

# Programming Languages Project: Final Report

## Eduardo Larios Fernández

A00569364 – 08/11/2018

Contents

[Programming Languages Project: Final Report 1](#_Toc530423229)

[Context of the Problem 2](#_Toc530423231)

[Proposed Solution 3](#_Toc530423232)

[Solution Architecture 5](#_Toc530423233)

[Server 5](#_Toc530423234)

[Client 5](#_Toc530423235)

[Results 8](#_Toc530423236)

[Paradigm Analysis 8](#_Toc530423237)

[Conclusions 9](#_Toc530423238)

[Setup Instructions 9](#_Toc530423239)

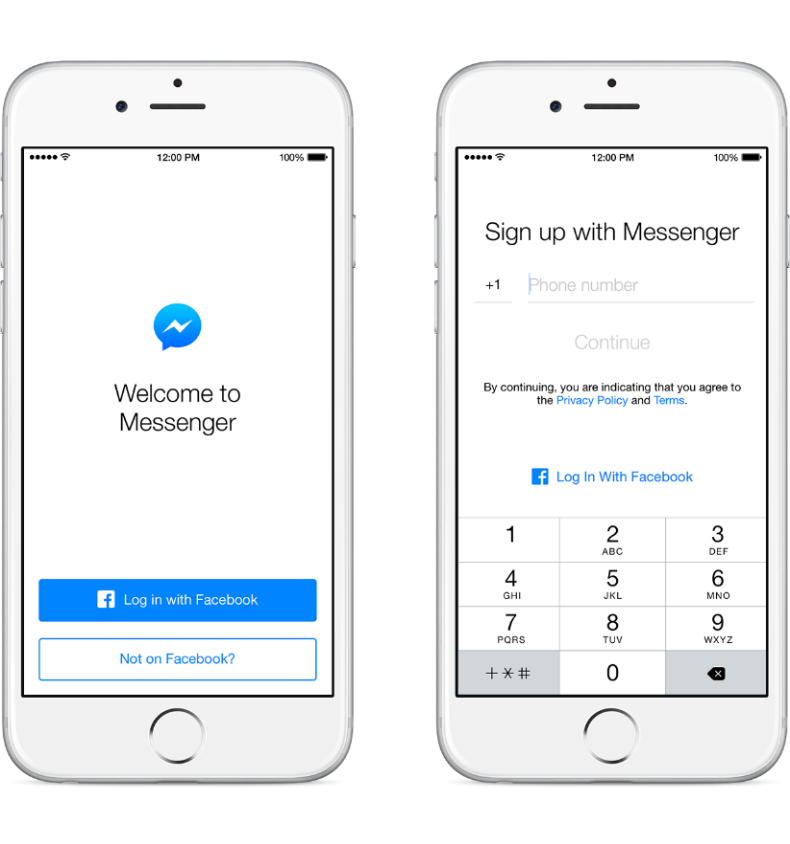
[Evidence 10](#_Toc530423240)

[References 11](#_Toc530423241)

# 

# Context of the Problem

It is not an uncommon situation to want to chat with people during an event but not having their contacts, or it not being worth the hassle of creating a chat thread that will only be used by the members through the duration of a single event.

Normally we wish to have the advantages of instant messaging and all the pluses it brings, like easy sharing of multi-media files and instant acknowledgement, to make collaboration easier in situations like courses and classes without being annoyed by the creation of the thread and having to go through the process of asking people their Messaging-IDs or phone numbers and adding them one by one to the chat.

One specific situation that I think is worth tackling is classes or courses where people may want to collaborate during the session but don’t know each other, and don’t want to add each other to a messaging service, or even exchange phone numbers.

It is understandable not to want to share your contact details so easily, as they could be misused by another party.

During these classes alumni normally want to share their thoughts with their peers or professors in real-time, as well as sharing any supplementary material that can be useful in context.

A “disposable-chat” is in my opinion a great solution to this problem.

Annoying Registration

Applications like Messenger and WhatsApp or even Discord and Slack are great when you maintain communication with people for a long enough period of time, that makes the tradeoff of signing up to a platform or sharing contact details more bearable, the convenience of having all the features of the messaging service wins over the annoyance of having to add everyone to it.

However, the main idea of this proposal is to give an alternative to the current state of instant messaging. There are way too many services that require logins, accounts, and contact details before they work, and very little in the way of local instant communication, and lots of them requiring a constant connection to the internet with their clients being more in line with web apps.

# 

# Proposed Solution

A small chat application could be created using a client-server architecture to allow people to communicate in these situations. A person could start a local server that is native to the local network and share it with other people who want to participate in the conversation.

People could connect to this small server with just a conversation ID to distinguish them from other possible local servers in the area, similar to the way Kahoot works to share a quiz with people in a local area.

People could then enter just the chat ID and their chosen username to join the conversation instantly and without the hassle of creating a permanent account or having to exchange phone numbers unwillingly with people who might be in their classes during just a week or semester.

Disposability

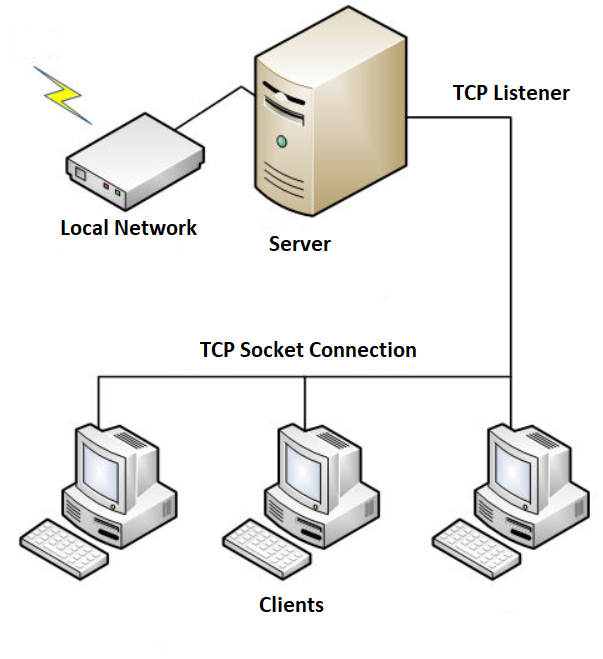
This implementation could then add common instant messaging features, like sharing media, files, video recordings, voice recordings, or eventually adding even more complex features like games or collaboration extensions.

The architecture would be simple and limited due to the way the server would be running locally on one of the users’ phone. The idea would be to limit the maximum number of possible connections depending on the hardware provided for the server or limiting functionality if a user doesn’t want to limit the number of possible clients.

Complex applications like games would likely suffer if the server hardware was incapable of keeping up with the load of all the clients’ data. If the server was created using more powerful hardware, like a laptop or a desktop PC, then it would be possible to maintain a stable connection with multiple clients even with substantial data load, leaving network latency and performance as the main limitation to have more clients.

However, for simple things like sending text and small files, like photos and audio recordings, a server being hosted on a modern smartphone should suffice for a decent number of clients. It is important to remember that the main point of the application is to allow small groups of people to communicate easily, not replace a fully-featured messaging application like WhatsApp, Facebook Messenger, or even more complex alternatives like Slack or Discord. Replacing the previously mentioned options is simply beyond the scope of the proposal, never mind the project.

# Solution Architecture

The solution exists at the moment as a proof of concept that uses a client-server architecture on a local scale to allow communication between parties in a local network. The implementation makes use of threads in both the client and the server to serve different purposes.

## Server

In the server each client connection is handled spawning a dedicated thread to listen for two-way communication between the client and the server using a TCP Listener on the server and a TCP Client on the client.

Because of the architecture messages are not sent directly between different clients, instead, each message first passes through the server and then, once it is verified, it is relayed to the other clients connected through their own thread-socket.

As is common in this type of architecture, the burden of processing all the requests falls on the server, as the clients have very simple logic that just allows them to connect to the server to send messages and receive broadcasts.

As each client has its own thread to attend its request it is important for the server to have multithreading capabilities in its hardware if it is to permit a high number of client connections. Desktop grade processors should be up to the task of handling dozens of threads with ease, as a small number of them will be handled by a hardware dedicated thread. Low core count processors with no multithreading capabilities might have low performance on this implementation.

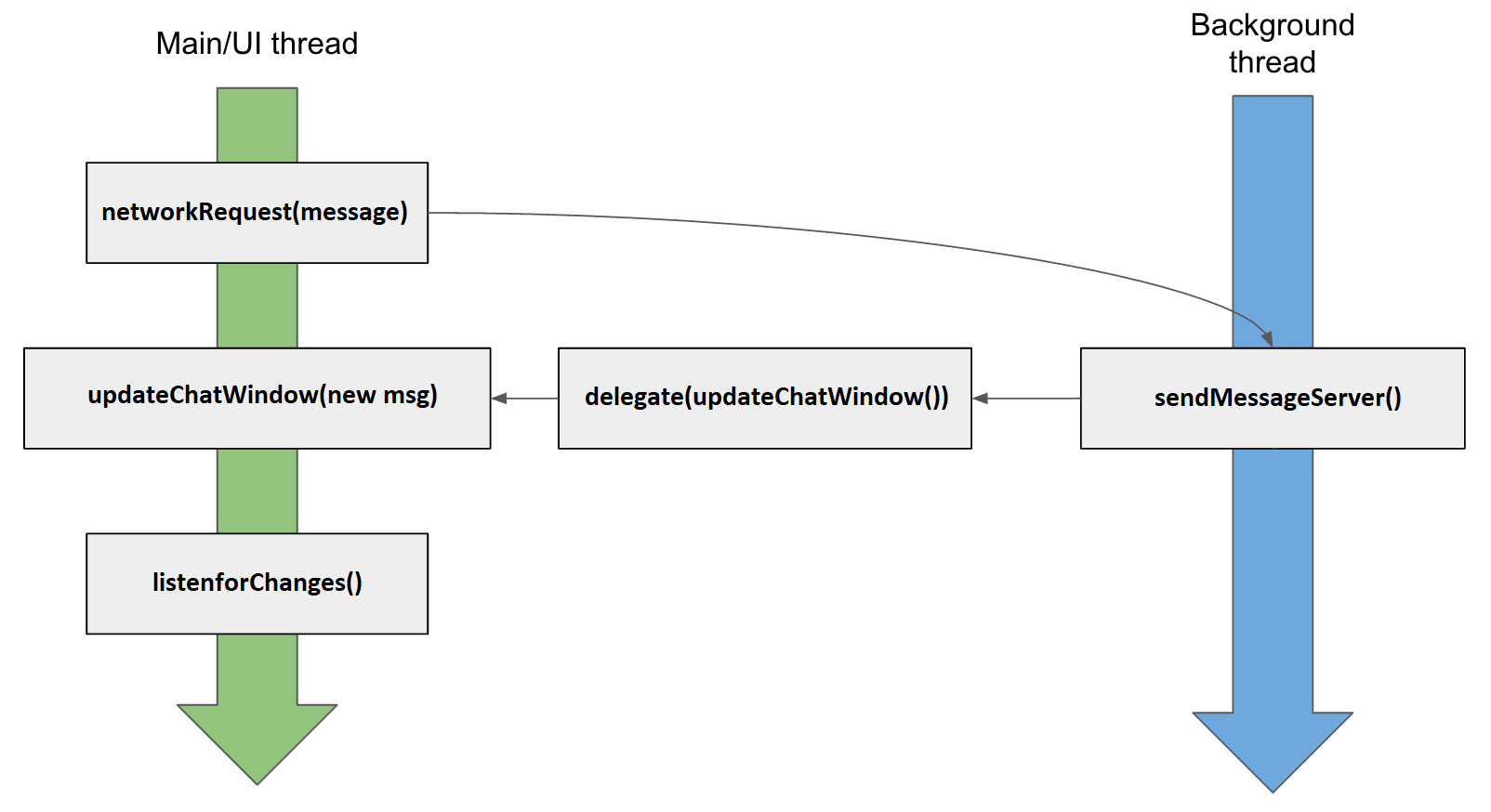
### Client

The client is implemented with a foreground UI thread, using Windows Presentation Foundation in the demo, and a background thread to implement the necessary network connectivity and verification logic.

It is necessary to keep the UI and logic threads separate to maintain a responsive design in the application, it is not recommended to do complex logic on the UI thread as it could become unresponsive to the user and exhibit bad performance when under heavy load, for example when showing media or processing a game.

Of course, these are not concerning at the moment as the application only supports messaging functionality at the moment, however, in case it is extended to support more functionality in the future, as described in the proposal, it can become a significant performance concern.

The separation of logic and presentation did present some problems during implementation, as most UI updates have to be done asynchronously with the use of delegates, a C# feature that allows a thread to call a different thread to execute a method that it is not able to complete. This happens because the logic thread cannot directly update the window object’s, as they are owned by the UI thread with a lock.

Trying to call the logic thread to directly update the chat window will crash the application with a deadlock, as it will keep waiting indefinitely for the lock to expire on the window, which will never happen as long as the UI thread (the one spawned by the window) keeps running, which is obviously necessary for the application. The following diagram explains the way a call to the server works.

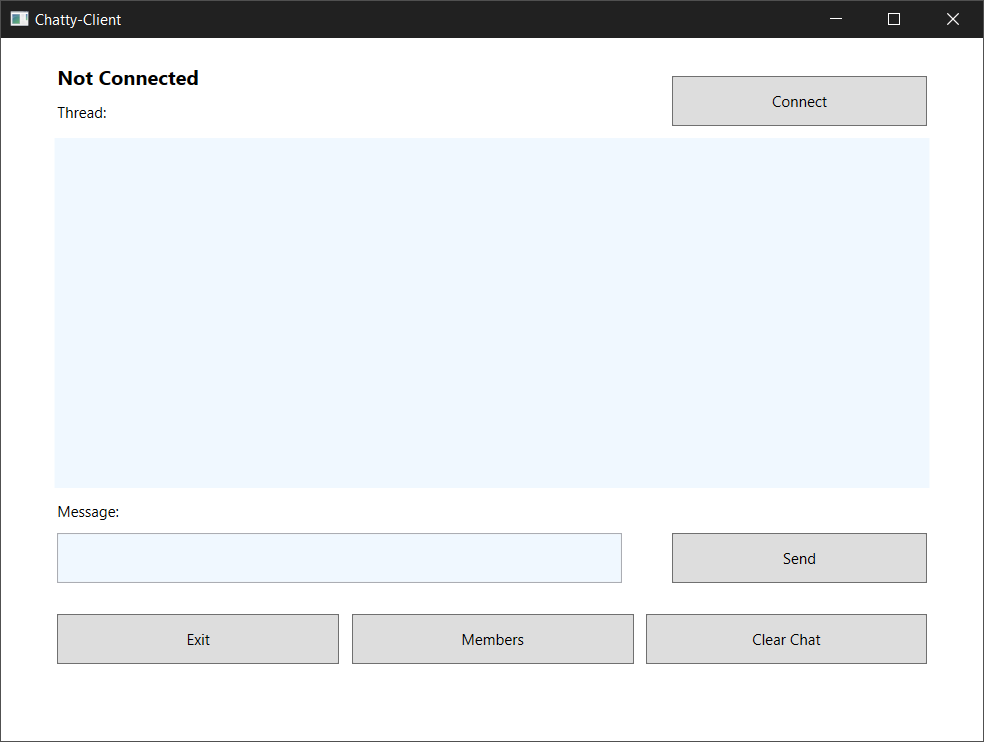
Multithreading Model (Client)

While this makes the logic more complex than a simple blocking application would be, for example using WinForms, it allows a clear separation of concerns that let’s the UI be completely independent of the logic, netting a more responsive application and better performance during intensive activities.

At the moment the UI is pretty barebones, allowing the connection to a server hosted on the local machine and a max number of 20 clients. The server is a simple console application that uses .NET Core to listen for incoming connections and handle its requests. The technology choice was a deliberate choice and not a coincidence.

While most people would select Java when working with multithreaded applications -as it runs in most operating systems, including Android, and has a pretty straightforward threading model- using .NET Core with [Windows Presentation Foundation](https://en.wikipedia.org/wiki/Windows_Presentation_Foundation), which uses XAML for the UI description layer, makes the porting of this application relatively simple.

At the moment the server (and console client) can work on Linux, Mac OS and Windows, while the graphical UI client at the moment only works on Windows using the complete .NET Framework. Using multiplatform UI frameworks based on XAML like [AvaloniaUI](https://github.com/AvaloniaUI/Avalonia) can permit the same graphical client to run on the previously mentioned platforms.



Current Desktop UI (WPF)

In the case of mobile ports, it would be possible to port almost the same logic to a [Xamarin.Forms](https://docs.microsoft.com/en-us/xamarin/xamarin-forms/) application, allowing the use of .NET and C# to run on both Android and iOS. While not in the works at the moment the possibility of using this application in multiple platforms is not completely out of reach and would help the original intent even more, allowing the use of multiple devices to communicate as needed.

# Results

The application was tested on several scenarios to document how it would perform. I was assisted by several people to test that the application allowed connectivity to the server on a local network, and was not restricted to running only on the local host of the server PC.

The local tests showed that it was possible to reach the maximum number of clients, 20 at the moment, to allow them to chat concurrently on the same server. The chat exhibited good performance as noted by the different testers, and didn’t show any sign of slowdowns on either the interface or the message delivery.

As was expected from the architecture, it should be able to scale up the service to several tens of users if a decently spec’d PC is available as the messaging server. Several of the participants of the test shared their opinions and experiences using the solution on a Google Survey that will be linked under the references section.

While the application worked correctly under the circumstances outlined previously, the users expected some extra features that were planned, but considered for a future iteration of the application. Some of these features included the possibility of sending files and posting images, gifs, and videos to the same chat.

The users did appreciate the simplicity of just selecting a pseudonym and getting right into the chat, without a need for logins or registrations. Even if the application is considerably barebones in its current state, the users considered the idea a valid one and hoped that something similar to it, just with more features, would be available for common usage.

# Paradigm Analysis

A vital part of the solution was the usage of multi-threading to handle multiple requests asynchronously in the backend of the application. In the case of the server a simple threading approach is used. Normally a single [TCP Listener](https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.tcplistener?view=netframework-4.7.2#remarks) (basically a wrapper for sockets in .NET) is used to maintain communication with a client over the network. This has significant overhead in comparison to using multiple threads to handle each a dedicated connection the client.

Using a thread for each connection allows the main thread to keep free most of the time to accept new connections or to close unused ones. The threads in the background implement the logic necessary to maintain the connection and to broadcast messages received from one client to the others. This approach also helps simplify the main thread logic, as it only acts as a starting point to new connections.

On the other side the usage of threads on the frontend, or client, of the application is necessary to maintain a responsive design. It is recommended by the .NET documentation and the WPF guidelines to use background threads for heavy computation or constant connections to a network resource.

Following these references Chatty implements an interface that is completely separated from the backend logic, and only uses delegates to communicate between the TCP Client running in the background, and the main UI thread using WPF in the frontend. The use of threading here allows the application to maintain good performance on its interface even during heavy load.

# Conclusions

It is indeed not trivial to create a multiplatform messaging client, props to the companies that manage to do it while retaining a consistent UI while having to work with a great variety of UI frameworks and platform SDKs. On the other side while a simple client-server architecture is in itself not overly complex, the use of threads to handle requests and maintain a responsive GUI do make the implementation considerably more complex.

As the first time working with threads in an application that wasn’t a trivial example I quickly found it hard to reason about some of the challenges presented. The server implementing a dedicated thread for each of the clients required a great amount of work to synchronize it correctly. While the .NET Framework core libraries help to make mutex creation simple, it is still a complex ordeal to know which resources should be share, when, and how. This helped me deepen my knowledge about semaphores and synchronization.

In the case of the client things where polarizing. While the client logic is considerably simple in comparison to the client, I had never worked with asynchronous programming on the frontend, and also had never used Windows Presentation Foundation.

The complexity of making a responsive non-blocking UI was staggering to say the least, and required long trips to the Microsoft documentation to solve lots of blocking issues. I was not familiar with delegates in .NET, which allow you to make a different thread execute code that the calling thread is unable to execute itself. While the idea was very simple, it required a lot of hand tuning to prevent deadlocks from happening from shared objects not being available at the moment they were to be accessed.

All in all, I consider the experience a big eye opener on the difficulty of creating truly multithreaded, concurrent applications that allow better performance for the user. Most people simply expect an application to work out of the box, and even us as developers can easily underestimate the complexity of dealing with threads, as we are used to simple to understand single-threaded applications.

As we’ve seen, the way forward is learning how to program using multiple execution threads, as big single-thread gains are now a thing of the past. The burden now falls on us developers to better utilize the hardware that is available and learn how to reason about with this new computing paradigm.

# Setup Instructions

The application setup should be simple enough to understand. However, it is important to note that at the moment the client and server use different frameworks to work. The server uses .NET Core, which requires the installation of its runtime in different platforms, including even Windows, as it runs managed code on a virtual machine, similar to Java.

The client on the other hand uses .NET Framework 4.7 to run WPF. These libraries are not available in .NET Core, and so, the current GUI client only works on Windows, which does have the complete libraries required from Windows 8 and onwards.

The steps to setup the application are the following:

1. Clone the repository from github with:

git clone <https://github.com/EduardoLarios/Chatty>

1. Open a CMD prompt and verify the existence of .NET Core 2.0+ with the following command:

dotnet –version

1. If installed go the server folder and execute the following command where the .csproj file resides:

dotnet run

1. Finally, a client executable binary should be available in the Client folder inside

/Chatty-UI/bin/Debug/Chatty-UI.exe

Simply double click the app to run it and follow the instructions on the screen.

i.e. Click on the “Connect” button

**Note:** If .NET Core is not installed in your computer please follow the instructions provided by Microsoft in the following link to install .NET Core 2.0+:

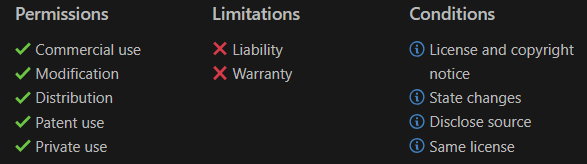
* https://www.microsoft.com/net/download

# Evidence

The evidence of the project is available in the following GitHub repository:

* <https://github.com/EduardoLarios/Chatty>

This repository includes the code, a client binary and the README and LICENSE files. The project is licensed under de GNU General Public License v3 that include the following permissions.



LICENSE

# References

The following references where consulted during the completion of the project.

1. Threads: <https://docs.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading>
2. Dispatchers: <https://docs.microsoft.com/en-us/dotnet/api/system.windows.threading.dispatcher?view=netframework-4.7.2>
3. Delegates: <https://docs.microsoft.com/en-us/dotnet/csharp/distinguish-delegates-events>
4. Windows Presentation Foundation: <https://www.tutorialspoint.com/wpf/index.htm>
5. Windows Presentation Foundation: <https://docs.microsoft.com/en-us/dotnet/framework/wpf/getting-started/walkthrough-my-first-wpf-desktop-application>
6. Architecture: <http://www.networkcomms.net/how-to-create-a-client-server-application-in-minutes/>
7. AvaloniaUI: <http://avaloniaui.net/docs/quickstart/>
8. Xamarin.Forms: <https://docs.microsoft.com/en-us/xamarin/xamarin-forms/get-started/hello-xamarin-forms/quickstart?pivots=windows>
9. GNU License: <https://tldrlegal.com/license/gnu-general-public-license-v3-(gpl-3)>
10. TCP Listener: <https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.tcplistener?view=netframework-4.7.2#remarks>
11. TCP Client: <https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.tcpclient?view=netframework-4.7.2#remarks>
12. Using threads in WPF: <https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/threading-model>
13. Results Survey: <https://docs.google.com/forms/d/1ej3DsUlRJTftemyT0_3EffpsPTkpMhRo2z8L2eWaMg0/edit?usp=sharing>