Code

Note: The code here is from the updated code upon completion 10/4/2017.

# App

## App.config

<?xml version="1.0" encoding="utf-8" ?>

<configuration>

<startup>

<supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />

</startup>

<system.serviceModel>

<client>

<endpoint address="net.tcp://localhost:3000/MsgOverlay" binding="netTcpBinding"

bindingConfiguration="" contract="Interfaces.IntServices" name="clientEP"

kind="" endpointConfiguration="" />

</client>

</system.serviceModel>

</configuration>

## App.xaml

<Application x:Class="ChatWindow.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:ChatWindow"

Startup = "Application\_Startup"

Exit="Application\_Exit">

<Application.Resources>

</Application.Resources>

</Application>

## App.xaml.cs

using System;

using System.Collections.Generic;

using System.Configuration;

using System.Data;

using System.Linq;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Input;

using System.Runtime.InteropServices;

using System.Windows.Interop;

namespace ChatWindow

{

/// <summary>

/// Function definitions mentioned in App.xaml

/// Such as start / close

/// </summary>

public partial class App : Application

{

SemiTransOverlay m\_startWindow = null;

/// <summary>

/// When the application starts it will open the initial window.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void Application\_Startup(object sender, StartupEventArgs e)

{

// Create the startup window

m\_startWindow = new SemiTransOverlay();

}

/// <summary>

/// When the application ends/closes the user will be disconnected and the application

/// will be shut down.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void Application\_Exit(object sender, ExitEventArgs e)

{

//Redundant code.m\_startWindow.disconnect();

m\_startWindow.Close();

}

}

}

# Direct Message

## DirectMessage.xaml

<Window x:Class="ChatWindow.DirectMessage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:ChatWindow"

mc:Ignorable="d"

Title="DirectMessage" Height="300" Width="500" ShowInTaskbar="False" Background="#FFCCCCCC">

<Grid Margin="0,0,0,-76">

<TextBox x:Name="txtMsgToSend" HorizontalAlignment="Left" Height="44" Margin="10,215,0,0" TextWrapping="Wrap" VerticalAlignment="Top" Width="405" Background="#FFD8D8D8" BorderBrush="White" SelectionBrush="Black" KeyDown="keyDown"/>

<TextBox x:Name="txtDisplay" HorizontalAlignment="Left" Height="191" Margin="10,10,0,0" VerticalAlignment="Top" Width="472" BorderBrush="White" SelectionBrush="Black" Background="#FFD8D8D8" Focusable="False" ForceCursor="True" IsEnabled="False" TextChanged="TextChanged\_ScrollHandle"/>

<Image x:Name="btnSend" HorizontalAlignment="Left" Height="44" Margin="420,215,0,0" VerticalAlignment="Top" Width="62" Source="/Resources/ChatRoom/Send.png" MouseEnter="btnSend\_MouseEnter" MouseLeave="btnSend\_MouseLeave" ToolTip="Send" MouseDown="btnSend\_Press"/>

</Grid>

</Window>

## DirectMessage.xaml.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Shapes;

namespace ChatWindow

{

/// <summary>

/// Interaction logic for DirectMessage.xaml

/// </summary>

public partial class DirectMessage : Window

{

/// <summary>

/// The parent of this window.

/// </summary>

private SemiTransOverlay m\_parentWin = null;

/// <summary>

/// The client's username

/// </summary>

private string m\_username = null;

/// <summary>

/// The destination client's username

/// </summary>

private string m\_chatBuddy = null;

/// <summary>

/// Direct Message constructor when they're receiving a message from another client.

/// </summary>

/// <param name="a\_parent">Window : the parent window, semiTransOverlay</param>

/// <param name="a\_chatWith">string : the chat buddy's client username</param>

/// <param name="a\_currUser">string : the current client's username</param>

/// <param name="a\_newMsg">string : the message being received from the chat buddy</param>

public DirectMessage(Window a\_parent, string a\_chatWith, string a\_currUser, string a\_newMsg)

{

m\_parentWin = a\_parent as SemiTransOverlay;

this.Owner = m\_parentWin;

m\_username = a\_currUser;

m\_chatBuddy = a\_chatWith;

InitializeComponent();

this.Title = a\_chatWith + " Direct Message";

displayToUsr(a\_chatWith + ": " + a\_newMsg);

this.Show();

}

/// <summary>

/// Direct Message constructor when the current client is sending a message to another client.

/// </summary>

/// <param name="a\_parent">Window : the parent window, semiTransOverlay</param>

/// <param name="a\_chatWith">string : the chat message's destination / chat buddy</param>

/// <param name="a\_currUser">string : the current client's username</param>

public DirectMessage(Window a\_parent, string a\_chatWith, string a\_currUser)

{

m\_parentWin = a\_parent as SemiTransOverlay;

this.Owner = m\_parentWin;

m\_username = a\_currUser;

m\_chatBuddy = a\_chatWith;

InitializeComponent();

this.Title = a\_chatWith + " Direct Message";

this.Show();

}

/// <summary>

/// Send a message to the other user

/// </summary>

private void send()

{

string trimmed = txtMsgToSend.Text.Trim();

if (!string.IsNullOrEmpty(trimmed))

{

//Use the parent's functionality to send the message through the server.

m\_parentWin.sendToDestination(m\_chatBuddy, txtMsgToSend.Text);

displayToUsr(m\_username + ": " + txtMsgToSend.Text);

}

txtMsgToSend.Clear();

}

/// <summary>

/// Display a full message including username + message to the user

/// </summary>

/// <param name="a\_msg">string : a message in the format of username + message</param>

public void displayToUsr(string a\_msg)

{

txtDisplay.AppendText(a\_msg + "\n");

}

/// <summary>

/// Send Message Command on Mouse Down on the Image

/// </summary>

/// <param name="a\_imgSend">Image : when clicking on the image, send the information</param>

/// <param name="e">Unused in this function</param>

private void btnSend\_Press(object a\_imgSend, MouseButtonEventArgs e)

{

send();

}

/// <summary>

/// When the mouse hovers over the clickable area the image changes

/// </summary>

/// <param name="a\_imgSend">Image : the area that is being hovered over</param>

/// <param name="e">Unused in this function</param>

private void btnSend\_MouseEnter(object a\_imgSend, MouseEventArgs e)

{

btnSend.Source = new BitmapImage(new Uri(@"/Resources/Chatroom/SendOnClick.png", UriKind.Relative));

}

/// <summary>

/// When the mouse leaves the clickable area the image changes back to the original

/// </summary>

/// <param name="a\_imgSend">Image : the area that is being watched</param>

/// <param name="e">Unused in this function</param>

private void btnSend\_MouseLeave(object a\_imgSend, MouseEventArgs e)

{

btnSend.Source = new BitmapImage(new Uri(@"/Resources/Chatroom/Send.png", UriKind.Relative));

}

/// <summary>

/// Send message on pressing enter while within the textbox

/// </summary>

/// <param name="a\_txtBox"></param>

/// <param name="a\_keyPressed">KeyEventsArgs : looking at the key pressed on the keyboard</param>

private void keyDown(object a\_txtBox, KeyEventArgs a\_keyPressed)

{

if (a\_keyPressed.Key == Key.Return)

{

send();

}

}

/// <summary>

/// Makes sure to keep the scroll at the bottom to show the most recent messages

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void TextChanged\_ScrollHandle(object sender, TextChangedEventArgs e)

{

txtDisplay.ScrollToEnd();

return;

}

}

}

# Global Chat

## GlobalChat.xaml

<Window x:Class="ChatWindow.GlobalChat"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:ChatWindow"

mc:Ignorable="d"

Title="Global Chat" Height="375" Width="550" ResizeMode="NoResize" ShowInTaskbar="False" Background="#FFCCCCCC" Closed="toggleGlobal">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="433\*"/>

<ColumnDefinition Width="111\*"/>

</Grid.ColumnDefinitions>

<TextBox x:Name="txtMsgToSend" HorizontalAlignment="Left" Height="56" Margin="10,280,0,0" TextWrapping="Wrap" VerticalAlignment="Top" Width="417" Grid.ColumnSpan="2" Background="#FFD8D8D8" BorderBrush="White" SelectionBrush="Black" KeyDown="keyDown"/>

<TextBox x:Name="txtDisplay" HorizontalAlignment="Left" Height="230" Margin="10,20,0,0" VerticalAlignment="Top" Width="417" Grid.ColumnSpan="2" BorderBrush="White" SelectionBrush="Black" Background="#FFD8D8D8" ForceCursor="True" VerticalScrollBarVisibility="Auto" IsReadOnlyCaretVisible="True" ScrollViewer.CanContentScroll="True" TextChanged="TextChanged\_ScrollHandle" IsEnabled="False"/>

<Label x:Name="lblPeople" Content="Active" Grid.Column="2" HorizontalAlignment="Left" Margin="1,0,0,0" VerticalAlignment="Top" RenderTransformOrigin="0.432,1.7" ToolTip="Add friends here!" FontFamily="Yu Gothic UI Light"/>

<Image x:Name="btnSend" HorizontalAlignment="Left" Height="44" Margin="25,286,0,0" VerticalAlignment="Top" Width="62" Source="/Resources/ChatRoom/Send.png" MouseEnter="btnSend\_MouseEnter" MouseLeave="btnSend\_MouseLeave" ToolTip="Send" MouseDown="btnSend\_Press" Grid.Column="2"/>

<ListBox x:Name="lstNames" Grid.Column="2" HorizontalAlignment="Left" Height="230" Margin="1,20,0,0" VerticalAlignment="Top" Width="100" Background="#FFD8D8D8" BorderBrush="White" FontFamily="Yu Gothic UI Light" FontSize="14">

<ListBox.Resources>

<ContextMenu x:Key="MyElementMenu">

<MenuItem Header="Message" Click="openDirectMessage"/>

<MenuItem Header="Add User"/>

</ContextMenu>

<Style TargetType="{x:Type ListBoxItem}">

<Setter Property="ContextMenu" Value="{StaticResource MyElementMenu}"/>

</Style>

</ListBox.Resources>

</ListBox>

</Grid>

</Window>

## GlobalChat.xaml.cs

using System;

using System.Collections.Generic;

using System.Collections.ObjectModel;

using System.ComponentModel;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Controls.Primitives;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Shapes;

namespace ChatWindow

{

/// <summary>

/// Interaction logic for GlobalChat.xaml

/// </summary>

public partial class GlobalChat : Window

{

/// <summary>

/// The initializer of the parent window to this class

/// </summary>

private SemiTransOverlay m\_parentWin = null;

/// <summary>

/// Initializer of the username

/// </summary>

private string m\_username = null;

/// <summary>

/// Initializer of the observable collection that will be used as the client's username list

/// provided by the server/semi-transparent overlay

/// </summary>

private ObservableCollection<string> m\_userList;

/// <summary>

/// Constructor for this window, sets up the list and parent window

/// </summary>

/// <param name="a\_parent">Window : the parent window to this window, semiTransOverlay</param>

/// <param name="a\_user">string : the username of the current client</param>

/// <param name="a\_globalList">List[string] : all the user's that have been connected to the server</param>

public GlobalChat(Window a\_parent, string a\_user, List<string> a\_globalList)

{

m\_parentWin = a\_parent as SemiTransOverlay;

this.Owner = m\_parentWin;

m\_username = a\_user;

InitializeComponent();

setListValues(a\_globalList);

this.Show();

}

/// <summary>

/// Updates a new user connecting by adding them to the <paramref name="m\_userList"/>

/// </summary>

/// <param name="a\_connected">string : the username of who had just connected</param>

public void updateJoin(string a\_connected)

{

if (a\_connected == null)

{

return;

}

m\_userList.Add(a\_connected);

}

/// <summary>

/// Updates the client by removing the user who had just left using <paramref name="m\_userList"/>

/// </summary>

/// <param name="a\_disconnected">string : the user who has disconnected.</param>

public void updateDisconnect(string a\_disconnected)

{

if(a\_disconnected == null)

{

return;

}

m\_userList.Remove(a\_disconnected);

}

/// <summary>

/// Sets the initial list of user's that are connected.

/// </summary>

/// <param name="a\_connections">List[string] : the initial list of client username's connected to the server</param>

public void setListValues(List<string> a\_connections)

{

if (a\_connections == null)

{

return;

}

// Initializes the collection used by the listbox

m\_userList = new ObservableCollection<string> (a\_connections);

lstNames.ItemsSource = m\_userList;

lstNames.Items.SortDescriptions.Add(new System.ComponentModel.SortDescription("", System.ComponentModel.ListSortDirection.Ascending));

}

/// <summary>

/// Displays message to the user

/// </summary>

/// <param name="a\_msg">string : the message to be displayed</param>

public void displayToUsr(string a\_msg)

{

txtDisplay.AppendText(a\_msg + "\n");

}

/// <summary>

/// When the mouse hovers over the clickable area the image changes

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void btnSend\_MouseEnter(object sender, MouseEventArgs e)

{

btnSend.Source = new BitmapImage(new Uri(@"/Resources/Chatroom/SendOnClick.png", UriKind.Relative));

}

/// <summary>

/// When the mouse leaves the clickable area the image changes back to the original

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void btnSend\_MouseLeave(object sender, MouseEventArgs e)

{

btnSend.Source = new BitmapImage(new Uri(@"/Resources/Chatroom/Send.png", UriKind.Relative));

}

/// <summary>

/// Send Message Command on Mouse Down on the Image

/// </summary>

/// <param name="a\_imgSend">Image : when clicking on the image, send the information</param>

/// <param name="e">Unused in this function</param>

private void btnSend\_Press(object a\_imgSend, MouseButtonEventArgs e)

{

send();

}

/// <summary>

/// Manages the sending of messages

/// </summary>

private void send()

{

string trimmed = txtMsgToSend.Text.Trim();

if (!String.IsNullOrEmpty(trimmed))

{

m\_parentWin.sendAllConnection(txtMsgToSend.Text);

displayToUsr(m\_username + ": " + txtMsgToSend.Text);

}

txtMsgToSend.Clear();

}

/// <summary>

/// When pressing the enter key to send a message.

/// </summary>

/// <param name="sender"></param>

/// <param name="a\_keyPressed">KeyEventsArgs : looking at the key pressed on the keyboard</param>

private void keyDown(object sender, KeyEventArgs a\_keyPressed)

{

if (a\_keyPressed.Key == Key.Return)

{

send();

}

}

/// <summary>

/// When the global chat is closed, it has to be known to the parentWin that it no longer

/// exists :: therefore, it can start a new instance the correct way through the G button.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void toggleGlobal(object sender, EventArgs e)

{

//upon window closed

m\_parentWin.globalStatus(false);

}

/// <summary>

/// Open the direct message chat for the correct selected user, and allow the parent

/// to handle creation.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void openDirectMessage(object sender, RoutedEventArgs e)

{

if (lstNames.SelectedIndex == -1)

{

return;

}

string userDest = lstNames.SelectedItem.ToString();

// parent window opening direct message window chat.

m\_parentWin.openDM(userDest);

}

/// <summary>

/// Make sure to keep the scroll at the bottom to show the most recent recieved message

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void TextChanged\_ScrollHandle(object sender, TextChangedEventArgs e)

{

txtDisplay.ScrollToEnd();

return;

}

}

}

# Log In

## Login.xaml

<Window x:Name="winLogin" x:Class="ChatWindow.LogIn"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:ChatWindow"

mc:Ignorable="d"

Title="LogIn" Height="375" Width="310" WindowStyle="None" Background="#FFCCCCCC"

WindowStartupLocation="CenterOwner" ShowInTaskbar="False" ResizeMode="NoResize">

<Grid x:Name="winLogIn">

<Image x:Name="imgIcon" Height="135" Margin="61,10,62,0" VerticalAlignment="Top" Source="/Resources/Icon.png" HorizontalAlignment="Center"/>

<TextBox x:Name="txtUserName" TextAlignment="Center" HorizontalAlignment="Center" Height="28" Margin="64,179,66,168" TextWrapping="Wrap" VerticalAlignment="Center" Width="180" FontFamily="Tahoma" FontSize="14" Background="#FFEAEAEA" AutomationProperties.IsRequiredForForm="True" AutomationProperties.HelpText="Username" Text="Username" Foreground="#FF666666" GotFocus="txtUserName\_GotFocus" LostFocus="txtUserName\_LostFocus" SelectionBrush="Black" BorderBrush="#FF727272"/>

<TextBox x:Name="txtPassword" TextAlignment="Center" HorizontalAlignment="Center" Height="28" Margin="64,226,67,121" TextWrapping="Wrap" VerticalAlignment="Center" Width="179" FontFamily="Tahoma" FontSize="14" Background="#FFEAEAEA" AutomationProperties.IsRequiredForForm="True" Foreground="#FF666666" Text="Password" KeyDown="txtPassword\_KeyDown" BorderBrush="#FF727272" SelectionBrush="Black"/>

<Image x:Name="btnLogin" HorizontalAlignment="Left" Height="64" Margin="112,269,0,0" VerticalAlignment="Top" Width="96" Source="/Resources/Login.png" MouseDown="btnLogin\_MouseDown"/>

<Button x:Name="btnReg" Content="New User? Register." HorizontalAlignment="Left" Height="27" Margin="164,338,0,0" VerticalAlignment="Top" Width="136" FontFamily="tahoma" BorderBrush="{x:Null}" BorderThickness="0,0,0,2" Background="#FFCCCCCC"/>

</Grid>

</Window>

## Login.xaml.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Shapes;

namespace ChatWindow

{

/// <summary>

/// Interaction logic for LogIn.xaml

/// </summary>

public partial class LogIn : Window

{

/// <summary>

/// SemiTransOverlay : maintain a reference to the parent window to use function calls,

/// as well as custom events.

/// </summary>

private SemiTransOverlay m\_parentWin = null;

/// <summary>

/// Custom events to gain the username for the semi-transparent overlay parent window.

/// </summary>

public event EventHandler<List<String>> RaiseCustomEvent;

/// <summary>

/// Login constructor, sets up the parent window as well as other components attached to the window

/// at the start.

/// </summary>

/// <param name="a\_parent"> Window : the parent window being passed in at creation. [SemiTransOverlay]</param>

public LogIn(Window a\_parent)

{

// must have the semi-transparent overlay as parent for each window created.

m\_parentWin = a\_parent as SemiTransOverlay;

this.Owner = m\_parentWin;

InitializeComponent();

// settings for the window that may be changed in the future.

txtPassword.IsReadOnly = true;

txtPassword.IsEnabled = false;

txtPassword.Focusable = false;

// size of the window based upon the parent's dimensions (screen dimensions)

this.Left = m\_parentWin.Left + (m\_parentWin.Width - this.ActualWidth) / 2;

this.Top = m\_parentWin.Top + (m\_parentWin.Height - this.ActualHeight) / 2;

this.Show();

}

/// <summary>

/// Initial focus of the chatbox, the text normally eill say Username

/// however, when the textbox is focused it should be cleared..

/// </summary>

/// <param name="a\_sender">login window is the sender.</param>

/// <param name="e"> Unused in this function</param>

private void txtUserName\_GotFocus(object a\_sender, RoutedEventArgs e)

{

if (txtUserName.Text == "Username")

{

txtUserName.Clear();

}

txtUserName.SelectionStart = txtUserName.Text.Length;

}

/// <summary>

/// Change username to lower before being compared and sent to other windows.

/// </summary>

/// <param name="a\_sender"> TextBox : username's textbox</param>

/// <param name="e"> Unused in this function</param>

private void txtUserName\_LostFocus(object a\_sender, RoutedEventArgs e)

{

txtUserName.Text = txtUserName.Text.ToLower();

}

/// <summary>

/// On a keydown event this will begin. Looking for the enter press to proc

/// the load in event.

/// </summary>

/// <param name="a\_sender"> TextBox : username text box</param>

/// <param name="e"> keyboard keys: "Key" entered to the keyboard</param>

private void txtPassword\_KeyDown(object a\_sender, System.Windows.Input.KeyEventArgs e)

{

if (e.Key == Key.Enter)

{

loginAttempt();

}

}

/// <summary>

/// When clicking on the image, this event will occur. Then there will be

/// a call to the login attempt.

/// </summary>

/// <param name="a\_sender">Image : login button image. </param>

/// <param name="e"> Unused in this function</param>

private void btnLogin\_MouseDown(object a\_sender, MouseButtonEventArgs e)

{

loginAttempt();

}

/// <summary>

/// Function to send back the user information to the parent window.

/// </summary>

private void loginAttempt()

{

List<String> userInfo = new List<String>();

userInfo.Add(txtUserName.Text.ToLower());

// Custom event goes back to the event handler within SemiTransOverlay

RaiseCustomEvent(this, userInfo);

this.Close();

}

}

}

# Semi Transparent Overlay

## SemiTransOverlay.xaml

<Window x:Name="winSemiTrans" x:Class="ChatWindow.SemiTransOverlay"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:ChatWindow"

mc:Ignorable="d"

Title="semiTransOverlay" Height="300" Width="300" WindowStartupLocation="CenterScreen" ResizeMode="NoResize" WindowState="Maximized" WindowStyle="None" AllowsTransparency="True" ShowInTaskbar="False">

<Window.Background>

<SolidColorBrush Opacity="0.7" Color="Black"/>

</Window.Background>

<Grid x:Name="ImgControls">

</Grid>

</Window>

## SemiTransOverlay.xaml.cs

using Interfaces;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Runtime.InteropServices;

using System.ServiceModel;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Interop;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Shapes;

namespace ChatWindow

{

/// <summary>

/// Interaction logic for semiTransOverlay.xaml

/// </summary>

public partial class SemiTransOverlay : Window

{

/// <summary>

/// Class containing username and password indices for use throughout the window.

/// </summary>

private class IndexOfConnectionInfo

{

public int username = 0;

public int password = 1;

}

/// <summary>

/// Contains session information based on what state the program is in:

/// (1) connected to global chat

/// (2) if the window is focused

/// (3) if the window is toggled

/// (4) the clients connected to the session

/// (5) current client's username

/// etc.

/// </summary>

private class SessionStatus

{

public bool isVisible = true;

public bool isWinFocus;

public bool isGlobalOpen;

public GlobalChat globalChatbox = null;

public List<string> globalList = new List<string>();

public string currentUser = null;

}

/// <summary>

/// IntServices : the style of services allowed by the m\_serverConnection

/// </summary>

public static IntServices m\_serverConnection;

/// <summary>

/// DuplexChannelFactory[IntServices] : allowing the client functionality of sending messages/connecting |

/// attaching the client to the services provided by the server<paramref name="m\_serverConnection"/>.

/// </summary>

private static DuplexChannelFactory<IntServices> m\_channelFactory;

/// <summary>

/// List[string] : list of the clients connected to the server. Initialized through first successful connection to the server.

/// </summary>

List<string> m\_connected = new List<string>();

/// <summary>

/// SessionStatus : containing variables to maintain the current states / information about the session.

/// </summary>

SessionStatus m\_sessionStatus;

/// <summary>

/// Dictionary[string, DirectMessage] : containing each direct message[DirectMessage (window)] to the recipient[string].

/// </summary>

Dictionary<string, DirectMessage> m\_chatWindows = new Dictionary<string, DirectMessage>();

/// <summary>

/// Constructor for the semi-transparent overlay window:

/// (1) Involves starting an instance of the Login window

/// (2) Connecting to the server.

/// (3) Initializing the session status variables.

/// </summary>

public SemiTransOverlay()

{

m\_channelFactory = new DuplexChannelFactory<IntServices>(new ImpClient(), "clientEP");

m\_serverConnection = m\_channelFactory.CreateChannel();

InitializeComponent();

// Initialize session variables to the appropriate state

m\_sessionStatus = new SessionStatus();

m\_sessionStatus.isWinFocus = true;

this.Show();

openLogin();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START : Login Handling \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// Creates a new login window for the client to attempt to login.

/// Custom Event is raised to handle clicking the login button to pass back information.

/// </summary>

private void openLogin()

{

LogIn childForm = new LogIn(this);

// New custom event, declared as attemptConnection which will obtain a List of strings

childForm.RaiseCustomEvent += new EventHandler<List<String>>(attemptConnection);

}

/// <summary>

/// Custom event handling the task of when the user requests to login to the server.

/// (1) Checks the username

/// (2) Connects to the server on success

/// </summary>

/// <param name="a\_sender">object : the login window itself</param>

/// <param name="a\_userInfo"> List[string] : the user information gained from the login window</param>

public void attemptConnection(object a\_sender, List<String> a\_userInfo)

{

List<string> userInfo = a\_userInfo;

IndexOfConnectionInfo index = new IndexOfConnectionInfo();

string name = userInfo[index.username];

List<string> temp = new List<string>();

//Determining if the user is able to login with the chosen username

int loginStatus = m\_serverConnection.login(userInfo[index.username], ref temp);

m\_sessionStatus.globalList = temp;

if (loginStatus == 1)

{

m\_sessionStatus.currentUser = name;

openWelcome();

return;

}

else if (loginStatus == 0)

{

MessageBox.Show("Duplicate Login, Pre-existing in server.");

Console.WriteLine("Log in failure. Duplicate in system.");

openLogin();

return;

}

else

{

MessageBox.Show("Error connecting to server. Please try again later.");

openLogin();

return;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END : Login Handling \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START : Global Chat \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// Creates a new custome event for the Welcome window in order to

/// open the global chat automatically on the Welcome window's closure.

/// </summary>

private void openWelcome()

{

Welcome childWindow = new Welcome(this, m\_sessionStatus.currentUser);

childWindow.RaiseCustomEvent += new EventHandler<bool>(openGlobalChat);

}

/// <summary>

/// Open the first instance of the global chat for the user.

/// </summary>

/// <param name="sender"> Unused in this function. </param>

/// <param name="a\_isSuccess">bool : if the window was closed</param>

private void openGlobalChat(object sender, bool a\_isSuccess)

{

if (a\_isSuccess)

{

// set up a new instance of the global chat

newInstGlobal();

// connect the user to the server list, and send the update to each client

// that is already connected.

m\_serverConnection.newClientConnection(m\_sessionStatus.currentUser);

setUpButtons();

}

else

{

return;

}

}

/// <summary>

/// Sets up a new instance of the global chat connected to the parent overlay.

/// Updates the session information

/// </summary>

private void newInstGlobal()

{

m\_sessionStatus.globalChatbox = new GlobalChat(this, m\_sessionStatus.currentUser, m\_sessionStatus.globalList);

m\_sessionStatus.isGlobalOpen = true;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END : Global Chat \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START : Set Up Overlay \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// Buttons that are bound to certain aspects of the screen will be set up here with the appropriate images

/// and functionality behind them.

/// </summary>

private void setUpButtons()

{

Image btnGlobalChat = new Image();

Image btnDirectMsg = new Image();

Image btnExit = new Image();

// function for appropriately creating buttons

btnSettings(btnExit, "/Resources/Exit.png", 10, 1);

btnSettings(btnGlobalChat, "/Resources/InformationOverlay/NewGlobal.png", 10, 2);

btnSettings(btnDirectMsg, "/Resources/InformationOverlay/NewChat.png", 10, 3);

// Adding the buttons to the Grid layout on the semi-transparent overlay.

ImgControls.Children.Add(btnGlobalChat);

ImgControls.Children.Add(btnDirectMsg);

ImgControls.Children.Add(btnExit);

// Create the functionality

btnGlobalChat.MouseDown += new MouseButtonEventHandler(btnGlobalChat\_MouseDown);

// btnDirectMsg.MouseDown += new MouseButtonEventHandler(btnDirectMsg\_MouseDown);

btnExit.MouseDown += new MouseButtonEventHandler(btnExit\_MouseDown);

}

/// <summary>

/// Sets the functionality and design of each "button" | really an image that is clickable.

/// </summary>

/// <param name="a\_img">Image : the image that is being manipulated in the function</param>

/// <param name="a\_src">string : the file path in the resource folder to the appropriate image</param>

/// <param name="a\_buffer">int : used as the amount of space between the picture and the next</param>

/// <param name="a\_imgIndex">int : used as an index for each image to move to certain positioning on the screen</param>

private void btnSettings(Image a\_img, string a\_src, int a\_buffer, int a\_imgIndex)

{

a\_img.Stretch = Stretch.Fill;

a\_img.HorizontalAlignment = HorizontalAlignment.Left;

a\_img.VerticalAlignment = VerticalAlignment.Top;

a\_img.Height = 65;

a\_img.Width = 65;

a\_img.Margin = new Thickness(this.Width - (a\_imgIndex \* a\_img.Width + a\_buffer), this.Height - (a\_img.Height + a\_buffer), 0, 0);

BitmapImage img = new BitmapImage();

img.BeginInit();

img.UriSource = new Uri(a\_src, UriKind.Relative);

img.EndInit();

a\_img.Source = img;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END : Set Up Overlay \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START : Overlay Functionality \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// toggle application useage, when the client uses the hotkeys it will either change the state of the program to "on" or "off"

/// </summary>

public void toggleApplication()

{

if (m\_sessionStatus.isVisible)

{

this.Hide();

// Windows that are "owned" by the semi-transparent overlay window will become hidden

foreach (Window owned in this.OwnedWindows)

{

owned.Hide();

}

m\_sessionStatus.isVisible = false;

}

else

{

this.Show();

// Windows that are "owned" by the semi-transparent overlay window will become visible

foreach (Window owned in this.OwnedWindows)

{

owned.Show();

}

m\_sessionStatus.isVisible = true;

}

}

/// <summary>

/// When clicking on the disconnect/exit button use a callback function to the

/// server to disconnect the user.

/// </summary>

/// <param name="a\_sender">Image : the clickable image representing the disconnect button</param>

/// <param name="e">Unused in this function</param>

private void btnExit\_MouseDown(object a\_sender, MouseButtonEventArgs e)

{

//Disconnect the user from the application and shut down the application

disconnect();

System.Windows.Application.Current.Shutdown();

}

/// <summary>

/// When clicking on the global chat, it will toggle the chat being minimized / visible to the user,

/// if the user had closed the chat it will open the chat entirely.

/// </summary>

/// <param name="a\_sender">Image : the clickable image representing the global chat button</param>

/// <param name="e">Unused in this function.</param>

void btnGlobalChat\_MouseDown(object a\_sender, MouseEventArgs e)

{

// Minimize global chat / toggle

if (m\_sessionStatus.isGlobalOpen == false)

{

//Open Global Chat back up

newInstGlobal();

return;

}

if (m\_sessionStatus.globalChatbox.WindowState == WindowState.Minimized)

{

m\_sessionStatus.globalChatbox.WindowState = WindowState.Normal;

}

else

{

m\_sessionStatus.globalChatbox.WindowState = WindowState.Minimized;

}

}

/// <summary>

/// Changing the global instance in the session status variable(s)

/// </summary>

/// <param name="a\_isOpen">bool : containing the value if the Global window is open or not.</param>

public void globalStatus(bool a\_isOpen)

{

m\_sessionStatus.isGlobalOpen = a\_isOpen;

if (a\_isOpen == false)

{

m\_sessionStatus.globalChatbox = null;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END : Overlay Functionality \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START : Direct Message Connection \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// When requesting to send a direct message to a client, the user will open

/// a new window with the destination client.

/// </summary>

/// <param name="a\_dest">string : the username of the destination of the message</param>

public void openDM(string a\_dest)

{

DirectMessage childWindow = new DirectMessage(this, a\_dest, m\_sessionStatus.currentUser);

m\_chatWindows.Add(a\_dest, childWindow);

}

/// <summary>

/// When receiving a message from a user, this will check if the chat window already exists in the

/// <paramref name="m\_chatWindows"/> or if it has to create a new chat window.

/// </summary>

/// <param name="a\_from">string : username of who the message is from</param>

/// <param name="a\_msg">string : the message that is being received</param>

public void incomingMessage(string a\_from, string a\_msg)

{

//check if window is already opened.

foreach (var chat in m\_chatWindows)

{

if(a\_from == chat.Key)

{

//Display the message

m\_chatWindows[a\_from].displayToUsr(a\_from + ": " + a\_msg);

return;

}

}

DirectMessage childWindow = new DirectMessage(this, a\_from, m\_sessionStatus.currentUser, a\_msg);

m\_chatWindows.Add(a\_from, childWindow);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END: Direct Message Connection \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START : Client Requests \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// Requests the server to send a message to a specified destination username, from

/// the current client username.

/// </summary>

/// <param name="a\_dest">string : the destination of where the message is going.</param>

/// <param name="a\_msg">string : the message of what the user wants to send.</param>

///

public void sendToDestination(string a\_dest, string a\_msg)

{

m\_serverConnection.sendToUser(m\_sessionStatus.currentUser, a\_dest, a\_msg);

}

/// <summary>

/// Adding a new user to the connection list.

/// </summary>

/// <param name="a\_newUser">string : the username of the connecting user.</param>

public void addNewConnectionList(string a\_newUser)

{

if (m\_sessionStatus.isGlobalOpen)

{

m\_sessionStatus.globalList.Add(a\_newUser);

m\_sessionStatus.globalChatbox.updateJoin(a\_newUser);

m\_sessionStatus.globalChatbox.displayToUsr(" " +a\_newUser + " joined the conversation!");

}

}

/// <summary>

/// Sending the message to all clients connected to the server

/// </summary>

/// <param name="a\_msg">string : message being sent to the global chat</param>

public void sendAllConnection(string a\_msg)

{

m\_serverConnection.sendToAll(m\_sessionStatus.currentUser, a\_msg);

}

/// <summary>

/// Updates the client's user list since someone has disconnected from the chat.

/// </summary>

/// <param name="a\_disconnected">string : username that has disconnected from the server</param>

public void removeFromConnected(string a\_disconnected)

{

//communicate with the global chat window's listbox

m\_sessionStatus.globalList.Remove(a\_disconnected);

m\_sessionStatus.globalChatbox.updateDisconnect(a\_disconnected);

m\_sessionStatus.globalChatbox.displayToUsr(" " + a\_disconnected + " has left the conversation.");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END: Client Requests \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START: Server Responses \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// Receiving the message from the server to the global chat

/// </summary>

/// <param name="a\_message">string : the message to display that was sent to the user</param>

public void messagePass(string a\_message)

{

if (m\_sessionStatus.isGlobalOpen)

{

// TODO :: This text should probably have it's own formatting - BLACK

m\_sessionStatus.globalChatbox.displayToUsr(a\_message);

}

}

/// <summary>

/// Requesting a disconnect from the server.

/// </summary>

public void disconnect()

{

m\_serverConnection.disconnectionHandle(m\_sessionStatus.currentUser);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END: Server Responses \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\*Please Note:

\* The below code is completely functional, however it will not function with

\* multiple instances of the ChatWindow running since the first instance would have

\* already used the CTRL+Q hotkey. This will make conflicts with any instance created

\* after it.

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START: Hotkey Declaration \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

[DllImport("User32.dll")]

private static extern bool RegisterHotKey([In] IntPtr hWnd, [In] int id, [In] uint fsModifiers, [In] uint vk);

[DllImport("User32.dll")]

private static extern bool UnregisterHotKey([In] IntPtr hWnd, [In] int id);

private HwndSource \_source;

private const int HOTKEY\_ID = 9000;

/// <summary>

/// When the application starts up, this function will start up using the application as the

/// argument "e"

/// </summary>

/// <param name="e">This full application</param>

protected override void OnSourceInitialized(EventArgs e)

{

base.OnSourceInitialized(e);

var helper = new WindowInteropHelper(this);

\_source = HwndSource.FromHwnd(helper.Handle);

\_source.AddHook(HwndHook);

RegisterHotKey();

}

/// <summary>

/// When the application closes, ensures that each hotkey bound will be removed.

/// </summary>

/// <param name="e">This full application</param>

protected override void OnClosed(EventArgs e)

{

\_source.RemoveHook(HwndHook);

\_source = null;

UnregisterHotKey();

base.OnClosed(e);

}

/// <summary>

/// Registers the hotkey CTRL+Q through hexadecimal values through Windows messages.

/// </summary>

private void RegisterHotKey()

{

var helper = new WindowInteropHelper(this);

//virtual key code "reminders" - these are the codes for Q and CTRL

const uint VK\_QKEY = 0x51;

const uint MOD\_CTRL = 0x0002;

if (!RegisterHotKey(helper.Handle, HOTKEY\_ID, MOD\_CTRL, VK\_QKEY))

{

throw new NotImplementedException();

}

}

/// <summary>

/// Unregisters the hotkey ( on the closure of the application ).

/// </summary>

private void UnregisterHotKey()

{

var helper = new WindowInteropHelper(this);

UnregisterHotKey(helper.Handle, HOTKEY\_ID);

}

/// <summary>

/// Windows hotkey "break" or notification. This will toggle the application if the hotkey is correct.

/// </summary>

/// <param name="hwnd">Windows stuff</param>

/// <param name="msg">int : looking for the value to be successful["posted"]</param>

/// <param name="wParam">IntPtr : the full hotkey ID</param>

/// <param name="lParam">Windows stuff</param>

/// <param name="handled">bool : to see if the hotkey was successfully read in and handled properly</param>

/// <returns>IntPtr</returns>

private IntPtr HwndHook(IntPtr hwnd, int msg, IntPtr wParam, IntPtr lParam, ref bool handled)

{

// WM\_HOTKEY is "posted" when the message is the registered hotkey press.

const int WM\_HOTKEY = 0x0312;

switch (msg)

{

case WM\_HOTKEY:

switch (wParam.ToInt32())

{

case HOTKEY\_ID:

//hot key successfully pressed.

toggleApplication();

handled = true;

break;

}

break;

}

return IntPtr.Zero;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END: Hotkey Declaration \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

}

# Welcome

## Welcome.xaml

<Window x:Name="winWelcome" x:Class="ChatWindow.Welcome"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:ChatWindow"

mc:Ignorable="d"

Title="Welcome" Height="375" Width="310" WindowStyle="None" ResizeMode="NoResize" ShowInTaskbar="False"

WindowStartupLocation="CenterOwner" Background="#FFCCCCCC">

<Grid>

<Button x:Name="btnCloseWelcome" Content="Close" HorizontalAlignment="Center" Margin="0,332.847,0,0" VerticalAlignment="Top" Width="203.358" Height="35" BorderBrush="#FF666666" Background="#FF666666" Foreground="White" FontFamily="Yu Gothic UI Semibold" FontSize="18" UseLayoutRounding="False" RenderTransformOrigin="0.5,0.5" Padding="0,0,1,1" Click="btnCloseWelcome\_Click"/>

<Label x:Name="lblWelcome" Content="Welcome" HorizontalAlignment="Center" Margin="0,200,0,0" VerticalAlignment="Top" VerticalContentAlignment="Center" HorizontalContentAlignment="Center" FontFamily="Tahoma" FontSize="20" Foreground="#FF666666"/>

<Image x:Name="imgWelcomeIcon" HorizontalAlignment="Center" Height="130" VerticalAlignment="Top" Width="170" Margin="0,20,0,0" Source="/Resources/Icon.png"/>

<Label x:Name="lblUserWelcome" Content="" HorizontalAlignment="Center" Margin="0,245,0,0" VerticalAlignment="Top" VerticalContentAlignment="Center" HorizontalContentAlignment="Center" FontFamily="Tahoma" FontSize="20" Foreground="#FF666666"/>

</Grid>

</Window>

## Welcome.xaml.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Shapes;

namespace ChatWindow

{

/// <summary>

/// Interaction logic for Welcome.xaml

/// </summary>

public partial class Welcome : Window

{

/// <summary>

/// The custom event to send a notification back to the parent window

/// so the overlay can take care of another action.

/// </summary>

public event EventHandler<bool> RaiseCustomEvent;

/// <summary>

/// The parent window to the current window being used [welcome window]

/// </summary>

private SemiTransOverlay m\_parentWin = null;

/// <summary>

/// Welcome window constructor, simply sets the screen to welcome the current client to the application

/// </summary>

/// <param name="a\_parent">Window : the window that this belongs to</param>

/// <param name="a\_username">string : the username of the current client</param>

public Welcome(Window a\_parent, string a\_username)

{

m\_parentWin = a\_parent as SemiTransOverlay;

this.Owner = m\_parentWin;

InitializeComponent();

lblUserWelcome.Content = a\_username;

this.Show();

}

/// <summary>

/// When the welcome window closes, sets off another event.

/// </summary>

/// <param name="a\_btn">object : a button that had been clicked</param>

/// <param name="e"></param>

private void btnCloseWelcome\_Click(object a\_btn, RoutedEventArgs e)

{

welcomeClosed();

}

/// <summary>

/// When the welcome window is closed the custom event is raised to let the

/// semi-transparent overlay know that it shoulud begin it's next event.

/// </summary>

private void welcomeClosed()

{

RaiseCustomEvent(this, true);

if (this.IsVisible)

{

this.Close();

}

}

}

}

# Interfaces and Implementation

## ImpClient.cs

using Interfaces;

using System;

using System.Collections.Generic;

using System.Linq;

using System.ServiceModel;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

namespace ChatWindow

{

/// <summary>

/// Class containing the interfaces to the client.

/// </summary>

[CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]

public class ImpClient : IntClient

{

/// <summary>

/// The message goes from the server to the client end. This will send a message to the windows.

/// </summary>

/// <param name="a\_message">string : message being forwarded</param>

/// <param name="a\_user">string : the user sending the message</param>

public void frwdMsg(string a\_message, string a\_user)

{

// Display message to the text box, format the message

((SemiTransOverlay)Application.Current.MainWindow).messagePass(a\_user + " : " + a\_message);

}

/// <summary>

/// When receiving a message from another user, this function will send the message

/// to the appropriate window.

/// </summary>

/// <param name="a\_msg">string : message to be displayed</param>

/// <param name="a\_source">string : the client's username of who the message is from</param>

public void whisperMsg(string a\_msg, string a\_source)

{

((SemiTransOverlay)Application.Current.MainWindow).incomingMessage(a\_source, a\_msg);

return;

}

/// <summary>

/// Adding a new user to the client-side list. Information supplied from the server.

/// </summary>

/// <param name="a\_connected">string : the new client's username who has connected</param>

public void addUserToList(string a\_connected)

{

((SemiTransOverlay)Application.Current.MainWindow).addNewConnectionList(a\_connected);

}

/// <summary>

/// Removing a client who has disconnected from the list of usernames

/// </summary>

/// <param name="a\_disconnected">string : the client's username who has disconnected.</param>

public void removeUserFromList(string a\_disconnected)

{

((SemiTransOverlay)Application.Current.MainWindow).removeFromConnected(a\_disconnected);

}

}

}

## IntClient.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.ServiceModel;

using System.Text;

using System.Threading.Tasks;

namespace Interfaces

{

[ServiceContract]

public interface IntClient

{

[OperationContract]

void frwdMsg(string a\_message, string a\_user);

[OperationContract]

void whisperMsg(string a\_msg, string a\_user);

[OperationContract]

void addUserToList(string a\_connected);

[OperationContract]

void removeUserFromList(string a\_disconnected);

}

}

## IntServices.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Runtime.Serialization;

using System.ServiceModel;

using System.Text;

namespace Interfaces

{

[ServiceContract(CallbackContract = typeof(IntClient))]

/\*

\* Interface for ImpServices.cs, allowing the client-side to use its functionality.

\* [OperationContract] <function\_name> are all the functions able to be called in the client.

\*/

public interface IntServices

{

[OperationContract]

int login(string a\_username, ref List<string> a\_userList);

[OperationContract]

void sendToAll(string a\_user, string a\_msg);

[OperationContract]

void sendToUser(string a\_dest, string a\_from, string a\_msg);

[OperationContract]

void newClientConnection(string a\_connected);

[OperationContract]

void disconnectionHandle(string a\_currentUser);

}

}

## ConnectedClient.cs

using Interfaces;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Server

{

/// <summary>

///

/// </summary>

public class ConnectedClient

{

/// <summary>

/// connection from the client to the services. Will be accessed to interact with other clients.

/// </summary>

public IntClient connection;

/// <summary>

/// get and set the username for the client being handled.

/// </summary>

public string username { get; set; }

}

}

## ImpServices.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Runtime.Serialization;

using System.ServiceModel;

using System.Text;

using System.Collections.Concurrent;

using Interfaces;

namespace Server

{

/// <summary>

/// Implementation for the operation contracts located in the Inter

/// </summary>

[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple, InstanceContextMode = InstanceContextMode.Single)]

public class ImpServices : IntServices

{

/// <summary>

/// ConcurrentDictionary [string, ConnectedClient] : where the dictionary is populated with [string]usernames and their [connectedClient]connection to the server.

/// </summary>

public ConcurrentDictionary<string, ConnectedClient> m\_connClients = new ConcurrentDictionary<string, ConnectedClient>();

// Conccurency will protect conflicts/race conditions since the server is multi threaded.

/// <summary>

/// If the client picks a unique username, a connection will be made to the server.

/// </summary>

/// <param name="a\_username"> string : the username requested by the client</param>

/// <param name="a\_userList"> List-strings : usernames to be a permanent copy for the

/// client after a connection is made, passed by reference.</param>

/// <returns>

/// return 0 upon duplicate username existing, does not fully connect.

/// return -1 upon error connecting to the server.

/// return 1 upon success.

/// </returns>

public int login(string a\_username, ref List<string> a\_userList)

{

// Make sure the user name does not already exist

foreach (var client in m\_connClients)

{

if(client.Key.ToLower() == a\_username.ToLower())

{

Console.WriteLine("Duplicate user information attempted.");

return 0;

}

}

// Create a new client chat user, with their user name & their connection

var establishedUserConnection = OperationContext.Current.GetCallbackChannel<IntClient>();

ConnectedClient newClient = new ConnectedClient();

newClient.connection = establishedUserConnection;

// Checking username again for uniqueness (race conditions).

newClient.username = a\_username;

Console.WriteLine("[ " +DateTime.Now.ToString("h:mm:ss tt") + " ] " + a\_username + " has connected to the server.");

if (!m\_connClients.TryAdd(a\_username, newClient)) {

Console.WriteLine("Failed to connected to server.");

return -1;

}

// Initial list of users for each new client that connects.

foreach (var client in m\_connClients)

{

a\_userList.Add(client.Key);

}

return 1;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START: Message Sending \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// Sends a message to all currently connected clients, excluding the sender.

/// </summary>

/// <param name="a\_username">string : user sending the message to the client list</param>

/// <param name="a\_msg">string : message to send to the client list.</param>

public void sendToAll(string a\_username, string a\_msg)

{

// Output the string with time, username, message to the server's console.

Console.WriteLine("[ " + DateTime.Now.ToString("h:mm:ss tt") + " ] " + a\_username + " SENT < " + a\_msg + " > TO ALL");

// Pass the message to each client excluding the sender.

foreach (var client in m\_connClients)

{

if (client.Key.ToLower() != a\_username.ToLower())

{

client.Value.connection.frwdMsg(a\_msg, a\_username);

}

}

}

/// <summary>

/// Sends a message from one client to another. Destination client needs to be connected to the server.

/// If the destination client is offline or does not exist, the user is unable to send messages.

/// </summary>

/// <param name="a\_source">string : the client requesting to send a message.</param>

/// <param name="a\_dest">string : the final destination of the message.</param>

/// <param name="a\_msg">string : the message to send to the <paramref name="a\_dest"/> client.</param>

public void sendToUser(string a\_source, string a\_dest, string a\_msg)

{

ConnectedClient getValue;

Console.WriteLine("[ " + DateTime.Now.ToString("h:mm:ss tt") + " ] " + a\_source + " SENT < " + a\_msg + " > TO " + a\_dest);

// If the destination client exists, will be stored in getValue

bool isValue = m\_connClients.TryGetValue(a\_dest, out getValue);

if (isValue)

{

// Using the callback function to whisper the message to the correct destination client

getValue.connection.whisperMsg(a\_msg, a\_source);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END: Message Sending \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START: Dis/Connection \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// When a different client connects to the server, the clients currently connected

/// will receive an update to their client lists so they are aware of the change.

/// </summary>

/// <param name="a\_connected">string : user that had connected.</param>

public void newClientConnection(string a\_connected)

{

if ((m\_connClients).Count <= 1)

{

return;

}

// If something is wrong with the client's username connecting.

if (a\_connected == null || a\_connected == "")

{

return;

}

foreach (var client in m\_connClients)

{

// Pass the client to everyone else

if (client.Key.ToLower() != a\_connected.ToLower())

{

client.Value.connection.addUserToList(a\_connected);

}

}

}

/// <summary>

/// Ensures the user disconnects properly while updating the disconnection

/// to the other connected clients.

/// </summary>

/// <param name="a\_disconnected">string : user that is requesting a disconnect</param>

public void disconnectionHandle(string a\_disconnected)

{

// if the user currently exists, attempt to remove

ConnectedClient userInfo;

if (m\_connClients.TryGetValue(a\_disconnected, out userInfo))

{

// if the client can be removed from the server list

ConnectedClient tmp;

if(m\_connClients.TryRemove(a\_disconnected, out tmp))

{

Console.WriteLine(a\_disconnected + " has left the server.");

}

else

{

Console.WriteLine("Unknown Failure removing: " + a\_disconnected + " from the server.");

return;

}

}else

{

Console.WriteLine("Error Disconnecting: " + a\_disconnected + " from the server.");

return;

}

// Callback action : update other clients

foreach (var client in m\_connClients)

{

// client is already removed, do not need to worry about checking for him i ntthe list

client.Value.connection.removeUserFromList(a\_disconnected);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END: Dis/Connection \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

}

## ServerMain.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.ServiceModel;

using System.Text;

using System.Threading.Tasks;

namespace Server

{

/// <summary>

/// Class containing the server start, and keeps the connection open.

/// This class holds:

/// implementation of all services available to the client.

/// </summary>

class ServerMain

{

public static ImpServices testServ;

/// <summary>

/// Opens the server connection and the instance of the implementation of services.

/// </summary>

/// <param name="args">(none) string of arguments that coukd be passed in at program start. None in this case.</param>

static void Main(string[] args)

{

testServ = new ImpServices();

using(ServiceHost newHost = new ServiceHost(testServ))

{

newHost.Open();

Console.WriteLine("Server is Online.");

Console.ReadLine();

newHost.Close();

}

}

}

}