Alternativen zum Windows Communication Framework (WCF)

Erstellen einer einfachen Chat-Anwendung mit .NET Core und IPC Service Framework

Beschreibung der Zielanwendung

Die Clients sind WPF-Fenster (siehe Bild). Startet man das Programm wird man zunächst nach einem Namen gefragt. Danach öffnet sich das eigentliche Hauptfenster. In diesem kann man unten einen Text eingeben. Beim Klicken auf „Senden“ wird dieser Text und der Name des Nutzers an den Server geschickt. Der Server sendet ihn außerdem mit Zeitstempel an alle Clients und speichert ihn intern in einem Chatverlauf. Beim ersten Anmelden bekommt ein Client außerdem den kompletten Chatverlauf seit dem Neustart des Servers (wir brauchen keine Datenbanken oder irgendwas, nur eine Liste mit Chatbeiträgen). Weiterhin wird in einem Chatfenster auch immer angezeigt wie viele Personen aktuell für den Chat angemeldet sind. Schließt jemand das Programm wird er automatisch abgemeldet und verschwindet aus der Liste.

Potenzielles Aussehen der Zielanwendung

Eingesetzte Technologien

- Plattform: Microsoft .NET Core 3.1
- Client: WPF-Anwendung mit MVVM Light
- Server: Konsolenanwendung mit IPC Service Framework

Quellcode

Der gesamte Quellcode ist auf GitHub öffentlich verfügbar: https://github.com/benjamin-hempel/Vergleich-WCF-Alternativen/tree/master/IPCChatApplication

Die Solution besteht aus zwei Verzeichnissen: WPFClient für den Client und IPCServer für den Server. Genaueres zum Quellcode wird auf den folgenden Seiten erläutert.

Starten der Anwendung

Es gibt zwei Möglichkeiten, die Anwendung zu starten:

1. Öffnen der Visual Studio Solution IPCChatApplication.sln und Starten der Projekte WPFClient und IPCServer.
2. Starten der .exe-Dateien für Client und Server. Diese befinden sich in folgenden Verzeichnissen (ausgehend vom Hauptverzeichnis):
    a. .\ WPFClient\bin\Debug\netcoreapp3.1\WPFClient.exe
    b. .\ IPCServer\bin\Debug\netcoreapp3.1\IPCServer.exe

Das Service Model im Detail

Das Service Model setzt sich aus zwei Komponenten zusammen:

1. Zum einen das Interface IChatService, welches die Methoden, die Client und Server unterstützen müssen, enthält.
2. Zum anderen die Klasse Message, welche eine einzelne Chat-Nachricht mit ihren Attributen modelliert.

Beide befinden sich im zwischen Client und Server geteilten Namespace IPCChatApplication.Shared.

Das Interface IChatService

Das Interface IChatService deklariert die folgenden Methoden, welche jeweils auf Client- und auf Serverseite implementiert werden müssen:

- void SendChat(string name, string text): Übermittelt die Nachricht vom User „name“ mit dem Inhalt „text“ an den Server und hat keinen Rückgabewert.
- void Join(string name): Teilt dem Server mit, dass der User „name“ dem Chat beitritt und hat keinen Rückgabewert.
- void LogOut(string name): Teilt dem Server mit , dass der User „name“ den Chat verlässt und hat keinen Rückgabewert.
- List<Message> GetChats(int index): Der Client fragt beim Server eine Liste aller gesendeter Nachrichten (seit dem letzten Neustart des Servers) an, beginnend mit der index-ten Nachricht. Gibt dem Client diese Liste zurück.
- List<string> RefreshUserList(): Der Client fragt beim Server eine Liste aller derzeit am Chat teilnehmenden User an und erhält vom Server diese Liste zurück.

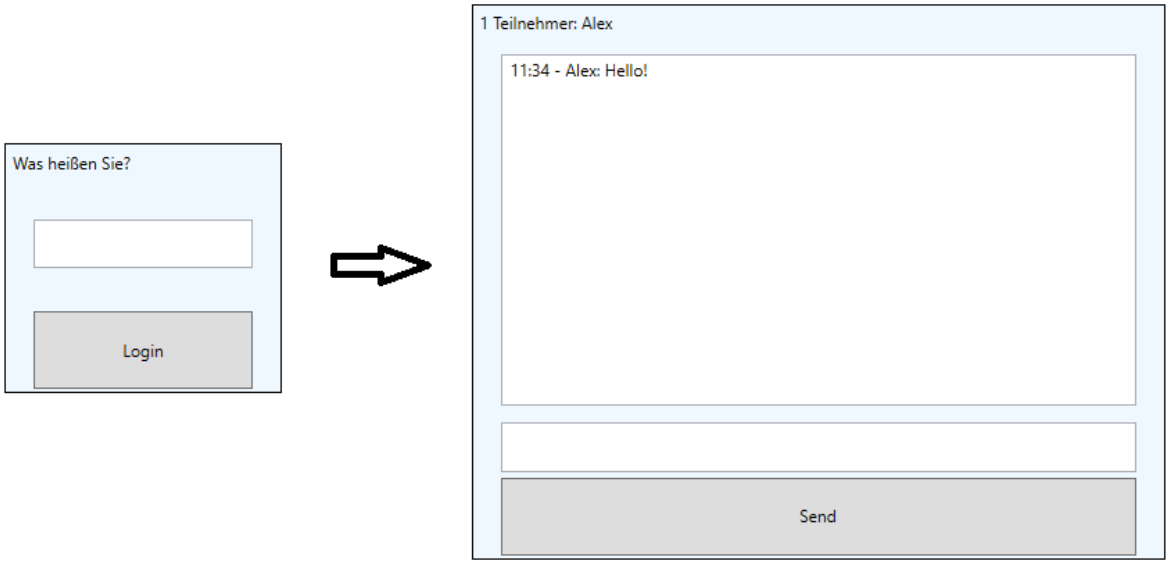
Die Klasse Message

Die Klasse Message modelliert eine einzelne Chatnachricht und besitzt folgende Attribute, für welche jeweils einfache get- und set-Methoden definiert werden:

- string user: Der User, von welchem die Nachricht gesendet wurde. Standardwert „Anonymous“.
- string text: Der Inhalt der Nachricht. Kein Standardwert.
- DateTime time: Der Zeitpunkt, zu welchem die Nachricht gesendet wurde. Standardwert ist der minimal definierbare Zeitpunkt (DateTime.MinValue).

Der Client im Detail

Das Hauptfenster der Anwendung, auch MainWindow genannt, setzt sich aus zwei User Controls zusammen: LoginUC und ChatViewUC. Zunächst wird nur die Eingabe für den Namen geöffnet, das eigentliche Chatfenster bleibt verborgen. Sobald ein Name eingetragen und auf „Login“ geklickt wurde, wird das Eingabefeld verborgen und das Chatfenster angezeigt.



Reihenfolge und Aussehen der User Controls

MVVMLight integriert das View-Model-Prinzip für WPF, welches Benutzeroberfläche und Programmlogik trennt. Es existieren zwei Klassen: MainViewModel enthält die Interaktionslogik für die gesamte Benutzeroberfläche, während ViewModelLocator die Verbindung mit der Oberfläche unterhält.

Konstruktor MainViewModel

```
public MainViewModel()
        {
            LoginVisibility = Visibility.Visible;
            ChatViewVisibility = Visibility.Collapsed;

            LoginCommand = new RelayCommand(LoginMethod);
            SendCommand = new RelayCommand(SendMethod);

            if (IsInDesignMode)
            {
                WindowTitle = "Chat Application (Design)";
            }
            else
            {
                WindowTitle = "Chat Application";
            }

            Application.Current.MainWindow.Closing += new
                CancelEventHandler(MainWindow_Closing);
        }
```

Zunächst wird das Login-Fenster mit der Namenseingabe als sichtbar gesetzt und das Chatfenster verborgen. Anschließend wird die Funktionalität für Login- und Senden-Button sowie das Closing-Event hinzugefügt. Sobald der User das Chatfenster schließt, wird er automatisch ausgeloggt.

Login-Funktion

```
public async void LoginMethod()
        {
            if (!string.IsNullOrEmpty(Username))
            {
                LoginVisibility = Visibility.Collapsed;
                ChatViewVisibility = Visibility.Visible;

                client = new IpcServiceClientBuilder<IChatService>()
                    .UseTcp(IPAddress.Loopback, 8000)
                    .Build();

                await client.InvokeAsync(x => x.Join(Username));

                var dispatcherTimer = new
                System.Windows.Threading.DispatcherTimer();
                dispatcherTimer.Tick += new EventHandler(RefreshMethod);
                dispatcherTimer.Interval = new TimeSpan(0, 0, 1);
                dispatcherTimer.Start();
            }
        }
```

Es wird zunächst geprüft, ob in der LoginUC ein Name eingegeben wurde. Ist dies der Fall, so wird diese versteckt und das eigentliche Chatfenster (ChatViewUC) angezeigt. Anschließend wird ein neuer IPC Service Client erzeugt, welcher das Interface IChatService implementiert und eine Verbindung zu einem Server aufbaut, welcher über TCP lokal auf Port 8000 erreichbar ist. Außerdem wird ein Dispatcher Timer gestartet, welcher asynchron jede Sekunde die Funktion RefreshMethod aufruft.

Refresh-Funktion

```
public async void RefreshMethod(object sender, EventArgs e)
        {
            List<string> userList = await client.InvokeAsync(x =>
                x.RefreshUserList());

            string users = userList.Count + " Teilnehmer:";
            foreach (string user in userList)
                users += " " + user;
            Userlist = users;

            int index;
            if (Messages == null) index = 0;
            else index = Messages.Count;

            var history = await client.InvokeAsync(x => x.GetChats(index));

            if (history == null) return;
            if (Messages == null)
            {
                Messages = new ObservableCollection<Message>(history);
                return;
            }
            foreach (var message in history) Messages.Add(message);
        }
```

Diese Funktion wird jede Sekunde asynchron vom Dispatcher aufgerufen. Zunächst wird die aktuelle Userliste angefordert und anschließend in der Benutzeroberfläche aktualisiert. Danach werden die seit dem letzten Refresh vom Server empfangenen Nachrichten geholt. Ist dies der erste Refresh (der Client hat sich also gerade erst neu eingeloggt), so werden alle Nachrichten angefordert. Falls die empfangene Liste leer ist, so wird die Methode verlassen. Falls vorher noch keine Nachrichten empfangen wurden, so wird die Observable Collection Messages angelegt, welche dann in der Benutzeroberfläche zu sehen ist. Ansonsten werden einfach die empfangenen Nachrichten an die bisherige Nachrichtenliste angehängt.

Der Server im Detail

Der Server setzt sich, neben dem oben beschriebenen Service Model, aus zwei Bestandteilen zusammen:

1. Der Klasse Program, welche die die Main-Funktion enthält.
2. Der Klasse ChatService, welche das Interface IChatService implementiert.

Die Klasse Program

```
private static IServiceCollection ConfigureServices(IServiceCollection services)
        {
            return services.AddIpc(builder =>
            {
                builder
                    .AddNamedPipe(options =>
                    {
                        options.ThreadCount = 2;
                    })
                    .AddService<IChatService, ChatService>();
            });
        }
```

Diese Funktion legt die Eigenschaften des IPC Services fest, welcher in der Main-Funktion (siehe unten erzeugt wird). So läuft die erstellte Named Pipe auf (maximal) 2 Threads. Weiterhin wird festgelegt, dass der Service das Interface IChatService in der Klasse ChatService implementiert.

```
static void Main(string[] args)
        {
            IServiceCollection services = ConfigureServices(new
                ServiceCollection());

            Console.WriteLine("Starting server...");

            new IpcServiceHostBuilder(services.BuildServiceProvider())
                .AddNamedPipeEndpoint<IChatService>(name: "pipeEndpoint", pipeName:
                    "chatPipe")
                .AddTcpEndpoint<IChatService>(name: "tcpEndpoint", ipEndpoint:
                    IPAddress.Loopback, port: 8000)
                .Build()
                .Run();
        }
```

In der main-Funktion wird der IPC Service Host erstellt, welcher später die Anfragen der Clients verarbeitet. Unter Verwendung des zuvor konfigurierten Services (siehe oben) wird ein Host gebaut, welcher über eine Named Pipe namens „ChatPipe“ sowie lokal über TCP auf dem Port 8000 erreichbar ist.

Die Klasse ChatService

Die Implementierung des Interfaces IChatService ist selbsterklärend. Es werden zwei statische Listen für alle Nachrichten seit dem letzten Neustart sowie für die aktuell angemeldeten User gehalten. In den Methoden wird jeweils auf diese zugegriffen.

Bewertung und Vergleich

Vorteile

- Das Konzept hinter IPC Service Framework ist leicht verständlich und schnell umzusetzen.
- Der Aufbau einer Anwendung mit IPC Service Framework ist vergleichsweise einfach.

Nachteile

- IPC Service Framework unterstützt keine Duplex-Verbindungen, d.h. der Server kann nicht unabhängig Daten an den Client senden. Somit ist keine „Push-Zustellung“ neuer Nachrichten möglich, wie sie für ein Chatprogramm ideal wäre. Dies ließe sich zwar durch die Aufrechterhaltung einer separaten Verbindung vom Server zum Client umgehen, eine ideale Lösung ist dies allerdings nicht. Die Integration von Duplex-Funktionalität wurde vom Autor in einer GitHub-Issue aufgegriffen (https://github.com/jacqueskang/IpcServiceFramework/issues/66), und es wurde kein baldiges Hinzufügen dieses Features in Aussicht gestellt.
- IPC Service Framework ist sehr schlecht bzw. so gut wie gar nicht dokumentiert. Es existieren lediglich die Quick-Start-Anleitung in der README.md des GitHub-Repos (https://github.com/jacqueskang/IpcServiceFramework), Antworten auf Issues und ein paar unvollständige Codebeispiele. Auch über Suchmaschinen lassen sich keine weiterführenden Informationen finden.
- IPC Service Framework wird nur sporadisch und hauptsächlich von einer Person entwickelt. Das Projekt ist zwar Open Source und andere Entwickler beteiligen sich daran, allerdings steht hinter dieser Lösung (noch) keine große Community, die das Projekt weiterführen könnte. Ein langfristiger Einsatz von IPC Service Framework ist daher mit großer Unsicherheit verbunden.

Fazit

Für kleine Projekte, bei welchen ausschließlich der Client Anfragen und Daten an den Server sendet und nicht umgekehrt, ist IPC Service Framework funktionell gut geeignet. Die Umsetzung ist schnell erledigt und das Konzept ist leicht zu verstehen.

Für größere Projekte oder solche, die Duplex-Unterstützung benötigen, ist IPC Service Framework allerdings nicht geeignet. Der Support für Duplex fehlt vollständig und wird laut Autor auch nicht demnächst folgen. Die größten Probleme sind jedoch die fehlende Dokumentation und die stets ungewisse Zukunft des Projekts. **Deshalb ist IPC Service Framework für Produktivanwendungen nicht zu empfehlen.**