**[WCF-Duplex zwischen Windows Sevice und GUI-Frontend](http://www.flexbit.at/blog/wcf-duplex-zwischen-windows-sevice-und-gui-frontend/" \o "WCF-Duplex zwischen Windows Sevice und GUI-Frontend)**

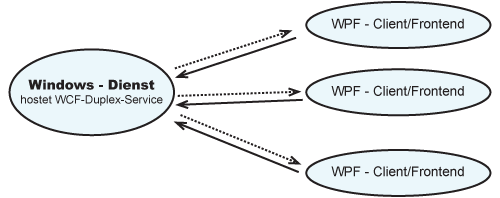
von [Daniel Lang](http://www.flexbit.at/blog/author/admin/) am Juli 10, 2011

Mithile von WCF lässt sich einfach und schnell eine grafische Benutzeroberfläche für einen Windows-Service entwickeln. Über einen Duplex-Contract werden einerseits Statusmeldungen und Informationen vom Dienst zur Benutzeroberfläche übertragen, andererseits kann der Anwender auch Steuerbefehle an den Dienst schicken. Durch den Einsatz von WCF und einem Subcriber-Modell, können beliebig viele Frontends gleichzeitig auf unterschiedlichen Computern verwendet werden.

**Entstehung**

In meinem letzten Artikel habe ich beschrieben, welche unterschiedlichen Möglichkeiten es zur Inter-Process-Communication zwischen einer [GUI und einem Windows-Service](http://www.flexbit.at/blog/gui-fur-windows-service-in-net/) gibt. Für unsere [Magento-Schnittstelle](http://www.mageconnector.com), die ebenfalls als Windows-Dienst im Hintergrund arbeitet, wollten wir eine einfache Möglichkeit schaffen um auch über das Netzwerk oder sogar eine Internetverbindung den Dienst kontrollieren und steuern zu können. Dazu haben wir uns letztendlich für den Einatz von WCF (Windows-Communication-Foundation) entschieden. Etwas verspätet aber doch, halte ich damit mein Versprechen und präsentiere unsere Lösung und auch etwas Beispielcode.

**Systemaufbau**

Wir haben einen Windows-Dienst, der im Hintergrund irgendeine lange Aufgabe ausführt und dabei Statusmeldungen via WCF an einen oder mehrere Frontends schickt. Der Dienst läuf mit .NET 4 und hostet einen WCF-Service, der einen Endpunkt über ein NamedPipe-Binding zur Verfügung stellt. Der Frontend ist eine relativ überschaubare WPF-Anwendung, der den WCF-Service konsumiert (=Client) und auch einen Callback-Kontrakt implementiert, damit der Dienst (=Server) jederzeit Nachrichten an den oder die Clients schicken kann. Hier eine schematische Darstellung:  


**Der Windows-Dienst**

Zunächst einmal erstellen wir einen Windows-Dienst, der im Sekunden-Takt irgendeine Aufgabe ausführt und dabei eine Nachricht in die Event-Log schreibt. Wie man grundsätzlich einen Windows-Dienst in C#/.NET erstellt kann man zur Auffrischung [hier](http://www.codeproject.com/KB/system/WindowsService.aspx) nachlesen. Interessanter wird es in der OnStart-Methode. Hier starten wir einen neuen Thread, der für die regelmäßige Ausführung der Aufgabe zuständig ist (deshalb auch *schedulerThread*):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | protected override void OnStart(string[] args)  {      ...        // start scheduler-thread      SchedulerThread = new Thread(this.ScheduleHandler);      SchedulerThread.Start();  } |

Hier die Methode des neuen Threads, die dann pro Sekunde einen Tick in die EventLog schreibt (vorerst):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | private void ScheduleHandler()  {      while (true)      {          var beginTime = DateTime.UtcNow;            try          {              EventLog.WriteEntry(EventLogSource, "Tick", EventLogEntryType.Information);          }          catch (Exception ex)          {              EventLog.WriteEntry(EventLogSource, "Error: " + ex.Message, EventLogEntryType.Error);          }            // wait until next processing-cycle          var duration = DateTime.UtcNow - beginTime;          int milisecondsToWait = (1000) - (int)duration.TotalMilliseconds;          if (milisecondsToWait > 0)              Thread.Sleep(milisecondsToWait);      }  } |

Zu guter letzt noch die killer-Methode, die den Thread wieder beendet, wenn der Windows-Dienst gestoppt wird:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | protected override void OnStop()  {      // stop scheduler-thread      killThread(SchedulerThread);  }    private void killThread(Thread threadToKill)  {      if (threadToKill != null && threadToKill.IsAlive)      {          // interrupt the thread so it has the opportunity to clean up          threadToKill.Interrupt();            // wait one seconds for the thread to finish          threadToKill.Join(1000);            // if the thread is still alive, kill it          if (threadToKill.IsAlive)              threadToKill.Abort();      }  } |

**Hosten des WCF Service**

Bis jetzt haben wir nur einen “normalen” Windows-Dienst, der pro Sekunde einen Tick in die Ereignisanzeige schreibt. Damit sich aber später unsere Frontend-Anwendung mit dem Dienst verbinden kann, hosten wir einen WCF Service innerhalb des Windows-Dienstes. In diesem Beispiel verwenden wir einen NamedPipe-Endpunkt, der sich für eine besonders schnelle Kommunikation auf demselben Computer eignet. Natürlich könnten wir hier aber auch ein TcpBinding oder ein WsDualHttpBinding verwenden, wenn die Kommunikation über das Netzwerk stattfinden soll. Wichtig ist hier nur, dass es sich um ein Duplex-kompatibles Binding handelt, damit wir über die WCF-Callbacks auch Nachrichten vom Server an den Client *pushen* können.

Das Hosten eines WCF-Services innerhalb eines Windows-Dienstes ist mihtilfe der ServiceHost-Klasse recht einfach. In unserem Beispiel verzichten wir zur Gänze auf eine Konfiguration über die app.config, sondern erstellen den ServiceHost programmatisch. Hier die finale OnStart und OnStop-Methode.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | protected override void OnStart(string[] args)  {      /////////////////////////////      // wcf-service host        controlServiceHost = new ServiceHost(typeof(ControlService), new Uri[] { new Uri("net.<pipe://localhost/servicecontrol>") });      controlServiceHost.AddServiceEndpoint(typeof(IControlService), new NetNamedPipeBinding(), "");        // add metadata-binding, so that a proxy can be generated for the client (not recommended for production-use!)      var serviceMetadataBehaviour = new System.ServiceModel.Description.ServiceMetadataBehavior();      controlServiceHost.Description.Behaviors.Add(serviceMetadataBehaviour);      controlServiceHost.AddServiceEndpoint(ServiceMetadataBehavior.MexContractName, MetadataExchangeBindings.CreateMexNamedPipeBinding(), "mex");        // start wcf-service      controlServiceHost.Open();        // start scheduler-thread      SchedulerThread = new Thread(this.ScheduleHandler);      SchedulerThread.Start();  }    protected override void OnStop()  {      // stop wcf-service      controlServiceHost.Close();        // stop scheduler-thread      killThread(SchedulerThread);  } |

Beim Starten des Windows-Dienstes wird somit ein WCF-Endpunkt mit der Adresse “net.pipe://localhost/servicecontrol” erstellt. Zusätzlich erstellen wir auch noch einen Metadata-Endpunkt, damit uns Visual Studio im Client-Projekt schnell und einfach einen Service-Proxy erstellen kann.

Bevor ich jetzt zeige, wie die Klasse *ServiceControl*, also die Implementierung unseres WCF-Services aussieht, ersetzen wir in der *ScheduleHandler* – Methode (siehe weiter oben) das Schreiben in die Eventlog, durch einen Aufruf unseres WCF-Services (Singleton!), der eine Nachricht (z.B. eine Statusmeldung) an alle Clients veröffentlicht:

|  |  |
| --- | --- |
| 1  2 | //EventLog.WriteEntry(EventLogSource, "Tick", EventLogEntryType.Information);  ControlService.CurrentInstance.PublishMessage("Tick"); |

**WCF Duplex Service**

Jetzt kommt der spannendere Teil… für die 2-Wege-Kommunikation brauchen wir einen sogenannten Duplex-Service, bei dem nicht nur der Client, sondern auch der Server eine Nachricht an den jeweils anderen schicken kann. Damit wird eigentlich ein Nachrichten-Pushing gemacht, d.h. der Server schickt immer dann – und nur dann – wenn er eine neue Nachricht hat, diese zu den Clients. Die schlechtere Alternative wäre hier einen normalen WCF-Dienst zu erstellen, wo der Client in einem kurzen Intervall *pollt* und den Server dabei nach neuen Nachrichten abfragt. Schlechter ist diese Variante deshalb, weil durch das ständige Aufrufen des Dienstes von dem Client unnötiger Overhead (Traffic) entsteht. Außerdem kommen die Nachrichten bei längeren polling-Intervallen (zugunsten eines verringerten Overheads) nur zeitverzögert am Client an.

Für die Umsetzung der Duplex-Kommunikation, bietet uns WCF das Konzept von Callback-Methoden, also Methoden, die vom Server aus auf dem Client aufgerufen werden. Damit der Server aber die entsprechenden Methoden am Client aufrufen kann um Nachrichten an diesen pushen, muss zunächst der Client eine Verbindung zum Server aufbauen und sich “subscriben”. Der Server speichert dann den CallbackChannel (also die Verbindung zum Client) in einer Liste ab, und kann dann bei Bedarf an alle Clients die Nachrichten schicken.

Aber erstmal langsam, hier die Interfaces für den Service-Contract:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | [ServiceContract(CallbackContract = typeof(IControlServiceCallback))]  public interface IControlService  {      [OperationContract]      void Subscribe();        [OperationContract]      void Unsubscribe();  }    public interface IControlServiceCallback  {      [OperationContract(IsOneWay = true, AsyncPattern = true)]      IAsyncResult BeginOnMessageReceived(string message, AsyncCallback acb, object state);      void EndOnMessageReceived(IAsyncResult iar);  } |

Das *IControlService* unterscheidet sich hier von einem “normalen” WCF-Service nur dadurch, dass wir auch gleich einen *CallbackContract* definieren. Im *IControlServiceCallback* definieren wir nur eine einzige Methode, die der Client implementierung muss. Diese ist hier asynchron ausgeführt und teilt sich somit in eine “Begin…” und eine “End…” – Methode. Der Grund warum hier eine asynchrone anstelle einer “normalen” Methode verwendet wird ist, dass dadurch der Server selbst die Callback-Methoden der Clients ansynchron ausführt. Das ist wichtig, denn wenn mehrere Clients verbunden sind, und der Server eine Nachricht an alle innerhalb einer Schleife “pusht”, dann würde ein einzelner Client mit einer fehlerhaften Verbindung den Aufruf an die nachfolgenden Clients verzögern (bis zum timeout, und das kann lang eingestellt sein!).

Als nächstes müssen wir das *IControlService* Interface am Server implementieren (*IControlServiceCallback* dann erst am Client). Hier einmal der ganze Code – Erklärungen dazu kommen darunter:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100 | [ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]  public class ControlService : IControlService  {      // returns the singleton-instance      public static ControlService CurrentInstance { get; private set; }        // holds the channels to all currently connected callbacks (=clients)      private volatile IDictionary<string, IControlServiceCallback> connectedClients;        // lock-object to synchronize between multiple concurrent threads      private object syncRoot;        public ControlService()      {          if (CurrentInstance != null)              throw new Exception("This service-class is intended to be instatiated only once!");          CurrentInstance = this;            syncRoot = new object();          connectedClients = new Dictionary<string, IControlServiceCallback>();      }        public void Subscribe()      {          // get the callback-channel to the client that called this method          var callbackChannel = OperationContext.Current.GetCallbackChannel<IControlServiceCallback>();          string sessionId = OperationContext.Current.Channel.SessionId;            // store it for later use (to push messages)          lock (syncRoot)          {              // make sure, that we don't add the same session twice (e.g. the client tries to subscribe multiple times)              if (!connectedClients.ContainsKey(sessionId))              {                  connectedClients.Add(sessionId, callbackChannel);                    // register for error-events of the current channel                  OperationContext.Current.Channel.Closing += new EventHandler(Channel\_Closing);                  OperationContext.Current.Channel.Faulted += new EventHandler(Channel\_Faulted);              }          }      }        public void Unsubscribe()      {          DisconnectClient(OperationContext.Current.Channel.SessionId);      }        void Channel\_Faulted(object sender, EventArgs e)      {          var channel = sender as IContextChannel;          if (channel != null)              DisconnectClient(channel.SessionId);      }        void Channel\_Closing(object sender, EventArgs e)      {          var channel = sender as IContextChannel;          if (channel != null)              DisconnectClient(channel.SessionId);      }        private void DisconnectClient(string sessionId)      {          // remove the session          lock (syncRoot)          {              if (connectedClients.ContainsKey(sessionId))              {                  connectedClients.Remove(sessionId);              }          }      }        public void PublishMessage(string message)      {          lock (syncRoot)          {              // iterate through all connected clients and push message to each of them              foreach (var callbackChannel in connectedClients.Values)              {                  // call the callback-method asynchronously, so that a leaking connection to one of the clients does not affect this loop                  var asyncResult = callbackChannel.BeginOnMessageReceived(message, new AsyncCallback(OnPushMessageComplete), callbackChannel);                  if (asyncResult.CompletedSynchronously)                      CompletePushMessage(asyncResult);              }          }      }        void OnPushMessageComplete(IAsyncResult asyncResult)      {          CompletePushMessage(asyncResult);      }        void CompletePushMessage(IAsyncResult asyncResult)      {          var callbackChannel = (IControlServiceCallback)asyncResult.AsyncState;          try          {              callbackChannel.EndOnMessageReceived(asyncResult);          }          catch { }      }    } |

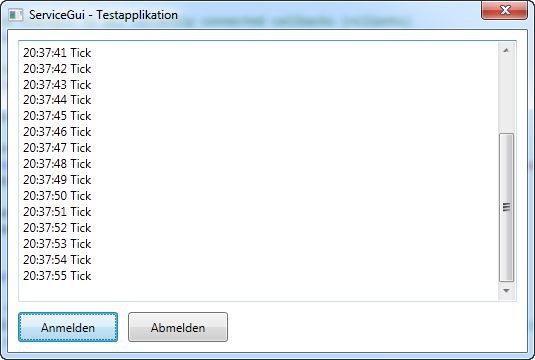
Ganz wichtiges Detail gleich zu Beginn ist folgendes:

|  |  |
| --- | --- |
| 1 | [ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)] |

Durch die Verwendung dieses InstanceContextMode erzwingen wir, dass das Service als Singleton ausgeführt, also nur ein einziges mal instantiiert wird. Das ist wichtig, denn in einem privaten Feld speichern wir die Liste der verbundenen Clients mit deren SessionId ab, die wir *PublishMessage* – Methode brauchen um die Callbacks der verbundenen Clients aufzurufen. Der Rest des Codes sollte selbsterklärend sein, bzw. aus den Kommentaren deutlich werden. Einzig dem *lock(syncRoot)* kommt noch eine besondere Bedeutung zu – Wir verwenden dieses um sicherzustellen, dass mehrere Threads nicht gleichzeitig das Dictionary der Clients verwenden. Zwar sind alle Collections des .NET – Frameworks an sich schon thread-safe, jedoch könnte es ohne dem expliziten locken zu Probleme kommen, wenn ein Thread gereade die foreach-Schleife beim Pushen durchläuft und gleichzeitig dem Dictionary hinzugefügt oder entfernt wird indem sich Clients an- oder abmelden.

**WPF Clientanwendung / Benutzeroberfläche für den Dienst**

Den Frontend habe ich in diesem Beispiel recht einfach gehalten. Er beinhaltet aber dennoch die wichtigsten Funktionen und sollte für seinen Zweck ausreichend sein:



In der Textbox werden Zeile für Zeile die eingehenden Nachrichten vom Server angezeigt. Die Nachrichten werden aber erst empfangen, wenn der Client am Server (am Windows-Dienst) “angemeldet” ist.

Für die Umsetzung habe ich in diesem Beispiel WPF verwendet. In einem neuen WPF-Projekt müssen wir dazu zunächst eine ServiceReference hinzufügen und können dabei die Adresse “net.pipe://localhost” verwenden. Visual Studio erkennt dann automatisch den Metadata-Endpunkt des laufenden Dienstes (vorher bitte installieren und starten!) und generiert einen entsprechenden Proxy. Hier der Code-Behind des MainWindows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | public partial class MainWindow : Window  {      private ControlServiceClient controlService;        public MainWindow()      {          InitializeComponent();            // register event-handler          this.btnSubschribe.Click += new RoutedEventHandler(btnSubschribe\_Click);          this.btnUnsubscribe.Click += new RoutedEventHandler(btnUnsubscribe\_Click);            // intialize callback          var callback = new ControlServiceHandler();          callback.OnMessageReceivedEvent += new ControlServiceHandler.MessageReceivedHandler(callback\_OnMessageReceivedEvent);          var callbackInstanceContext = new InstanceContext(callback);            // create wcf-proxy          controlService = new ControlServiceClient(callbackInstanceContext);      }        void btnSubschribe\_Click(object sender, RoutedEventArgs e)      {          // subscribe to receive messages from the wcf-service          controlService.Subscribe();      }        void btnUnsubscribe\_Click(object sender, RoutedEventArgs e)      {          // unsubscribe from wcf-service          controlService.Unsubscribe();      }        void callback\_OnMessageReceivedEvent(string message)      {          Dispatcher.BeginInvoke((Action) (() =>              {                  tbLog.AppendText(DateTime.Now.ToLongTimeString() + " " + message + Environment.NewLine);                  tbLog.ScrollToEnd();              }));      }  }    [CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant, UseSynchronizationContext = false)]  public class ControlServiceHandler : IControlServiceCallback  {      public delegate void MessageReceivedHandler(string message);      public event MessageReceivedHandler OnMessageReceivedEvent;        public void OnMessageReceived(string message)      {          if (this.OnMessageReceivedEvent != null)              this.OnMessageReceivedEvent(message);      }  } |

Interessant ist hier nur die Klasse *ControlServiceHandler*, die den CallbackContract des WCF-Service implementiert und somit die vom Server kommenden Nachrichten empfängt, sobald via *Subscribe* am Server angemeldet wurde. Hier sieht man auch schon, dass bei der Generierung des *IControlServiceCallback* am Client nur mehr eine Methode *OnMessageReceived* da ist, während im selben Interface am Server durch das *AsyncPattern* noch die beiden Methoden *BeginOnMessageReceived* und *EndMessageReceived* definiert wurden.

Ganz wichtig ist hier, dass *ConcurrencyMode.Reentrant* verwendet wird, und der WPF-eigene *SynchronizationContext* deaktiviert wurde! Letzterer ist der Grund für unzählige Posting auf [StackOverflow](http://www.stackoverflow.com), bei denen über “hängende” Benutzeroberflächen geklagt wird. Grund dafür ist, dass die Callback-Methode am Client in einem eigenen Thread ausgeführt wird und man die UI bekanntlich nur im UI-Thread ändern darf. Um dieses Problem zu vereinfachen hätte WPF diesen *SynchronizationContext* eingeführt um dem Programmierer den manuellen BeginInvoke-Aufruf zu ersparen, nur leider funktioniert der in vielen Situation nicht ordentlich. Deshalb verlassen wir uns hier besser auf die bekannten Methoden und verwenden im Event-Handler der MainWindow-Klasse *Dispatcher.BeginInvoke* um den WCF-Thread mit dem UI-Thread zu synchronisieren.

So, das war’s auch schon. Wir haben jetzt einen Windows-Dienst geschaffen und einen WPF-Client entwickelt, der die Nachrichten des Windows-Service empfängt und anzeigt. Das Beispielprogramm kann einfach erweitert werden, sodass der Client auch Befehle an den Dienst schicken kann oder z.B. auch über ein NetTcpBinding die Ausführung von mehreren Clients auf unterschiedlichen Computern im Netzwerk zulässt.

Den vollständigen Code zum Projekt gibt es [hier zum Herunterladen](http://www.flexbit.at/blog/wp-content/uploads/2011/07/ServiceGui.zip).

Ich hoffe mit diesem Beitrag dem einen oder anderen helfen zu können, und freue mich über Verbesserungsvorschläge und Tipps wie man meinen Code optimieren könnte!

Tags: [IPC](http://www.flexbit.at/blog/tag/ipc/), [Service](http://www.flexbit.at/blog/tag/service/)

**Ähnliche Artikel:**

* [GUI für Windows Service in .NET](http://www.flexbit.at/blog/gui-fur-windows-service-in-net/)

[← GUI für Windows Service in .NET](http://www.flexbit.at/blog/gui-fur-windows-service-in-net/)

**7 Kommentare to “WCF-Duplex zwischen Windows Sevice und GUI-Frontend”**

Sven Scheidler Juli 10, 2011 at 16:06 [#](http://www.flexbit.at/blog/wcf-duplex-zwischen-windows-sevice-und-gui-frontend/#comment-29)

Hallo Daniel,  
sehr interressanter Artikel, da ich einen ähnlichen WCF-Aufbau habe.  
Nur habe ich bis dato noch nicht ganz kapiert, wie ich von einem Client einen anderen bestimmten Client über den WCF-Service (Server) anspreche. Alle Clients “mit einem Rutsch” anzusprechen, ist ja mit obiger Lösung möglich, aber direkt einen bestimmten?!  
Die Clients müßten sich folglich immer eindeutig anmelden, z.B. mit einem Namen o.ä..  
Die SessionID kennt ja nur der HostService.  
Vielleicht kannst Du mir da auf die Sprünge helfen?  
Sven

[Daniel Lang](http://www.flexbit.at/softwareentwicklung) Juli 11, 2011 at 20:52 [#](http://www.flexbit.at/blog/wcf-duplex-zwischen-windows-sevice-und-gui-frontend/#comment-32)

Hallo Sven,

Ich bin mir nicht ganz sicher, ob ich deine Frage richtig verstanden habe, aber es klingt als möchtest du eine Art “ChatServer” bauen, wo sich ein Client mit dem anderen Client austauschen kann und diese Kommunikation über den Server geleitet wird mit dem beide verbunden sind.

In diesem Fall würde der Subscribe-Methode noch einen Paramater mitgeben, mit der sich der Client eindeutig identifiziert (z.B. ClientID). Serverseitig verwendest du dann als Value im Dictionary nicht einfach nur den CallbackChannel, sondern packst diesen in eine eigene Klasse rein, wo du auch die ClientID speicherst. Wenn du dann noch irgendeine Servicemethode “SendTo(string message, string targetClientID)” hast, dann suchst du dir aus dem Dictionary den passenden Eintrag für die targetClientID heraus und kannst den ebenfalls gespeicherten CallbackChannel verwenden um diesem Ziel-Client die Nachricht weiterzureichen…

Ich hoffe das hilft dir. Falls nicht, lass es mich bitte wissen – ich kann dir auch etwas Beispielcode präparieren.

Sven Scheidler Juli 13, 2011 at 15:20 [#](http://www.flexbit.at/blog/wcf-duplex-zwischen-windows-sevice-und-gui-frontend/#comment-33)

Hallo Daniel,

Du liegst im wesentlichen richtig mit der Annahme “sowas ähnliches wie ein ChatServer”. Nur das es keiner ist.  
Meine Anwendung besteht aus mehreren Clients, einer Admin-Oberfläche (<- ist ja auch ein Client) und dem Server (WCFService) dazwischen (tcpbinding).  
Nun möchte ich nur in der Admin-Oberfläche erreichen, das ich einen speziellen Client ansprechen kann. Die Clients sind jeweils auf verschiedenen PC und der Admin-Oberfläche mit Computernamen bekannt.  
Nun verbinden sich die Clients mit dem Server, dabei soll als Ident der Computername übergeben werden, dann könnte ich von der Admin-Oberfläche doch eigentlich einen bestimmten Ansprechen.  
Soweit zur Theorie, mit der Praxis haperts bei mir!

Gern würde ich Dein Angebot mit Beispielcode annehmen, da ich nicht so wirklich weiter komme. Dein Vorschlag mit der Klasse hab ich nicht so richtig verstanden.

Vielen Dank im Voraus  
Sven

[Daniel Lang](http://www.flexbit.at/softwareentwicklung) Juli 14, 2011 at 17:34 [#](http://www.flexbit.at/blog/wcf-duplex-zwischen-windows-sevice-und-gui-frontend/#comment-34)

Hallo Sven!

Ich bau’ dir am Montag kommende Woche etwas Code zusammen, mit dem du das machen können wirst…

Schöne Grüße,  
Daniel

Sven Scheidler Juli 15, 2011 at 21:17 [#](http://www.flexbit.at/blog/wcf-duplex-zwischen-windows-sevice-und-gui-frontend/#comment-35)

Hallo Daniel,

schon mal vielen Dank für Deine Hilfe.

Aber eine Sache hab ich da noch:  
MIr fiel auf, wenn ich den Windows-Service (Server) neu starte und der Client noch aktiv ist,  
ist keine Verbindung zwischen den beiden mehr möglich.  
Starte ich den Client danach ebenfalls neu, ist die Verbindung wieder korrekt.  
Wie kann man das Bereinigen?  
Übrigens habe ich den Binding-Param ‘ReceiveTimeout’ auf TimeSpan.MaxValue gestellt, da bei Inaktivität seitens des Clients der Channel geschlossen wird!

Vieleicht kannst Du mir ja DIE Lösung erklären, sehr gern auch mit Code…

Schönes WE,  
Sven

[Daniel Lang](http://www.flexbit.at/softwareentwicklung) Juli 19, 2011 at 11:37 [#](http://www.flexbit.at/blog/wcf-duplex-zwischen-windows-sevice-und-gui-frontend/#comment-38)

Hallo Sven,

Dass die Verbindung nach einem Server-Neustart nicht mehr möglich ist, ohne den Client neu zu starten liegt daran, dass der WCF-Proxy im Client schon im Konstruktur erstellt wird. Ich hab das geändert.

Außerdem hab ich dir wie versprochen den Code etwas erweitert, sodass du die Clients gemeinsam mit deren Computername am Server registriert hast und über eine Servicemethode “SendMessageToClient” gezielt einen einzelnen ansprechen kannst. Die Methode “GetAllConnectedClients” gibt dir eine Liste aller verbundenen Clients zurück.

Den Code gibt es hier zu downloaden: <http://www.flexbit.at/blog/wp-content/uploads/2011/07/serviceguiext.zip>

Bitte lass mich wissen, ob das hilfreich war. Falls ja, ein “like” auf unserer Facebook-Seite <http://www.facebook.com/flexbit> wäre super – falls nicht, sag Bescheid und wir schauen weiter!

Lg, Daniel

Sven Scheidler August 5, 2011 at 15:34 [#](http://www.flexbit.at/blog/wcf-duplex-zwischen-windows-sevice-und-gui-frontend/#comment-65)

Hallo Daniel,

vielen Dank für Deine Hilfe, nun hab ich das auch hinbekommen.  
Alle Clients sind einzeln erreichbar.

Noch ein Hinweis von mir vielleicht. Wenn Duplex-Verfahren angewendet werden, sollte man beachten, das standardmäßig nur bis zu 10 gleichzeitige Session am Host erlaubt/eingestellt sind. Das war dann nochmal ein Stolperstein.  
Aber durch anpappen eines ‘ServiceThrottlingBehavior’ an den Host, sind dann mehrere Sessions erlaubt/einstellbar.  
Sollte man bei Duplex-Verfahren, zumindest mit meiner Konstellation, beachten.