree

Class: DMAJ0916

Title: Technology and Programming Report – 3rd Semester 2017

Project participants (Group 2):

-Ralfs Zangis

-Andrei-Eugen Birta

-Hannes Heiskonen

-Stoycho Nenov Anastasov

Supervisors:

-Brian Hvarregaard

-Per Trosborg

Due date: 2017-December-18 (13:45 UTC+1)

Submission date: 2017-December-17

Birta Zanigis Heiskonen Nenov

1. Introduction

This document summarizes the collaboration of Group 2 for the programming and technology exam of the 3rd Semester. The group consists of 4 members of 4 different nationalities. Despite the major differences in our opinions (we agreed on certain rules and guidelines to follow, thoroughly elaborated in the accompanying document called group contract), we managed to harness the benefits of diverse ideas and identify multiple possible approaches to certain problems.

Although we followed a combination between SCRUM and Extreme Programming, as our development method, we decided to structure this report in as formal way as possible.

2. Problem Statement

The idea of this project is to create a web service that communicates with different types of clients, and allows users all over the world to gather round in topic-specialized chatrooms and facilitate passionate, real time discussions among small groups of people. On top of that, the service will provide several means of entertainment, to maintain a healthy community, such as: playing YouTube videos, creating playlists of your favorite YouTube videos or playing a game of Rock-Paper-Scissors. Some other goals that we intended to aim towards are: scalability, openness and transparency.

3. Development Process

As previously mentioned, we decided to follow a combination between Extreme Programming and SCRUM, as our development method, meaning that everything we managed to achieve in this project was done iteratively, each iteration having its own development phases.

3.1. Requirements

Our focus, for the project, is creating a web service that is capable of handling as many types of clients, as possible; whether they are running windows, mac, iOS, android or whatever other OS there is, should not make any difference, and all can use and consume the service. As a secondary aiming point, we wanted to make the system as fast as possible, in order to improve user experience. Security being on the 3rd place, simply because the service will not deal with any “vital” information from the user, such as personal ids or any payment details.

3.2. Analysis

a. Type of Service

When we took the decision on which API to follow, two main choices were on the table: Representational State Transfer (REST) and Simple Object Access Protocol (SOAP).

REST is not bound to a single protocol, which allows for greater extensibility of our software. Also, REST has relatively low degree of coupling between the client and the service allowing better maintainability for our service. Lastly, REST is stateless – messages exchanged between the server and the client, have all the necessary information for the message to be processed. However, having all this extra data in the messages can be considered redundant information (in the case where that some or all this information is not used) which can have a small negative impact on latency.

SOAP which is not architecture but a protocol (as it can be seen from the name), on the other hand, if well implemented could offer slightly better performance. However, the knowledge and experience required to use it properly would be increasing both development and maintenance costs and decrease scalability, because of the growing coupling.

All the above-mentioned reasons, determined us to create a RESTful service.

b. Framework

In terms of framework, we were split between WCF and WebAPI. But since our goal was first heterogeneity and second speed, we chose WCF, simply because it can have multiple types of bindings including TCP which can be faster, has binary data format, lower compatibility level; and HTTP, which has xml data format, has greater compatibility level. Furthermore, it offers great flexibility as it automatically selects the appropriate type of binding depending on what the user device is compatible with (as long as such type of a binding is configured).

The main reasons we even considered WebAPI was because it is easier to understand and learn, it is great for HTTP services, and because it can be faster than WCF. However, the difference in speed, at least for the number of users we anticipate, is so insignificant that it’s not worth the trade off with the compatibility WCF offers.

c. Database

There are two types of database technologies: Relational Databases, which are great at organizing and retrieving structured data; and Non-Relational Databases, which are best used when the data is inconsistent, incomplete or simply massive.

For our project, we have chosen to go with a Relational Database, simply because pros such as: strict ACID support, data normalization, supports joins, limitless indexing, and being one of the most common used technologies*, outweigh the cons of having a non-relational database, cons such as: working with joins can be difficult, slow mass updates, difficulty tracking schema changes.

As for engines, there are several choices that we considered, for a Relational Database, some of which are: Oracle Database, SQL Server and MySQL; and since all three of them were using dialects of the same language (SQL), it went down to the very basics when we took the decision on which to use.

As a final decision, we chose SQL Server 2014, because of the following: SQL Server executes and commits each instruction, unlike Oracle which requires explicit command to commit the changes; ease of use, since not only were we thought on how to use it, but also compared to Oracle, which give so many other settings and configurations that can be set to the wrong value; and performance.

d. Clients

d.1. Dedicated Client

For this type of client, we had to choose between several options, some of which were: WinForms, WPF and XAML.

Our choice was WinForms. We chose to go with it for many reasons. First, for us, design is not so important as functionality. Second, we did not want to spend time on a spike about new ways to create dedicated client. Third, it was made pretty clear that majority of the points which count towards project's grade come from the backend of the application. Finally, we were already familiar with a java version of WinForms, in eclipse, called "Window Builder".

Following a popular trend in programming, we decided to implement all the features in the dedicated client first, then, after finding errors and exceptions and fixing them, to "correctly" implement the features in the web client.

d.2. Web Client

Just as for the dedicated client, we had to choose between several options, as well. Some of our options were: MVC, web forms and web pages.

We have chosen ASP.NET MVC. At first, we eliminated Single Page Application and Web pages, because our application needs definitely more than 1 web page. Arguments against Web Pages were that we have never worked with web pages before and we did not want to do spike on it either. We decided to go with Web Forms, at first. It was easy to implement, user-friendly and works similarly to WinForms, which we used to create desktop application earlier. But after electing a new product owner, the decision to switch to MVC was made, simply because in the process of creating the web client, we became intrigued on the twists we have encountered so far. Also, MVC being more frequently used in actual businesses, than web forms only gave us more reasons to make the switch.

e. Middleware

e.1. Service-Database

In order to save data in our database of choice (SQL Server 2014), we needed something to help C# communicate with it, and for that we had taken into consideration two possibilities: ADO.NET and ADO.NET Entity Framework.

On one hand ADO.NET Entity Framework has better security, data encapsulation and helps reducing the redundant code, all of which were strong reasons to consider using it, however cons such as slightly slower performance and lack of optimization in terms of a flexible database model, determined us to go with ADO.NET, which even tough has bigger queries and is slightly harder to use, does not require, the program, to know the database model beforehand and not making spike on new technology, makes it perfect for an agile method of development.

e.2. Service-Clients

The communication between client and service is another important part of our project. We chose WCF as our framework, because of its ability to communicate using different bindings. We decided to take full advantage of it and use different types of bindings, for different things, depending on which binding fits best.

We had a look at all the bindings WCF supports and after a “walk and talk” meeting, we reduced their number to five, and ended up using two, in the end. The two bindings we chose are: “WsHttpBinding”, which we are using for actions invoked only by the client, and “WsDualHttpBinding” (with a custom-made configuration), which allows both end points to send requests to each other independently with a duplex contract. Although http bindings have reduced performance, compared to other types of bindings such as IPC or TCP, mainly because of their data format of choice, which is XML, compared to binary; it offers far greater compatibility with other types of devices, and its message level security, help improve the quality of our service.

Some other types bindings we have considered for our project were IPC, which got dismissed because it is used only when both service and client are on the same machine, which contradicts one of our main goals of this project; and TCP, which did not get accepted because they are mainly used for intranet communication, as opposed to http, which is used for communications over the internet.

f. Concurrency

In this project we have three places where our users will have to “fight” for a place:

- Join chat (as a single user)
- Join chat with group (either as part of a group or as the leader of the group)
- Join game

We decided to approach this problem using the pessimistic way of handling it. Why the pessimistic approach? Because we found it being the best way of preventing the lost updates problem.

The concurrency between users would happen like this: two users try to join chat where max number of places is 5, 4 of which being occupied. Both, user1 and user2 would see the chatroom having one available place and both would try to join it at the exact same time. Without handling this problem, both users would successfully join the chatroom, making a total of 6 users inside, even though the limit was set to 5. A similar type of issues appears when users try to join a chatroom with a group, the only difference being that the complexity of the problem would slightly increase, considering the fact that either all users in the group have to successfully join or none should be able to.

3.3. Design

a. Domain Model

Our domain model (Fig.1) was one of the few diagrams we created for the project. Much thought and time has been invested in its creation; and the following bullet points should answer any questions related to it.

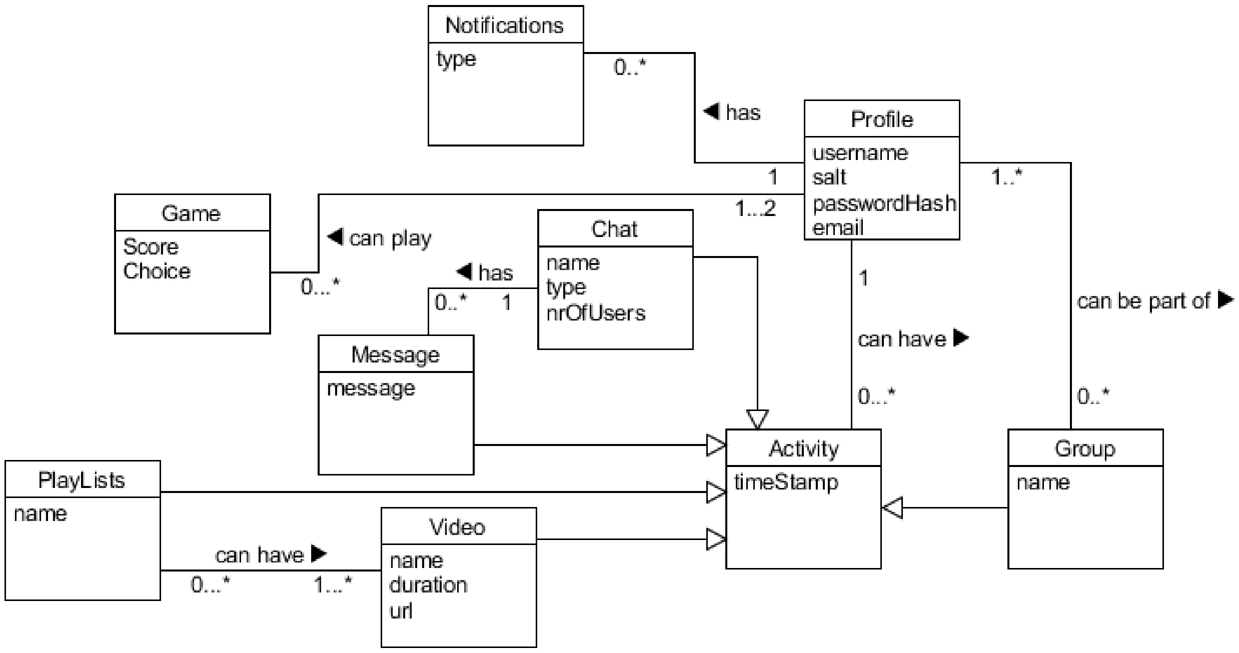


Figure 1

- Profile- all user related information.
- Activity- holds a timestamp and reference to user, to know which user, what and when, they created something.
- Group- depicts a premade team/fellowship of users that want to do things together.
- Chat- holds the location (chatroom) where messages will be displayed.

- Message- holds the actual text written by a user.
- Video- holds the reference of video to be played.
- Playlist- holds user created playlists.
- Game- represents the Rock-Paper-Scissors game, that users can play
- Notifications- holds user's notifications, for now, only being invited to a chat.

b. Database Model

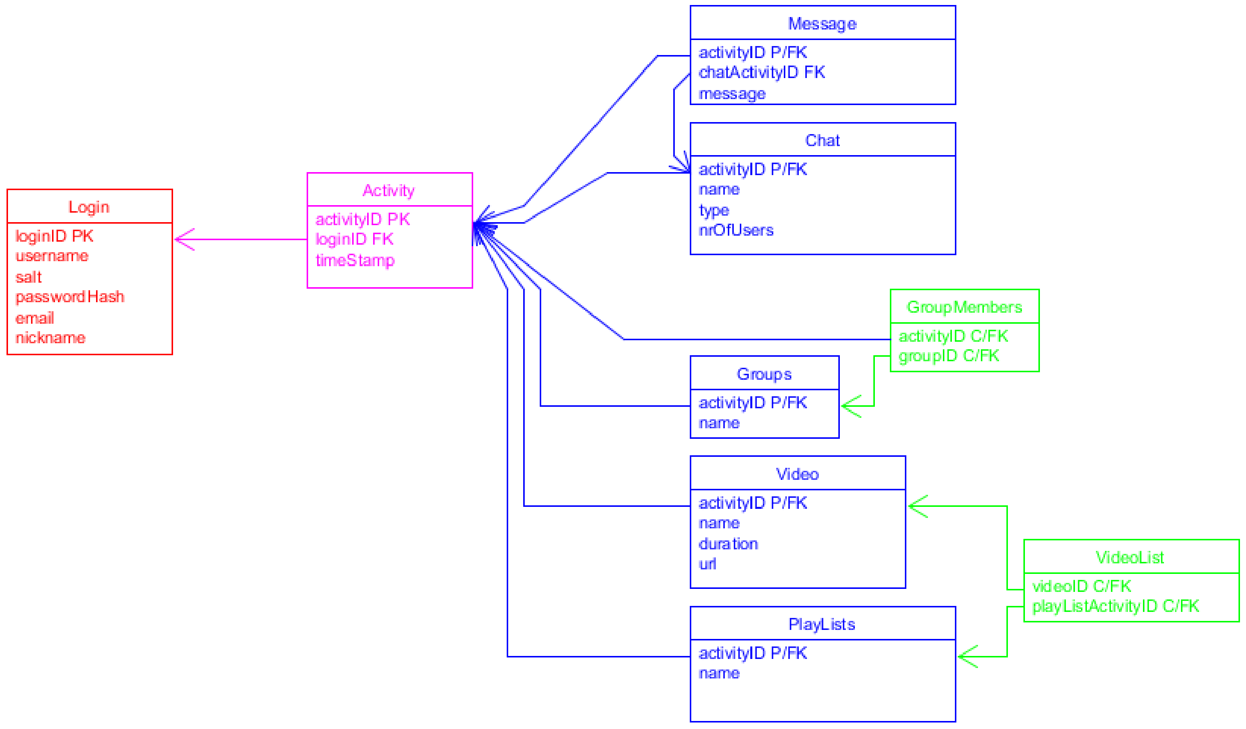


Figure 2

Figure 2 represents how our database looks like. And as you can see we have color coded it: Red represents who did something, Magenta represents when something was done, Blue what was done, and Green is our way of handling Many-to-Many relations, in order to respect the 3 Normalization Forms.

c. Service Architecture

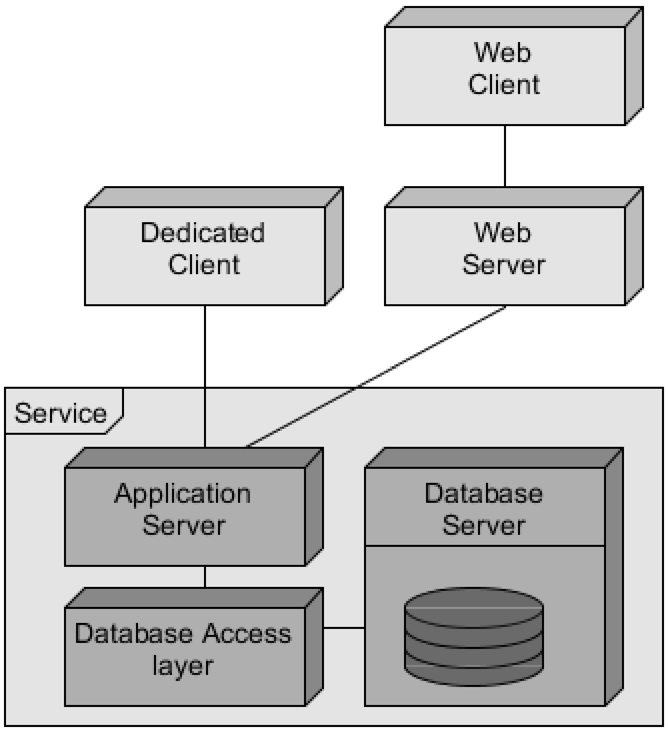


Figure 3

Our service's architecture is a multi-tier architecture (as can be seen in Fig.3), which provides several benefits, such as easy expandability (for example adding a mobile client) and low-cost maintainability. Not only that, but also, this architecture helps us achieve the goal we have set for ourselves, for this project, that being to pursue "high cohesion and low coupling". Other possible architectures which we could've decided to choose were the classic web architecture and client/server architecture, both being dismissed because of their lower level of flexibility.

d. Design Class Diagram

As you can see in our design class diagram, for the service (Fig.4 although hard to read, we invite you to see it in our included files "DesignClassDiagram.uxf"), we have decided to organize the code in several "tiers" (depicted by a different color, in the figure), to help us identify where a specific class should be.

Also, in order to improve maintainability and ease future changes, we added interfaces to both controller and service classes.

Just to clear things out, this diagram's sole purpose was to help visualize the way our service looks like, and was not used in the making of it.

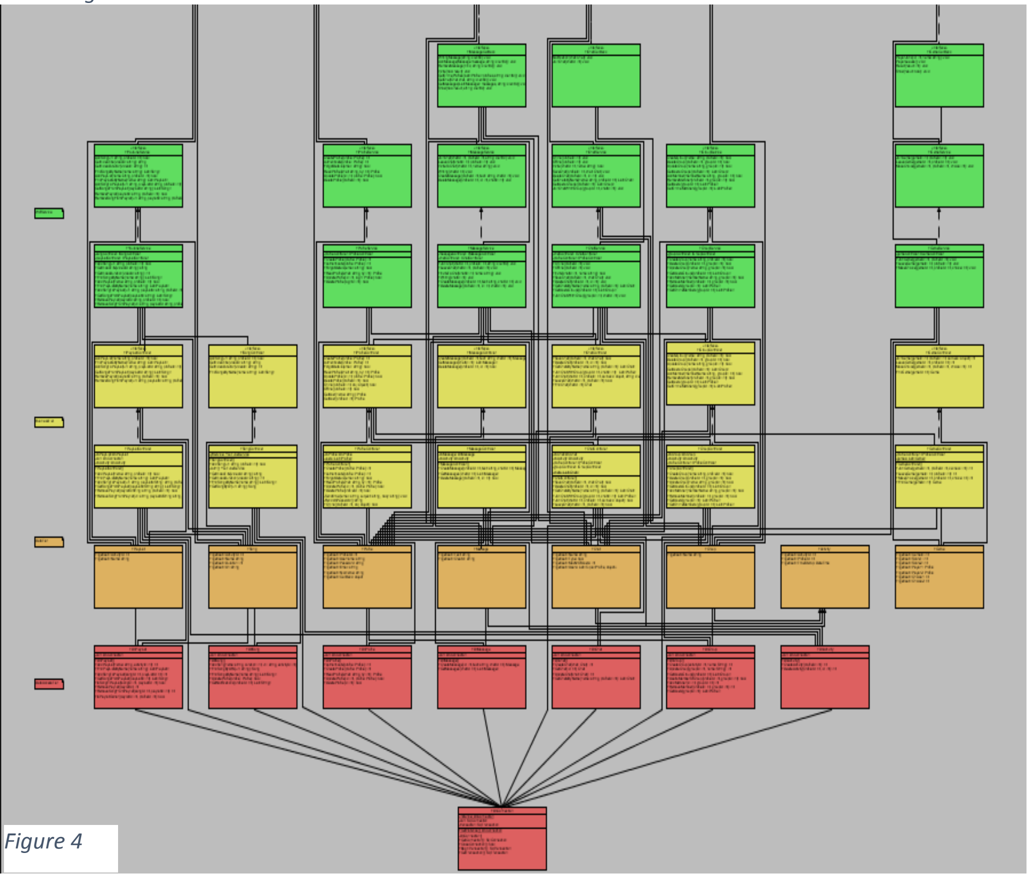
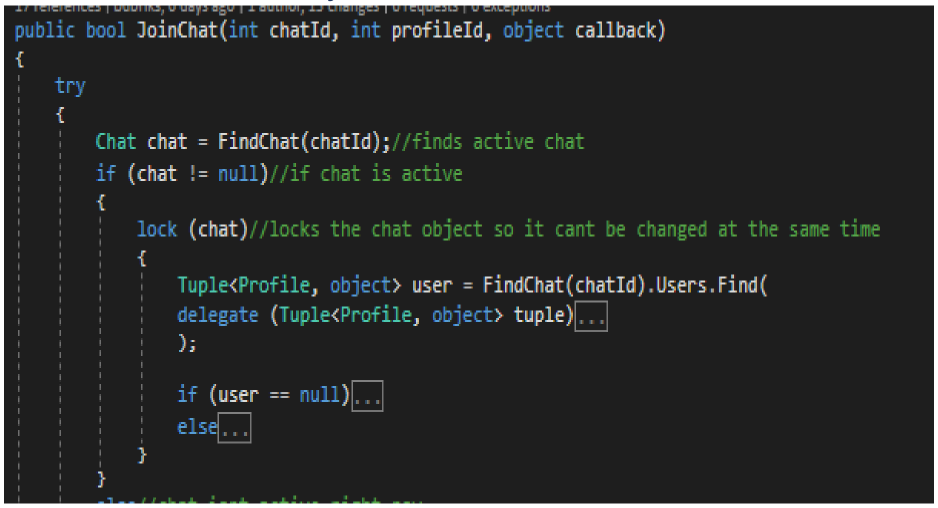


Figure 4

3.4. Implementation

a. Concurrency



```
public bool JoinChat(int chatId, int profileId, object callback)
{
    try
    {
        Chat chat = FindChat(chatId);//finds active chat
        if (chat != null)//if chat is active
        {
            lock (chat)//locks the chat object so it cant be changed at the same time
            {
                Tuple<Profile, object> user = FindChat(chatId).Users.Find(
                delegate (Tuple<Profile, object> tuple)...
                );

                if (user == null)...
                else...
            }
        }
```

Figure 5

The way we handle this issue is by using “locks” (Fig.5 shows an example of how we are using locks): after a user starts joining a chatroom, the service locks that specific chatroom until the action is successfully finished. As for joining a chatroom with a group, the chat is locked and then we receive all chats users and online group members and compare them, if they aren’t already in chat they are added to new list. After all online user, who are not in chat, are added to list, the length of it is checked and if its size is bigger than user limit, no one joins. Otherwise they all reserve place in chat, after which they all receive callback to join this chat. Then the same method (JoinChat) is used, the only difference being that the method is called by each group member. They join and use their already booked place and add their callback object.

As for the ACID properties of our project, we tried ensuring them as follows:

- Atomicity: by using transactions;
- Consistency: by testing the data before saving it into the database;
- Isolation: by using Repeatable Reads and locks;
- Durability: by catching any exception and giving inputs as parameters.

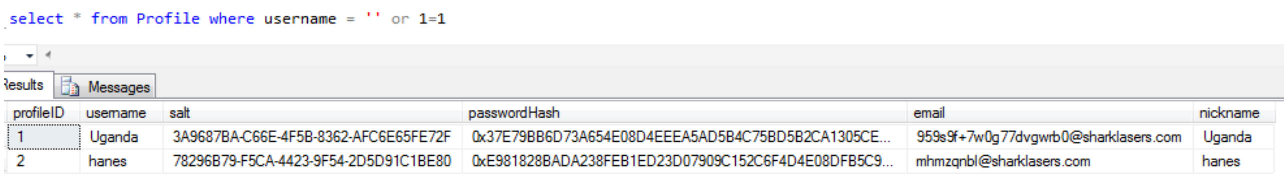
b. Security

b.1. Password storage

This is the first step we took towards security measures in our project. The last thing we want is someone to compromise the database and manage to obtain the passwords of unsuspecting users. What we did was, adding ‘salt’ (a unique random string) to the password and then hashing it before storing, in the database. This ensures that even if someone manages to see all the values in the database, they will not be able to make much use of it as the password is hashed. And because of the salt two same passwords will still have different hash values making it even harder to crack. Of course, we have to store the salt in the database, because it is added to the password every time, before it is hashed and checked against the password hash stored.

b.2. SQL Injection***

SQL Injection is a wide-spread way of executing malicious code on a database. It occurs when a user enters data in such a way that it executes SQL commands that are not supposed to be executed.



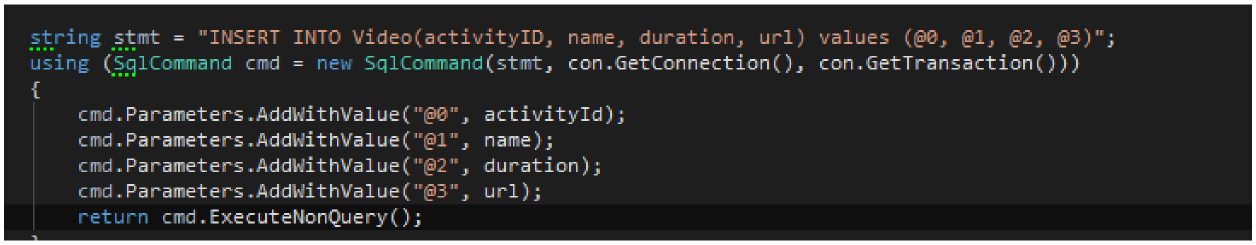
```
select * from Profile where username = '' or 1=1
```

profileID	username	salt	passwordHash	email	nickname
1	Uganda	3A9687BA-C66E-4F5B-8362-AFC6E65FE72F	0x37E79BB6D73A654E08D4EEEA5AD5B4C75BD5B2CA1305CE...	959s9f+7w0g77dvgwrb0@sharklasers.com	Uganda
2	hanes	78296B79-F5CA-4423-9F54-2D5D91C1BE80	0xE981828BADA238FEB1ED23D07909C152C6F4D4E08DFB5C9...	mhmzqnbl@sharklasers.com	hanes

Figure 6

For example, as you can see in Fig. 6, any user introducing such type of statements could potentially drop the database, or even worse, get someone else's private information, essentially transforming our service into malware.

Declaring an SQL command that takes only parameters (as can be seen in Fig.7) rather than strings, prevents users with bad intentions of executing any code that they should not. Not only is this safer, but it also makes the program more robust by returning an integer of how many rows were affected by this statement. In this way we can easily check if the statement was successfully executed. And if that weren't enough, it makes the code way more readable and easier to write instead of having a large number of concatenated strings.



```
string stmt = "INSERT INTO Video(activityID, name, duration, url) values (@0, @1, @2, @3)";
using (SqlCommand cmd = new SqlCommand(stmt, con.GetConnection(), con.GetTransaction()))
{
    cmd.Parameters.AddWithValue("@0", activityId);
    cmd.Parameters.AddWithValue("@1", name);
    cmd.Parameters.AddWithValue("@2", duration);
    cmd.Parameters.AddWithValue("@3", url);
    return cmd.ExecuteNonQuery();
```

Figure 7

b.3. Man-in-Middle

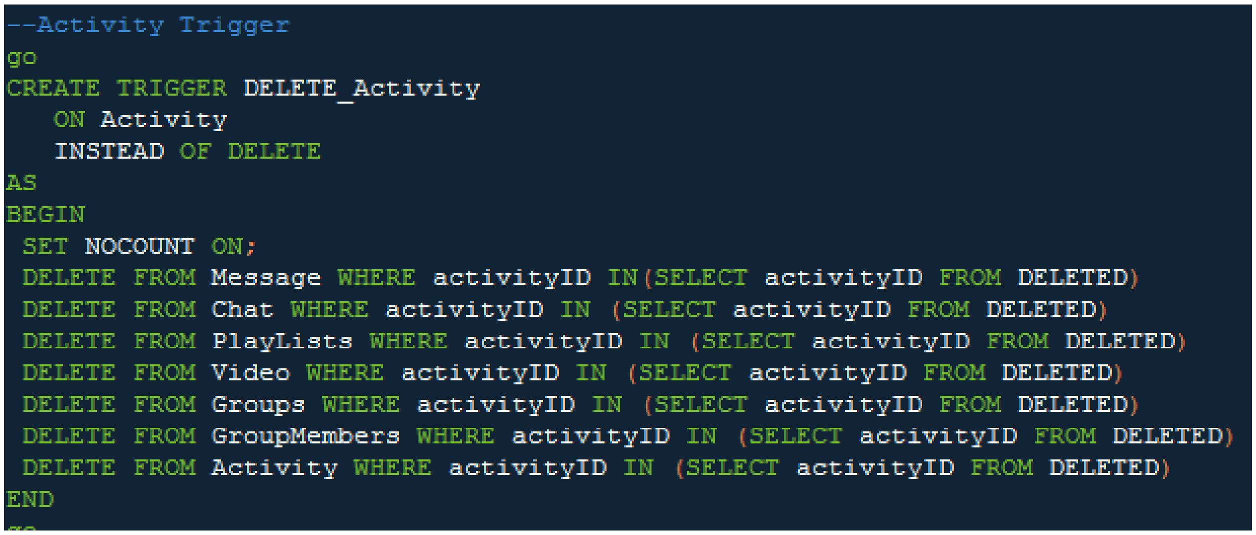
Man-in-middle attack is when someone comes in between the user and the endpoint, fooling the user to connect to a clone of the end-point and then forwarding the message, effectively fooling both the client and the service.

This could allow the attacker to view all the data transmitted and abuse it. However, since we are using "wsHttpBinding" and "wsDualHttpBinding", which have a message layer encryption, all the messages are encrypted from the beginning to the end, preventing the hacker from using any data they might have collected, simply because it's unreadable.

c. Database Triggers

Since one of the features, that would ease our programming task, we have decided in the beginning was cascade deletion, and SQL Server 2014 does not allow such thing when a table is referenced, or references, by multiple foreign keys, we needed to find another way of dealing with phantom data. Shortly after the problem was encountered, we stumbled upon this ingenious way of doing it: Deletion Triggers.

What is a deletion trigger? Well, when a row is deleted from a table, instead of doing the default command, the server would execute this special piece of code, which specifies from which tables to delete and what.

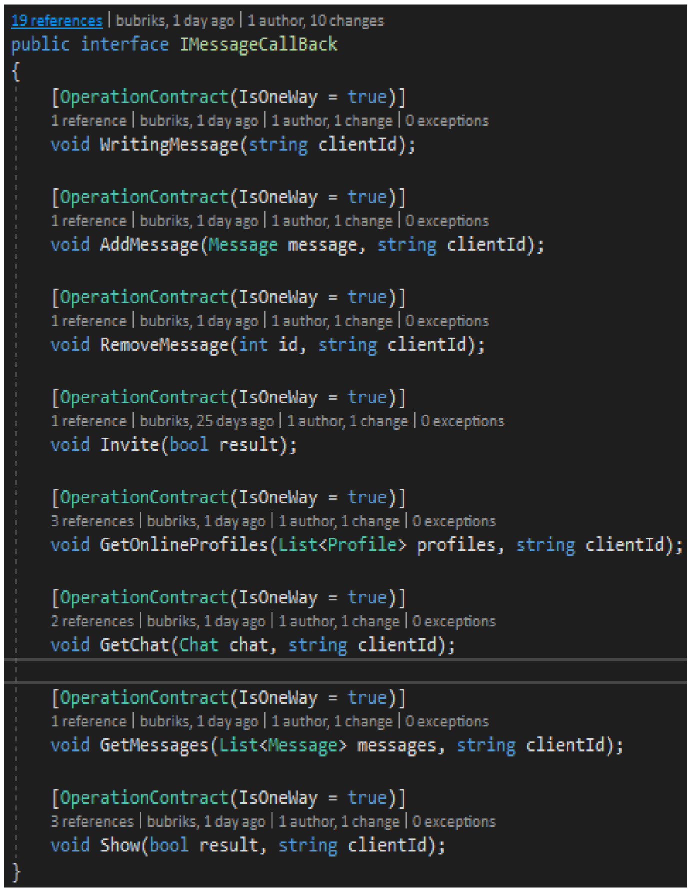


```
--Activity Trigger
go
CREATE TRIGGER DELETE_Activity
   ON Activity
   INSTEAD OF DELETE
AS
BEGIN
 SET NOCOUNT ON;
 DELETE FROM Message WHERE activityID IN(SELECT activityID FROM DELETED)
 DELETE FROM Chat WHERE activityID IN (SELECT activityID FROM DELETED)
 DELETE FROM PlayLists WHERE activityID IN (SELECT activityID FROM DELETED)
 DELETE FROM Video WHERE activityID IN (SELECT activityID FROM DELETED)
 DELETE FROM Groups WHERE activityID IN (SELECT activityID FROM DELETED)
 DELETE FROM GroupMembers WHERE activityID IN (SELECT activityID FROM DELETED)
 DELETE FROM Activity WHERE activityID IN (SELECT activityID FROM DELETED)
END
```

Figure 8

As you can see (Fig.8), before deleting the entry that was commanded to be deleted, the engine would check in all of the mentioned tables if the entry's foreign key, is referenced, then proceed to delete that entry, before continuing the check, and finally executing the commanded entry's deletion.

d. Callbacks to the clients



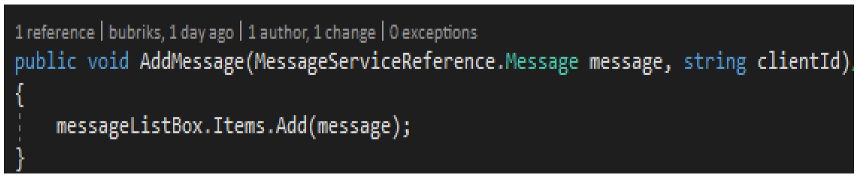
```
19 references | bubriks, 1 day ago | 1 author, 10 changes
public interface IMessageCallBack
{
    [OperationContract(IsOneWay = true)]
    1 reference | bubriks, 1 day ago | 1 author, 1 change | 0 exceptions
    void WritingMessage(string clientId);

    [OperationContract(IsOneWay = true)]
    1 reference | bubriks, 1 day ago | 1 author, 1 change | 0 exceptions
    void AddMessage(Message message, string clientId);

    [OperationContract(IsOneWay = true)]
    1 reference | bubriks, 1 day ago | 1 author, 1 change | 0 exceptions
    void RemoveMessage(int id, string clientId);

    [OperationContract(IsOneWay = true)]
    1 reference | bubriks, 25 days ago | 1 author, 1 change | 0 exceptions
    void Invite(bool result);

    [OperationContract(IsOneWay = true)]
    3 references | bubriks, 1 day ago | 1 author, 1 change | 0 exceptions
    void GetOnlineProfiles(List<Profile> profiles, string clientId);

    [OperationContract(IsOneWay = true)]
    2 references | bubriks, 1 day ago | 1 author, 1 change | 0 exceptions
    void GetChat(Chat chat, string clientId);

    [OperationContract(IsOneWay = true)]
    1 reference | bubriks, 1 day ago | 1 author, 1 change | 0 exceptions
    void GetMessages(List<Message> messages, string clientId);

    [OperationContract(IsOneWay = true)]
    3 references | bubriks, 1 day ago | 1 author, 1 change | 0 exceptions
    void Show(bool result, string clientId);
}
```

Figure 9

In order to notify all the users in a chatroom, that someone is typing a message or that someone has sent a message, or simply joined the chatroom, we needed a way to contact the client, from the service's side. That something turned out to be callbacks. Figure 9 shows an example of our callback methods, which are part of the service's operation contract and are declared in an interface, in the same place where all the other methods, related to Messages, are. Another solution to this issue would've been to make the client refresh the information every so often, but we thought that callbacks are much more elegant. Also refreshing would've drastically increased the network debit, thus reducing the performance of our system.

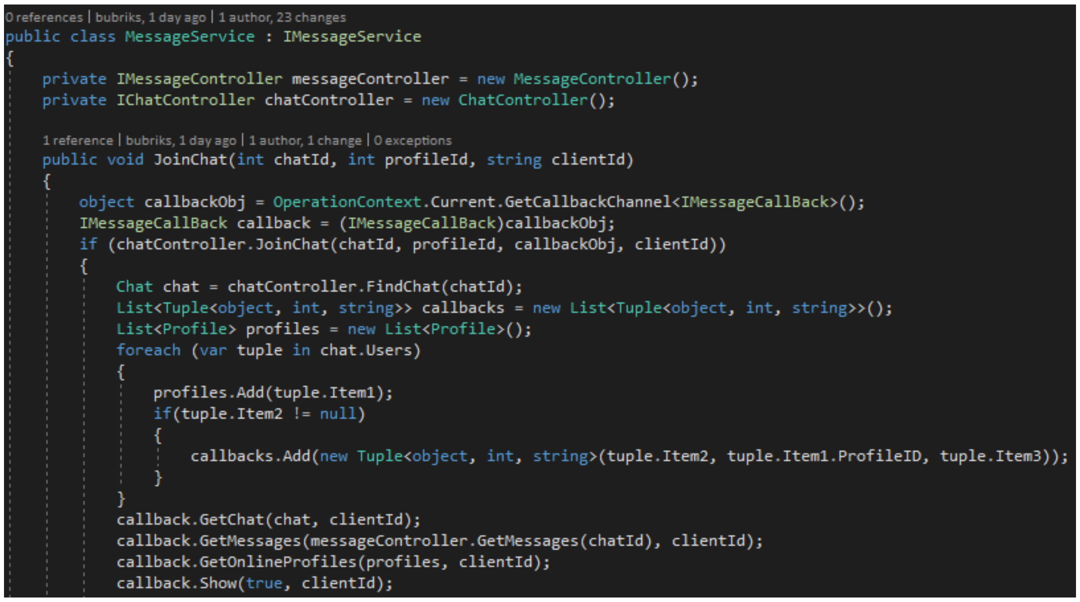
The methods themselves are implemented inside the client's code. (fig 10)

After every time a user joins a chatroom, a callback object, of type "IMessageCallBack", is created and assigned to a Profile and stored in List of Tuples, as can be seen in figure 11.



```
1 reference | bubriks, 1 day ago | 1 author, 1 change | 0 exceptions
public void AddMessage(MessageServiceReference.Message message, string clientId)
{
    messageListBox.Items.Add(message);
}
```

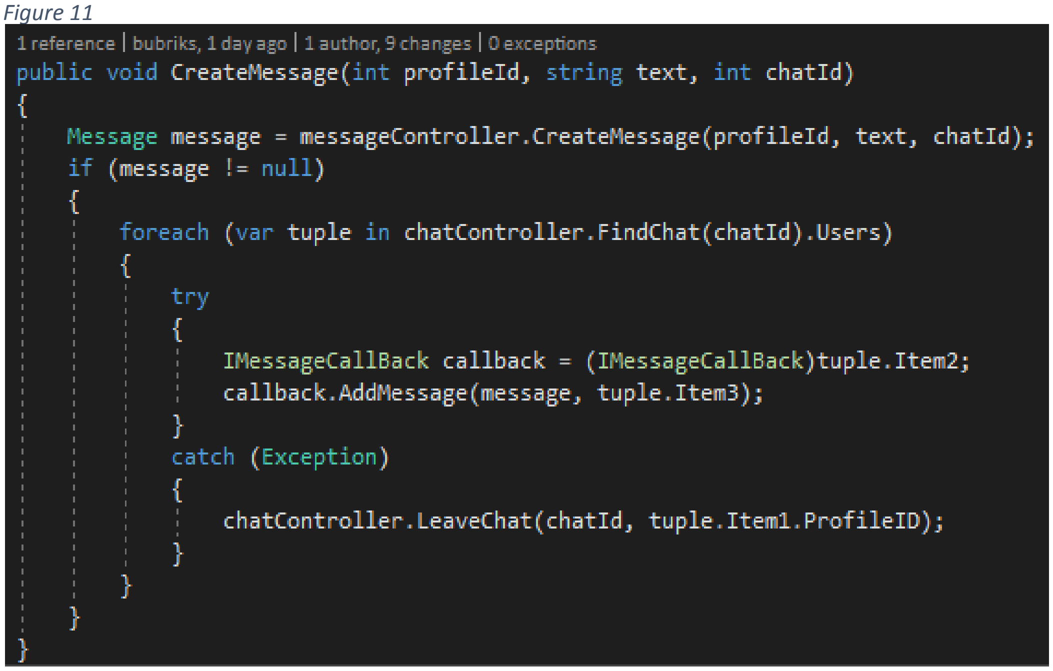
Figure 10



```
0 references | bubriks, 1 day ago | 1 author, 23 changes
public class MessageService : IMessageService
{
    private IMessageController messageController = new MessageController();
    private IChatController chatController = new ChatController();

    1 reference | bubriks, 1 day ago | 1 author, 1 change | 0 exceptions
    public void JoinChat(int chatId, int profileId, string clientId)
    {
        object callbackObj = OperationContext.Current.GetCallbackChannel<IMessageCallBack>();
        IMessageCallBack callback = (IMessageCallBack)callbackObj;
        if (chatController.JoinChat(chatId, profileId, callbackObj, clientId))
        {
            Chat chat = chatController.FindChat(chatId);
            List<Tuple<object, int, string>> callbacks = new List<Tuple<object, int, string>>();
            List<Profile> profiles = new List<Profile>();
            foreach (var tuple in chat.Users)
            {
                profiles.Add(tuple.Item1);
                if(tuple.Item2 != null)
                {
                    callbacks.Add(new Tuple<object, int, string>(tuple.Item2, tuple.Item1.ProfileID, tuple.Item3));
                }
            }
            callback.GetChat(chat, clientId);
            callback.GetMessages(messageController.GetMessages(chatId), clientId);
            callback.GetOnlineProfiles(profiles, clientId);
            callback.Show(true, clientId);
```

Figure 11

The previously created callback objects are then used every time a new “notification” must be done for the affected users. For example, the following image (fig.12), shows how callback objects are used in order to notify the other members of a chatroom, that someone wrote a specific message.

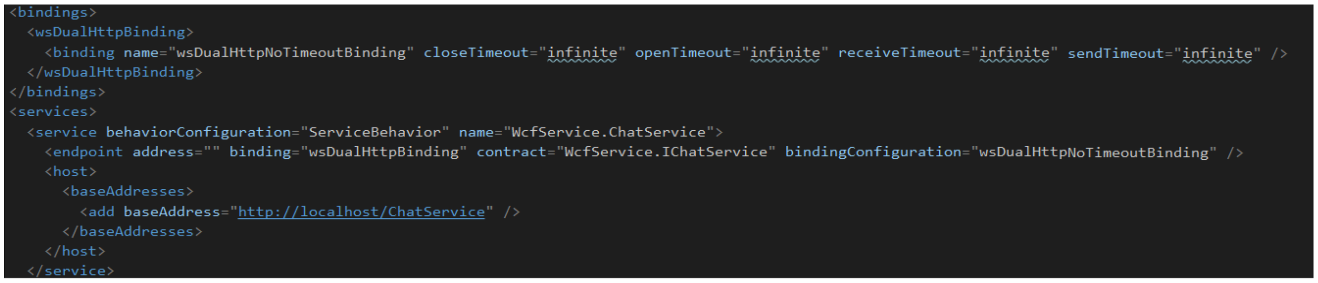


```
1 reference | bubriks, 1 day ago | 1 author, 9 changes | 0 exceptions
public void CreateMessage(int profileId, string text, int chatId)
{
    Message message = messageController.CreateMessage(profileId, text, chatId);
    if (message != null)
    {
        foreach (var tuple in chatController.FindChat(chatId).Users)
        {
            try
            {
                IMessageCallBack callback = (IMessageCallBack)tuple.Item2;
                callback.AddMessage(message, tuple.Item3);
            }
            catch (Exception)
            {
                chatController.LeaveChat(chatId, tuple.Item1.ProfileID);
            }
        }
    }
}
```

Figure 12

e. Keeping connection to clients from timing-out

One of the main problems we encountered, after creating a successful connection to our clients, was the connection being dropped after a few minutes of inactivity, from both the service’s and client’s sides. The issue would appear when the service tries to callback a specific user, without having a connection to them. To solve this issue, we created a custom binding configuration, as can be seen in figure 13.



```
<bindings>
  <wsDualHttpBinding>
    <binding name="wsDualHttpNoTimeoutBinding" closeTimeout="infinite" openTimeout="infinite" receiveTimeout="infinite" sendTimeout="infinite" />
  </wsDualHttpBinding>
</bindings>
<services>
  <service behaviorConfiguration="ServiceBehavior" name="WcfService.ChatService">
    <endpoint address="" binding="wsDualHttpBinding" contract="WcfService.IChatService" bindingConfiguration="wsDualHttpNoTimeoutBinding" />
    <host>
      <baseAddresses>
        <add baseAddress="http://localhost/ChatService" />
      </baseAddresses>
    </host>
  </service>
```

Figure 13 This configuration would prevent the connection from being deleted, until either the service or the client would manually close it.

3.5. Tests

a. Testing the system

Since we have decided to work following a combination between Extreme Programming and SCRUM methods, one of the main “rules” was to develop the service while practicing Test Driven Development. And because of that we have ended up having 104 tests, which turned out to be quite useful. On the way towards making the final product, we found ourselves, numerous times, being “saved” by already existing tests, simply because they pointed out that the newly created code is disrupting the previously working one.

b. Performance

Here we have measured the time it takes for certain methods to be executed completely. The tests were conducted using the “System.Diagnostics.Stopwatch” class by starting the timer at the beginning of a button click and printing the result as soon as the action was completed.

The following chart (fig.14) shows the time it took to login with up to 16 online users by sending a login request every second:

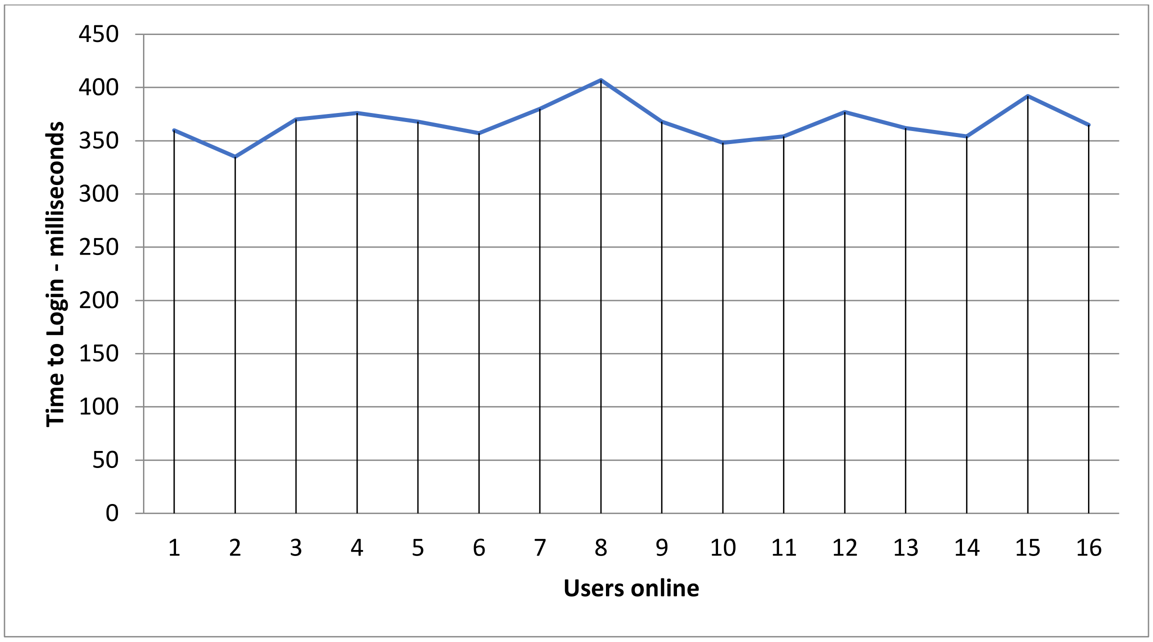


Figure 14

Average: 367ms, Best: 335ms, Worst: 407ms.

As you can see the results are very close to each other and the differences can be accounted to outside factors such as current load of the network it goes through, background OS process etc.

Compared to the time it took to join a chatroom as a group of 20 users simultaneously which was 569 milliseconds, we can conclude that there is barely any decay in performance, and can safely assume that the service will be able to handle at least 50 concurrent clients.

In the next chart (Fig.15) you can see concurrent requests to join a chat. This was done by creating groups with various sizes and joining a chat as a group (which sends concurrent requests to join depending on the group size).

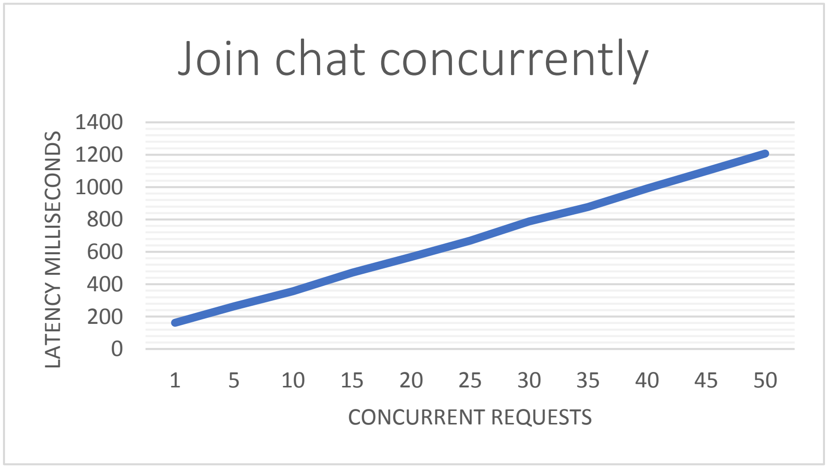


Figure 15

It is obvious that the decrease in performance is steady and predictable. For every additional 5 concurrent requests the latency increases by approximately 100 milliseconds. Based on this evidence we can assume that at about 100 concurrent requests a delay of more than 2 seconds will occur, and this might be annoying for some users. Also judging from the tests, we can speculate that at around 200 concurrent join requests the system will be on the verge of being unusable as the delay will be around 5 seconds which will cause a time-out. But, in order to solve this problem, we have set a “decent” limit in chatroom to 25 places, where we expect a delay of approximately 550 milliseconds, in worst case scenario.

All tests have been done on a laptop with the following configuration:

- Cpu: intel i7-6700HQ
- RAM: 8Gb DDR4
- GPU: nvidia 950m
- Storage: 250Gb SSD

Of course, the more “power”, the more clients it should be able to handle, however we decided to test our service on one of the machines we had access to. During the test, both clients and the service was running on the same machine.

4. Conclusion

4.1. Denouement

In conclusion, during this semester we managed to not only gain knowledge about various software development frameworks and a new programming language, but also a new development concept, that being programming a distributed service.

Our product, turned out to be quite close to what we imagined when we were pitching the idea. Most, if not all the features, we intended to create, were implemented in the application. For the next few sprints (sometime in the future), we will bring the web client to the dedicated client’s level and make the UI look fancier; all in order to be able to eventually deploy the project on the internet, so people can start consuming our service.

To see how we worked and what files we created, one has to follow the link, which will take you to our GitHub repository: https://github.com/bubriks/Turakas

As an ending note, we would like to thank all the readers, who invested their time in reading this paper, also the guiding teachers, who helped and guided us throughout the entire process. All files used in the creation of this report are attached to the hand-in folder, in case you would like to inspect them in great detail.

4.2. References

*According to James Serra, Big Data/Data Warehouse Evangelist at Microsoft in a presentation Published on Mar 15, 2016

**According to Microsoft public documentation on Windows-forms and MVC, available at the following links:

- https://docs.microsoft.com/en-us/dotnet/framework/winforms/windows-forms-overview
- https://docs.microsoft.com/en-us/visualstudio/designers/introduction-to-wpf

***Inspired by an article from W3School on SQL Injections, available at the following link:

- https://www.w3schools.com/sql/sql_injection.asp

4*Inspired by Microsoft public documentation on isolation levels, available at the following link:

- https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels
- https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/configuring-system-provided-bindings

5* Inspired the use of SignalR:

- https://docs.microsoft.com/en-us/aspnet/signalr/overview/getting-started/tutorial-getting-started-with-signalr-and-mvc

6* Inspired the use of callbacks:

- https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/how-to-create-a-duplex-contract

7* Special thanks to the community on “Stackoverflow” for helping with some basic tasks and visual studio configurations.