Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Peter Ožbot

Realizacija potisnega obveščanja z uporabo SignalR in WCF

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Boštjana Slivnika

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Knjižnica SignalR in ogrodje Windows Communication Framework (WCF) omogočata različne načine izvedbe komunikacije med mobilnimi aplikacijami, spletnimi aplikacijami in strežniki. Na osnovi enostavne aplikacije po lastni izbiri primerjajte enostavnost uporabe knjižnice SignalR in ogrodja (WCF) za realizacijo potisnega obveščanja.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Peter Ožbot, z vpisno številko **63040119**, sem avtor diplomskega dela z naslovom:

Realizacija potisnega obveščanja z uporabo SignalR in WCF

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Boštjana Slivnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 15. oktober 2014 Podpis avtorja:

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
WCF	windows communication foundation	windows komunikacijsko ogrodje
SOA	service-oriented architecture	storitveno usmerjena arhitektura
IIS	internet information services	internetno informacijske storitve
WP8	windows phone 8	windows telefon 8
WPF	windows presentation foundation	windows predstavitveno ogrodje
API	application pogramming interface	programski vmesnik
RPC	remote procedure call	klic za oddaljeni postopek
XAML	extensible application markup language	razširljivi označevalni jezik
HTML	hypertext markup language	jezik za označevanje nadbesedila

Povzetek

Komunikacijo med strežniki in odjemalci lahko realiziramo na več načinov. Osnoven način je, da odjemalec povprašuje in strežnik odgovarja. V primeru, ko odjemalec ne ve, kdaj mu bo določena informacija na voljo, mora v intervalih spraševati strežnik. To je neučinkovito in boljša alternativa temu je, da strežnik obvešča odjemalce. Takemu načinu komunikacije pravimo potisno obveščanje.

Implementacijo potisnega obveščanja smo izvedli s knjižnico SignalR in z ogrodjem WCF. Obe tehnologiji smo raziskali in ju primerjali.

Za primerjavo tehnologij smo implementirali obveščanje o dogajanju v nadzoru različic kode. V našem primeru smo imeli dva tipa odjemalcev. En odjemalec je vmesnik za upravljanje z nadzorom različic kode Git. Ta obvešča strežnik o akcijah, ki se dogajajo nad kodo. Drugi tip odjemalca pa je namizna aplikacija in aplikacija za Windows Phone 8. Obe aplikaciji sprejemata obvestila o dogajanju v nadzoru različic kode, ki jim jih pošilja strežnik.

Ključne besede: SignalR, WCF, Windows Phone 8, Git, potisno obveščanje.

Abstract

Communication between servers and clients can be realised in several ways. A basic method is, when the client requests and the server responds. In case the client does not know, when the specific information will be available, it needs to ask the server in intervals. This is ineffective, and a better alternative is when the server informs the clients. Such communication is called server push.

Server push was implemented with the SignalR library and with the WCF framework. Both technologies have been explored and compared.

For comparison of technologies we implemented the notification of activity in revision control. In our case, we had two types of clients. The first client is an interface for managing the control of the Git revision control. It informs the server about actions, which take place over the code. The second client is a desktop application and an application for Windows Phone 8. Both applications receive notifications of activities in revision control.

Keywords: SignalR, WCF, Windows Phone 8, Git, server push.

Poglavje 1

Uvod

Pri praktično popolni pokritosti z internetno povezavo in vedno večji razširjenosti elektronskih naprav prihaja do vedno boljšega izkoriščanja možnosti komunikacije med njimi. Standarden primer komunikacije je, da se na napravi izvaja neka aplikacija, ki se obnaša kot odjemalec. Ta aplikacija povprašuje strežnik po podatkih in čaka, da jih strežnik vrne.

V primerih, ko je okolje tako, da odjemalec sam zahteva vsebino, je tak način zadovoljiv. S takim načinom pride do težav, ko strežnik želi obvestiti odjemalce, ne da bi sami to prej zahtevali. Kot primer takega obveščanja bi lahko vzeli prikazovanje tečaja delnic. Ko se spremeni vrednost tečaja, je treba obvestiti odjemalce, da spremenijo prikazano ceno in tako obvestijo uporabnika o spremembi. Za take primere je treba drugače implementirati komuniciranje med odjemalci in strežniki.

Obstajajo seveda razni triki, ki se uporabljajo za simuliranje obveščanja s strani strežnika. Na primer odjemalec v intervalih povprašuje strežnik po novostih in strežnik odgovarja glede na trenutno stanje. Komunikacija v tem primeru deluje na isti način, ampak zaradi implementacije povpraševanja v intervalih ima uporabnik občutek, da ga strežnik obvešča. Rešitve, narejene na tak način so delujoče, ampak po nepotrebnem uporabljajo vire, ki so na voljo.

Z implementacijo obveščanja, ki v intervalih povprašuje strežnik, je po navadi še ena težava. To je zakasnitev med pojavitvijo nove informacije in tem, kdaj je odjemalec obveščen. Ta zakasnitev je minimalna velikosti intervala. Problem nastane v primeru, ko se interval krajša, tako pride do prevelike uporabe virov. V primeru, ko je interval preveč dolg, je zakasnitev previsoka.

Boljša rešitev je uporaba novejših tehnologij, ki omogočajo povezavo med strežnikom in odjemalcem, ki se ne prekine takoj, ampak ostane odprta čez cel čas. S tako povezavo med strežnikom in odjemalcem lahko strežnik pošilja podatke odjemalcem, ne da bi ti sami to zahtevali. Takemu obveščanju pravimo potisno obveščanje. Na tak način se minimizira uporaba virov in obveščanje poteka v realnem času. Praksa ni tako svetla, ker obstajajo določene omejitve in najbolj optimalno delovanje je samo v popolnih pogojih.

Trenutno obstaja več načinov, s katerimi implementiramo potisno obvešča-nje v Microsoftovem okolju. Osnovni način je uporaba ogrodja Windows Communication Foundation (WCF) [1]. Drugi način je uporaba knjižnice SignalR [2].

V tej diplomski nalogi sem izdelal aplikacijo, ki uporablja potisno obveščanje. Aplikacija se obnaša kot strežnik, ki pošilja obvestila vsem povezanim odjemalcem. Za implementacijo potisnega obveščanja sem uporabil knjižnico SignalR. Uporabljal sem Microsoftove tehnologije in orodja za razvoj. Rešitev sem tudi primerjal z ogrodjem WCF in podal prednosti in slabosti izbire različnih pristopov.

Poglavje 2

Knjižnica SignalR

SignalR je knjižnica, ki je namenjena reševanju problematike komunikacije v realnem času z uporabo potisnega obveščanja. To pomeni, da strežnik obvesti odjemalce o spremembi takoj, ko se zgodi. Razvita je bila z namenom izgradnje sistema za komunikacijo med strežnikom in odjemalci, ki bo preprosta za uporabo, hitrega delovanja in celoten sistem bo deloval enako dobro pri različnem številu odjemalcev. Napisana je v C# programskem jeziku.

Velika prednost knjižnice je, da je z njo zelo preprosto implementirati funkcionalnosti, ki se izvajajo v realnem času. To pomeni, če odjemalec A pošlje sporočilo za vse ostale odjemalce, povezane na strežnik, jim ni treba le-tega v intervalih spraševati, ali je zanje prispelo kakšno sporočilo, ampak so obveščeni takoj, ko prispe. Točno tako delovanje je resnično samo v idealnih okoliščinah, kot bom razložil kasneje. Druga velika prednost pa je to, da je mogoče imeti eno implementacijo za odjemalce na različnih platformah. Na primer en odjemalec je spletna aplikacija, drugi pa namizni program. Izdelali so namreč orodje, ki samodejno izdela kodo za uporabo na drugih platformah. Potebna je samo definicija, napisana v C# jeziku.

2.1 Komunikacija v realnem času

Kot že omenjeno, je sporočanje med odjemalci in strežnikom potisno. To se doseže z uporabo tehnologije WebSocket [3]. To je protokol, ki nudi obojesmerno (full-duplex) komunikacijo preko TCP povezave. Deluje tako, da nudi standardiziran način komunikacije med strežnikom in odjemalcem. Zaradi tega, strežnik lahko

pošlje odjemalcu določene podatke, ne da jih odjemalec prej zahteval. Komunikacija poteka preko vrat 80, kar pomeni, da komunikacija poteka tudi v okoljih, kjer so druga vrata blokirana s požarnimi zidovi. To je resnično potisno obveščanje, vendar protokola ne podpira večina obstoječih strežnikov oziroma odjemalcev [4], čeprav je bil le-ta leta 2011 standardiziran (RFC 6455). Težava je tudi, ker morata oba, strežnik in odjemalec, podpirati ta protokol. Zaradi dejstva, da je WebSocket slabo razširjen, ima SignalR implementirano logiko izbire načina prenosa. To pomeni, da najprej poizkuša vzpostaviti povezavo z najboljšo možno izbiro, to je WebSockets, in če ta ni možna, začne poizkušati z drugimi načini prenosa podatkov.

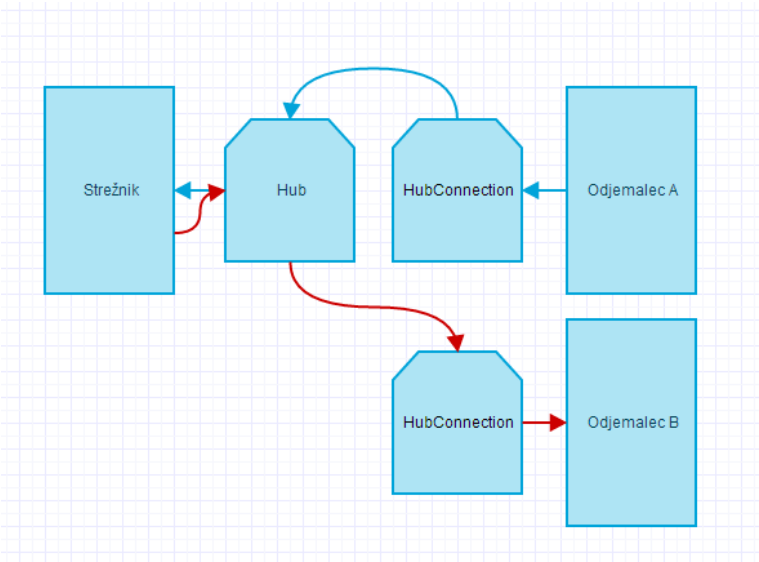
2.2 Načini prenosa podatkov

Načini prenosa podatkov, ki jih knjižnica izbira, so razdeljeni v dve skupini. Prva skupina so transporti standarda HTML5, to je seveda že omenjeni WebSockets. Druga izbira, ki se nanaša na podporo HTML5, je Server Sent Events [5]. Server Sent Events je znan tudi po imenu EventSource. To so dogodki, ki jih pošilja strežnik in prejema odjemalec. Tehnologija je podobna kot WebSockets, ampak taki dogodki so enosmerna komunikacija strežnika proti odjemalcu. Ideja je podobna dogodkom iz različnih programskih jezikov.

Če ni možen noben način prenosa iz skupine standarda HTML5, se izbira načine iz tako imenovane skupine Comet [6]. To so načini, ko odjemalec vztraja v vzdrževanju dolgoročno "odprtega" HTTP zahtevka, preko katerega strežnik pošlje podatke odjemalcu. Prvi način v tej skupini je specifično vezan na Internet Explorer in se imenuje Forever Frame [6].

Forever Frame je način, ki ustvari okvir (Frame) za pošiljanje strežniku zahteve, ki se ne zaključi. Preko te zahteve strežnik obvešča odjemalca. Odjemalec pa sporoča strežniku preko navadnega HTML zahtevka.

Način, ki je izbran najbolj pogosto, to pomeni, ko ni podpore za HTML5, je Ajax long polling [7]. To je nadgradnja navadnega povpraševanja v intervalih. Ajax long polling deluje tako, da pošlje zahtevo strežniku in čaka, da strežnik odgovori ali pa da zahteva poteče. Ko dobi odgovor, takoj sproži novo zahtevo in čaka. Med ponovnim povezovanjem je luknja v komunikaciji, kar pomeni, da



Slika 2.1: Primer delovanja obveščanja

določeni klici iz strežnika pridejo zakasnjeno.

Način izbire načina prenosa je odvisen tudi od nastavitev povezave in okolja, kjer se povezava vzpostavlja.

Ker se v večini primerov, vsaj za zdaj, izbira način prenosa Ajax long polling, obveščanje ni potisno, ampak je videti zelo podobno temu.

2.3 Delovanje

Knjižnica služi kot programski vmesnik (API – application programming interface) za izmenjavo sporočil med odjemalcem in strežnikom preko oddaljenih klicev (RPC – remote procedure call).

Vmesnik nudi dva razreda za komunikacijo – to sta HubConnection ter Hub. HubConnection je razred, preko katerega se odjemalec poveže na strežnik in pošilja ter prejema sporočila strežnika. Hub pa deluje kot nekakšna centralna točka, preko katere se izvajajo vsi klici. Pomembno je vedeti, da se pri vsakem klicu ustvari nov primerek Hub-a.

Slika predstavlja primer delovanja. Najprej odjemalec A izvede klic preko razreda HubConnection, ki pošlje sporočilo na naslov, kjer gostuje Hub. Strežnik

obdela sporočilo in preko Hub-a izvede klic, ki ga prejme HubConnection odjemalca B. Strežnik lahko obvesti veliko odjemalcev hkrati ali pa točno določenega. Povezava med odjemalcem in strežnikom je trajna, to pomeni, da se ne vzpostavlja na vsak izvedeni klic, ampak je odprta cel čas trajanja povezave.

2.4 Uporaba SignalR

2.4.1 Načini Gostovanja

Spletne aplikacije v .Net ogrodju je mogoče gostiti na dva načina. Enako je mogoče gostiti aplikacijo, ki uporablja SignalR. Eden izmed načinov je gostovanje z Internet Information Services (IIS) [8]. Drugi način je tako imenovani Self-Host.

IIS je razširljiv spletni strežnik, ki ga je izdelal Microsoft za namene gostovanja spletnih aplikacij. Spletni strežnik podpira vse glavne protokole HTTP, FTP, SMTP in njihove varne različice. Spletni strežnik je del operacijskega sistema Windows, ampak ni privzeto vključen in ga je treba ročno aktivirati. Novejše različice podpirajo tudi WebSockets.

Gostovanje Self-Host namreč pomeni to, da je mogoče gostovati brez dodatne aplikacije, na primer v Windows servisih ali neposredno v namizni aplikaciji oziroma v katerem koli procesu. Za gostovanje se uporablja prav tako prosto dostopna knjižnica Owin [9]. Ta knjižnica je samo izločen pogon za gostovanje spletnih aplikacij iz IIS.

2.4.2 Gostovanje SignalR

Z IIS je uporaba popolnoma ista kot z vsako spletno aplikacijo. Izdelati je namreč treba spletno aplikacijo in dodati potrebne reference na knjižnico SignalR ter klicati razširitveno metodo, ki preslika in vzpostavi vse potrebno za gostovanje Hub-a.

```
public class Startup {
    public void Configuration(IAppBuilder app) {
        app.MapSignalR();
    }
}
```

Zaradi integracije med Owin in SignalR je gostovanje kot self-host skoraj enako preprosto kot z IIS. Enako je potrebno preslikati, kje se nahaja Hub in zagnati samo gostovanje.

```
public class Startup {
    public void Configuration(IAppBuilder app) {
        app.Map("/signalr", map => {
            map.RunSignalR();
        });
    }
}
```

Pri zagonu je treba povedati tudi, na katerem naslovu naj se gosti.

```
WebApp.Start<Startup>("http://localhost:8080/");
```

2.4.3 Definiranje Hub-a

Pri vzpostavljanju sistema za komunikacijo s SignalR je treba najprej poskrbeti za strežnik, ki bo gostil in razporejal prejeta sporočila. Za predstavitev strežnika se uporablja razred Hub, ki je del knjižnice SignalR. S pomočjo tega razreda definiramo metode ali klice, ki jih bo strežnik sprejemal. Svojo implementacijo definiramo tako, da napišemo definicijo razreda, ki deduje iz razreda Hub.

Primer definicije Hub-a, ki sprejema klice preko metode Send.

```
public class MyHub : Hub {
    public void Send(string message) {
        Clients.Others.Receive(message);
    }
}
```

V tem primeru, ko odjemalec izvede klic Send na MyHub – to je strežnik, ta obvesti vse ostale odjemalce preko metode Receive. Vsak odjemalec namreč registrira metodo, preko katere ga lahko strežnik kliče, v tem primeru naj bi to bila metoda Receive.

Strežnik lahko sprejema katere koli podatke kot parametre metod, ki jih ima. Pri osnovnih tipih ni težav, za bolj kompleksne razrede pa je treba poskrbeti, da jih je mogoče serializirati. Vsi podatki se namreč prenašajo kot besedilo.

Zavedati se je treba tudi, da se na vsak klic ustvari nov primerek razreda MyHub, na drugem naslovu v pomnilniku. To pomeni, da se ohranjajo samo statični podatki oziroma takšni, ki se ohranjajo v celem procesu in ne samo v primerku razreda MyHub.

V prikazanem primeru so obveščeni vsi povezani odjemalci, razen tistega, ki je izvedel klic na strežnik. Poleg tega, ima strežnik možnost obveščanja odjemalce preko različnih kriterijev:

- Samo tistega, ki je izvedel klic.
- Vse povezane odemalce.
- Mogoče je tudi razporejati odjemalce v različne skupine in potem pošilj-ati podatke samo določenim.

Poleg tega, da se Hub uporablja za razporejanje, kdo bo obveščen, ga je mogoče uporabiti za druge načine. Ko se namreč zgodi posamezen klic, ima dostop do različnih informacij o klicatelju (odjemalcu) in tudi, kdaj se vsak posamezen poveže ter ostale podatke o sami povezavi.

Že iz tega, kar definicija Hub-a ponuja, je videti, da je kot nekakšna skupna točka, preko katere se razporejajo klici med več odjemalci. Zaradi preprostosti in možnosti razvijalca, da spreminja in dodaja kar koli želi, se lahko uporablja za marsikaj drugega.

2.4.4 Definiranje Odjemalca

Odjemalec je predstavljen kot povezava z razredom HubConnection. Odjemalec je lahko spletna stran, uporablja se JavaScript. Lahko je tudi aplikacija, ki se izvaja, kjer je nameščeno .Net ogrodje. To je lahko Widows Phone ali pa osebni računalnik z operacijskim sistemom Windows.

Za vzpostavitev povezave kot JavaScript odjemalec, je treba najprej vključ-iti skripte, ki so del knjižnice SignalR.

```
<script src="Scripts/jquery.signalR-2.1.0.min.js"></script>
```

Vključiti je treba tudi skripto, ki je bila samodejno zgenerirana iz definicije Hub-a. Skripto vključimo kot naslov do definicije Hub-a v JavaScript. Treba je vedeti, da se datoteka na naslovu zgenerira ob zagonu strežnika in pred tem ne obstaja.

```
<script src="http://localhost:8080/signalr/hubs"></script>
```

Pri vzpostavitvi povezave je treba najprej nastaviti naslov strežnika. Potem pridobimo referenco do samodejno generiranega Hub-a. Preko tega razreda se bodo izvajali vsi klici.

```
$.connection.hub.url = "http://localhost:8080/signalr";
 var myHub = $.connection.myHub;
```

Za prejemanje klicev in sporočil strežnika je treba registrirati metodo, ki naj se izvede, ko strežnik kliče istoimensko metodo. S to metodo se obdelajo vsi klici strežnika.

```
myHub.client.receive = function(message) { }
```

Ko so te osnovne stvari nastavljene, je zadnja točka zagon povezave. Paziti je potrebno, da se klici proti strežniku začnejo izvajati po tem, ko je povezava vzpostavljena. Knjižnica nam ta problem zelo poenostavi z nudenjem registracije metode, ki naj se izvede, ko je povezava uspela. Na ta način vemo, kdaj lahko začnemo pošiljati podatke.

```
$.connection.hub.start().done(function () {
```

```
        myHud.server.send("neki podatki");
});
```

Drugi način je povezava preko odjemalcev, ki podpirajo ogrodje .Net. Ideja je ista kot v primeru JavaScript odjemalca. Najprej je treba nastaviti naslov, registrirati metodo, ki naj se izvede ob klicih strežnika in zagnati povezavo. Različna je samo sintaksa in to zaradi jezika C#.

```
var connection = new HubConnection("http://localhost:8080/");

var myHub = connection.CreateHubProxy("MyHub");

myHub.On<string>("Receive", param => {
          Console.WriteLine(param);
      });

connection.Start();
```

Enako velja, da je treba najprej vzpostaviti povezavo in ogrodje nudi enak način povezovanja metod. V primeru C# se to dela s pomočjo razreda Task, ki ima metodo ContinueWith.

Poglavje 3

Ogrodje Windows Presentation Foundation

V .Net okolju je alternativa implementaciji obveščanja mogoča z uporabo skupine programskih vmesnikov, združenih v ogrodje pod imenom Windows Presentation Foundation (WCF) [1]. Ogrodje se uporablja za implementacijo komunikacije med dvema končnima točkama. Glavni namen je uporaba v storitveno-orientiranih aplikacijah. Take aplikacije so zgrajene po principu storitveno usmerjene arhitekture (SOA) [13]. SOA se uporablja kot načrtovalski vzorec za implementacijo komunikacije med dvema končnima točkama. Ena izmed točk ponuja določeno storitev in druga jo zahteva.

Zaradi razširljivosti se lahko uporablja tudi za druge oblike komunikacije, kot na primer za med procesno komunikacijo.

Glavni gradnik ogrodja je končna točka (endpoint), to je abstraktna predstavitev vmesnika, ki sprejema oziroma pošilja podatke. Komunikacija poteka po izmenjavi podatkov o tem, kaj se bo prenašalo in kako, to je definirano v definiciji vmesnika. Definiciji, ki je opis točke, pravimo tudi pogodba.

Ena izmed teh dveh točk je lahko storitev, ki nudi množici odjemalcev različne podatke, ali pa eden izmed dveh odjemalcev, ki komunicirata med seboj. Čeprav model razlikuje med odjemalcem in storitvijo, to pomeni, da storitev čaka odjemalčevo zahtevo po določenih podatkih in vsi odjemalci se lahko obnašajo tudi kot ponudniki storitev. Iz tega sledi, da je mogoče izvesti obveščanje s strani

strežnika. To je mogoče na dva načina. En način je, da se v definiciji vmesnika definira, da končna točka sprejema in pošilja podatke. To definiramo tako, da nastavimo, kateri vmesnik se bo uporabil za metodo povratnih klicev. Na tak način dosežemo hkratno komunikacijo, to je komunikacija, ki poteka hkrati v obe smeri. Drugi način pa je uporaba omrežja vsak z vsakim (P2P). To pomeni, da ni nobena točka glavna kot ponudnik storitve, ampak so vse točke med seboj povezane kot enakovredni udeleženci.

Iz arhitekture bo vidno, da WCF ponuja razvijalcu še veliko drugih opcij za implementacijo različnih funkcionalnosti. Vseh ne bom opisoval, ker nima smisla, podrobneje bom opisal samo tisto, kar sem sam uporabil.

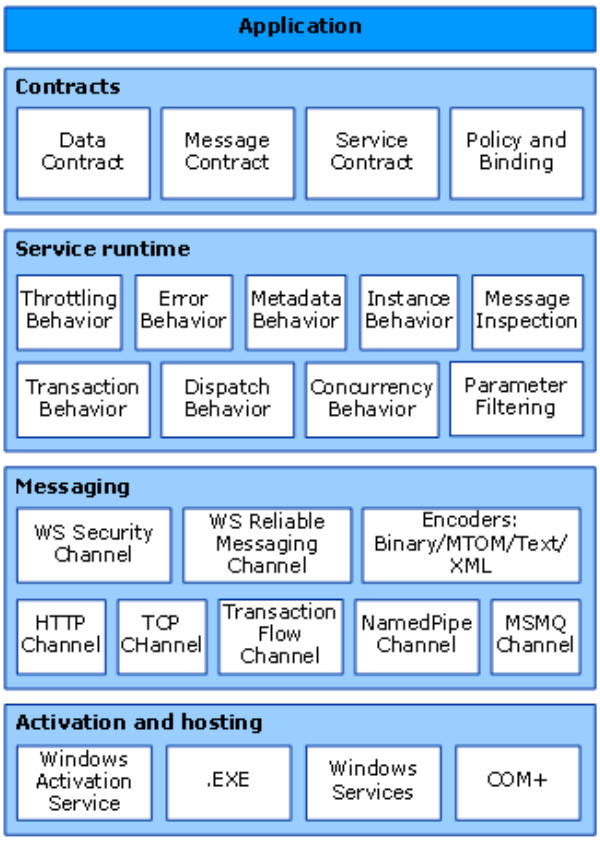
3.1 Arhitektura WCF

Pogodbe (ang. *contracts*) definirajo vse parametre, ki jih storitev lahko prejema ali pošilja. Parametri sporočil so definirani z XML shemo, to omogoča vsem sistemom, ki tako shemo razumejo, da lahko razumejo sporočila. Pogodbe definirajo tudi podpise metod storitve in so predstavljene z vmesniki, ki so del C#. Definirajo tudi pogoje za komunikacijo s storitvijo in kateri transport ter kodiranje bosta izbrana.

Servisna rutina (ang. *service runtime*) je nivo, ki vsebuje obnašanja, ki se zgodijo med delovanjem storitve. Dušenje (ang. *throttling*) skrbi za število sporočil, ki se procesirajo; to število se spreminja glede na potrebe in zmogljivost. V tem nivoju je tudi del, ki skrbi za nadziranje napak in sporočanje le-teh odjemalcem. Obnašanje različic (ang. *instance behavior*) definira, koliko različic storitve bo obdelovalo klice.

Sporočanje (ang. *messaging*) je nivo, ki je sestavljeno iz kanalov. Kanal je komponenta, ki procesira sporočila. Obstajata dva tipa kanalov: transportni in protokolni. Transportni berejo in pišejo sporočila iz omrežja, protokolni pa dodatno obdelujejo sporočila z dodatnimi podatki.

Zadnji nivo je množica opcij za gostovanje. Mogoče je gostovanje z IIS ali pa kot že omenjen self-host, kot Windows service ali kot izvedljiva datoteka.



Slika 3.1: Arhitektura WCF ogrodja

3.2 Uporaba in delovanje

Za uporabo WCF ni treba uporabljati nobenih dodatnih knjižnic, ampak je potrebno samo dve referencirati iz .Net ogrodja. To sta System.ServiceModel in System.RunTime.Serialization. ServiceModel vsebuje vso logiko, potrebno za delovanje, Serialization pa ponuja ogrodje za serializacijo podatkov, ki se prenašajo.

Uporabil sem peer-to-peer funkcionalnost za obveščanje med strežnikom in odjemalci. Verjetno ta uporaba ni čisto logična, toda potreboval sem tako obveščanje, kjer strežnik pošilja odjemalcem podatke, ne da jih sami zahtevajo. Lahko bi uporabil tudi full-duplex, ampak odločil sem se za peer-to-peer.

Najprej je treba definirati vmesnik, ki definira metode in njihove parametre. Vmesniku je potrebno dodati atribut ServiceContract. Vmesnik tako služi kot neka pogodba, ki se jo vsi odjemalci držijo in jim omogoča medsebojno komunikacijo.

Metode, ki definirajo, kako se bodo podatki pošiljali, morajo imeti dodaten atribut po imenu OperationContract. Dodatni atributi za metode obstajajo zato, ker vmesnik lahko definira druge lastnosti, ki niso povezane s komunikacijo. Z atributom je tudi mogoče določiti smer pošiljanja, avtomatski odgovor, potrebno avtorizacijo itd.

```
[ServiceContract]
public interface IServiceContract
{
    [OperationContract(IsOneWay = true)]
    void SendMessage(string message);
}
```

Druga naloga je implementacija tega vmesnika pri odjemalcih. Z izdelavo knjižnice, kjer se ta nahaja, lahko to uporabimo pri različnih odjemalcih. Razred, ki implementira ta vmesnik, je treba označiti z atributom ServiceBehavior. Ta označi razred, ki se bo uporabljal in s tem atributom je mogoče nastaviti način izdelave različice razreda. Pomembna lastnost je, da se lahko nastavi ali izdela samo ena različica ali pri vsakem klicu nova. Razvijalcu je v veliko primerih lažje, če se ne ustvari vedno nova, kajti tako lahko obdrži stanje med klici.

V sami implementaciji je treba inicializirati dve komponenti. Prva je gostitelj,

ki je definiran z razredom Host. Temu razredu je treba podati različico definicije odjemalca, preko katere se izvajajo metode ob prejetih klicih, to je naša implementacija pogodbe (ServiceContract). Ko želimo sprejemati sporočila, moramo najprej klicati metodo Start, za prekinitev povezave pa Close. Razred nudi tudi podatke o stanju povezave.

Druga komponenta pa je kanal, ki se bo uporabljal. Ta kanal pridobimo z razredom ChannelFactory. Preko tega kanala se pošiljajo podatki drugim odjemalcem z metodami, definiranimi v naši pogodbi. Tej "tovarni" kanalov povemo tudi ime konfiguracije končne točke. Kanal pridobimo z metodo CreateChanel. Kanal je preprosto še ena implementacija pogodbe, ki se ustvari dinamično in je namenjena pošiljanju namesto sprejemanju. To pomeni, da se na našo implementacijo izvajajo metode pri prejemanju podatkov, za pošiljanje pa potrebujemo ustvarjeni kanal.

```
[ServiceBehavior(InstanceContextMode =
    InstanceContextMode.Single)]
public class Client : IServiceContract {

    private ServiceHost host = null;
    private ChannelFactory<IChatBackend> channelFactory = null;
    private IChatBackend _channel;


    public void DisplayMessage(string message) {
        Console.WriteLine(message);
    }

    public void SendMessage(string message) {
        _channel.DisplayMessage(message);
    }

    public void StartService() {
        host = new ServiceHost(this);
        host.Open();
        channelFactory = new
```

```
            ChannelFactory<IChatBackend>("MyServiceEndpoint");
        _channel = channelFactory.CreateChannel();
    }
}
```

Zadnja stvar, ki je potrebna za delovanje, je konfiguracijska datoteka. To je standardna XML datoteka za aplikacije v .Net okolju. V njej je mogoče definirati nastavitve aplikacije. Nekaj je že obstoječih nastavitev, lahko pa tudi ustvarimo svoje. V to datoteko spada tudi konfiguracija za delovanje WCF.

Definirati je potrebno vezave (ang. *bindings*) med različnimi končnimi točkami, to pomeni določiti, kako bo potekala komunikacija. V mojem primeru je to netPeerTcpBinding. Potrebna je tudi definicija odjemalca in storitve. V mojem primeru je vsak odjemalec oboje. V obeh sekcijah je treba definirati, katero pogodbo in katere vezave se bodo uporabljale v primeru odjemalca ali storitve. Nastaviti je potrebno tudi naslov in ime.

Možna je tudi konfiguracija identitete točke in izbira varnostnih mehanizmov [14]. Identiteta točke se uporablja za avtentikacijo med točko in klicateljem.

Primer konfiguracijske datoteke za mojo aplikacijo.

```
<system.serviceModel>
  <bindings>
    <netPeerTcpBinding>
      <binding name="Wimpy">
        <resolver mode="Pnrp" />
        <security mode="None">
          <transport credentialType="Password" />
        </security>
      </binding>
    </netPeerTcpBinding>
  </bindings>
  <client>
    <endpoint address="net.p2p://MyService"
       binding="netPeerTcpBinding"
       bindingConfiguration="Wimpy"
          contract="WCFClient.IServiceContract"
       name="MyServiceEndpoint" kind="" endpointConfiguration="">
    </endpoint>
  </client>
  <services>
    <service name="">
      <endpoint address="net.p2p://MyService"
         binding="netPeerTcpBinding"
         bindingConfiguration="Wimpy" name="MyService"
            contract="WCFClient.IServiceContract" />
    </service>
  </services>
</system.serviceModel>
```

Poglavje 4

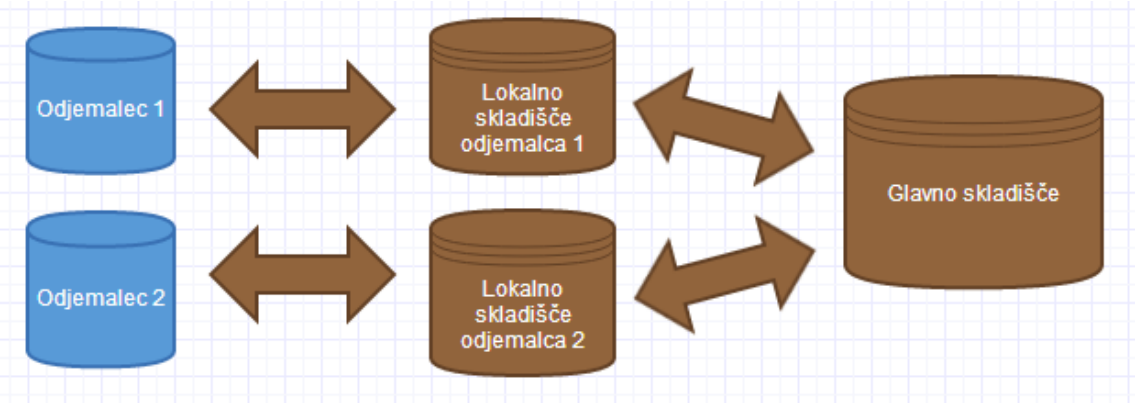
Primerjava obeh načinov implemetnacije

Primerjavo SignalR knjižnice in WCF tehnologije sem predstavil na primeru obveščanja o dejavnostih, ki se dogajajo v orodju za nadzor različic programske kode. Nadzor različic je uporabljen kot strežnik, ki pošilja obvestila o dogajanju. To sem dosegel tako, da sem vrinil kodo, ki izvaja obveščanje med kodo drugih operacij. Odjemalci, ki prejemajo sporočila, so namizna aplikacija, spletna aplikacija in Windows Phone 8.

4.1 Git

Naloga takega orodja je, da beleži vse spremembe in dogajanje s shranjeno kodo kot na primer, kdo jo je spremenil, zakaj in kdaj. Vsi podatki so shranjeni v tako imenovanem skladišču (repository). Pomembnejša naloga je, da vzdržuje stanje, ko imajo vsi najnovejšo različico kode, oziroma, da se ne prepišejo urejanja več programerjev hkrati.

Za boljšo predstavitev knjižnice SignalR sem si želel sam izdelati preprost vmesnik. Svoj vmesnik sem želel imeti zato, da bi lahko dodal kodo, ki izvaja obveščanje, neposredno med kodo za delovanje Git-a. Alternativa temu bi bila uporaba obstoječih vmesnikov. To bi pomenilo, da bi moral v intervalih prečesavati



Slika 4.1: Primer zgradbe skladiščenja kode

zgodovino dogajanja in potem pošiljati obvestila. Ampak tako obveščanje ne bi bilo potisno.

Drugi pogoj je tudi, da obstaja implementacija za .NET ogrodje. Vse te pogoje je izpolnjeval nadzor različic Git.

Git sem izbral tudi zato, ker imam s tem orodjem že nekaj izkušenj in ker sem našel implementacijo, napisano v C#, kar sem lahko uporabil v .Net okolju. Sprva sem želel uporabiti knjižnico GitSharp [10]. To je prosto dostopna implementacija Git-a, ki jo je razvilo nemško podjetje Eqqon. Ker je manj izpopolnjena in se je razvoj na njej končal, je nisem izbral. Možna težava je tudi, da ne bi našel dovolj pomoči, ker je razvoj prekinjen že od leta 2010. Druga knjižnica, ki sem jo našel, je LibGit2Sharp [11], ki sem jo tudi izbral. Uporaba je preprosta, dokumentacija je dovolj dobro napisana, zato sem zlahka dosegel, kar sem želel.

4.1.1 Implementacija vmesnika

Najprej je treba ustvariti centralno skladišče. Za to sem uporabil kar GitHub [12]. To je spletna stran, kjer si lahko vsak izdela svoje skladišče. Stran je namenjena tudi ogledovanju kode. Preprosta uporaba spletne aplikacije GitHub mi je tudi prihranila nekaj časa. Zaradi načina delovanja Git sem moral narediti tudi lokalno skladišče, ampak to se zgradi preko izdelanega vmesnika.

Uporaba LibGit2Sharp je preprosta. Za pridobivanje podatkov iz skladišča ali pošiljanje podatkov v skladišče, je treba najprej ustvariti predstavitev tega

skladišča z razredom Repository. Razred je treba inicializirati z lokacijo skladišča. Za dostop do skladišča je za avtentikacijo in avtorizacijo potreben podpis, ki je zgrajen z uporabniškim imenom in e-poštnim naslovom. Podpis je predstavljen z razredom Signature.

```
public static Signature signature = new Signature("PeterOzbot",
    "peter.ozbot@gmail.com", DateTimeOffset.Now);
Repository repository = new Repository(repositoryPath);
```

Za samo pridobivanje podatkov (ang. *pull*) je potreben razred za skladišče in podpis. Možne so tudi nastavitve o lokalnem združevanju, avtorizaciji in obveščanju o napredku.

```
PullOptions pullOptions = new PullOptions() {
        MergeOptions = new MergeOptions() {
            FastForwardStrategy =
                FastForwardStrategy.Default
        },
        FetchOptions = new FetchOptions{
            Credentials = new Credentials();
        }
    };
MergeResult mergeResult = repository
    .Network.Pull(signature, pullOptions);
```

Rezultat so seveda preneseni in shranjeni podatki v lokalnem skladišču in razred MergeResult, ki nosi podatke o rezultatu združevanja, če se je le-to zgodilo.

Za pošiljanja podatkov (ang. *push*) je treba najprej zapisati podatke v samo datoteko, ki se želi poslati v glavno skladišče. To je del samega ogrodja .Net.

```
File.WriteAllText(filePath, content ?? string.Empty, encoding ??
    Encoding.ASCII);
```

Lokacijo datoteke je treba dodati v seznam, ki je lastnost razreda Repository in klicati funkcijo za potrditev (ang. *commit*). Funkcija potrebuje podpis in komentar, ki je lahko prazen niz.

```
repository.Index.Stage(filename);
Commit commit = repository.Commit("New commit", signature,
    signature);
```

Kot rezultat, je vrnjen razred Commit, ki nosi podatke o avtorju, podatkih o strukturi in sporočila o odgovoru.

4.2 Implementacija za Windows Phone 8

Čeprav ima operacijski sistem, ki ga ima Windows Phone 8 (WP8) [19] enako jedro (kernel) kot operacijski sistem Windows 8, se je pri razvoju aplikacij treba zavedati, da okolje podpira zelo okleščeno različico .Net ogrodja. Sklepal sem, da vseeno podpira dovolj, da bom implementiral, kar sem si zadal in da bom obhode za manjkajoče stvari našel na internetu. Tehnologija je relativno nova in pri prehodu iz verzije 7 na 8 so spremenili ogrodje, ki je na voljo razvijalcu. Zaradi tega je virov in primerov uporabe malo. Aplikacije Windows Phone 8 so zgrajene podobno kot Windows Presentation Framework (WPF) [15]. Vmesnik je zgrajen z Extensible Application Markup Language (XAML) [20], zadaj pa je C#.

Posebnost razvoja za WP8 je tudi to, da je potrebna plačljiva licenca, da se lahko izdelane aplikacije preizkusijo na samem telefonu. Cena licence je bila ob izidu €100, ampak so potem ceno znižali na €15. To priložnost sem tudi sam uporabil. V primeru, da ne bi imel dostopa do naprave, obstaja možnost emulatorja. Emulator uporablja tehnologijo Hyper-V [16]. To ni nič drugega kot okolje za navidezne stroje. Problem je, da ta sistem deluje samo na določenih procesorjih, ki podpirajo virtualizacijo strojne opreme in tako imenovano Data Execution Prevention tehnologijo [17]. To pomeni, da je emulator za WP8 virtualni stroj in zato se mora povezati z drugimi odjemalci kot še en ločen računalnik in ni mogoče izkoristiti prednosti, ki jih nudi razvoj na lokalnem računalniku. To je v glavnem preprostost.

Čeprav je WCF starejši in bolj izpopolnjen in SignalR nova ideja, sem bil bolj v skrbeh, da bom imel težave s SignalR. Izkazalo se je, da je ravno obratno. Implementacija SignalR je potekala brez težav in kode ni bilo treba spreminjati. Z WCF pa sploh nisem uspel implementirati peer-to-peer načina povezovanja. Okleščena

verzija ServiceModela namreč podpira samo osnovne načine povezovanja, ki niso zadoščali temu, kar sem želel narediti [18]. Alternativa bi bila, da bi ročno implementiral oziroma simuliral enako delovanje s periodičnim povpraševanjem odjemalca. Mogoče je podpora te funkcionalnosti s uporabo SignalR odvrnila Microsoft pri tem, da bi to v WP8 podprli tudi z WCF. Zakaj je tako, nisem našel nobene dobre razlage.

4.3 Primerjava implementacij

Pri implementaciji potisnega obveščanja se je treba najprej zavedati, kaj bomo pošiljali in kdo bodo odjemalci. Za izbiro tehnologije, s katero bomo to izvedli, je okolje, v katerem delamo, zelo pomemben faktor. V primeru, če želimo pošiljati podatke kot tok podatkov (data stream) tega ne moremo doseči z izrabo SignalR. Z WCF pa to ni nobena težava. To je zato, ker SignalR izrablja protokol HTTP, WCF pa podpira TCP. V svojem primeru sem pošiljal samo kratka sporočila in obe implementaciji sta popolnoma zadovoljili potrebe takega preprostega obveščanja.

Razlika pa se takoj pokaže pri tem, kdo so odjemalci. Če je odjemalec spletni brskalnik, nastane težava pri izrabi WCF, saj moramo sami implementirati popolnoma vse. Praktično to pomeni, da se WCF ne izrablja v takih primerih. Z uporabo SignalR pa to ni težava. Razvijalci so namreč poskrbeli za samodejno generiranje razredov, potrebnih za delovanje in tudi sama knjižnica deluje na protokolu HTTP.

Razlika med tehnologijami se pokaže tudi pri zahtevnosti izrabe in implementaciji želenega. Kot je vidno iz primerov kode, ki sem jih pokazal pri opisu posameznih implementacij, je izraba SignalR precej bolj preprosta kot WCF. Iz razloga, ker WCF ponuja veliko več možnosti implementacije želenega, je potreben precej daljši čas, da se razvijalec nauči izrabljati ogrodje. Čeprav je WCF obsežnejši, pa je nenavadno to, da ima SignalR vgrajeno sporočanje o napakah, WCF pa takega sporočanja nima. Za nadzor napak med obveščanjem nudi razvijalcu knjižnica SignalR dogodke, ki se prožijo, ko pride do določenih napak. Z izrabo WCF pa mora vsak razvijalec sam implementirati na tak način, da periodično pregleduje, če je povezava še živa.

Pomembna razlika je tudi v tem, da je mogoče z WCF definirati točne pod-

pise metode in tudi tipe parametrov. To pomeni, da ni mogoče, da bi strežnik poslal napačne podatke oziroma take, ki jih odjemalec ne pričakuje. Pri veliki dinamičnosti knjižnice SignalR pa ni definirano niti ime metode, s katero se odjemalec oglaša. Strežnik samo izvaja klice in upa, da so odjemalci pripravljeni na to, kar dobivajo. Enako je v drugo smer, ko odjemalci pošiljajo podatke strežniku. Težava se pokaže, ko želimo spreminjati obstoječo implementacijo in pozabimo povsod spremeniti imena metod ali parametre. Z WCF take napake ne moremo narediti, saj nam to sporoči že prevajalnik.

Poglavje 5

Zaključek

Med izdelavo diplomske naloge sem raziskal in preizkusil dva načina implementacije potisnega obveščanja. Za implementacijo sem najprej opisal in uporabil knjižnico SignalR. V drugem poglavju sem raziskal WCF. Opisal sem samo ključne lastnosti ogrodja WCF, saj je precej bolj obsežno od tega, kar sem sam potreboval. V nadaljnjih poglavjih sem predstavil orodje Git in podal primer implementacije vmesnika za obdelavo akcij nad nadzorom različic kode. Na koncu sem še vrinil kodo za obveščanje v vmesnik za Git in implementiral odjemalce kot namizno aplikacijo in aplikacijo za Windows Phone 8.

Čeprav sem za implementacijo potisnega obveščanja uporabil le določene dele knjižnice SignalR in ogrodja WCF, sem vseeno odkril ključne lastnosti obeh pristopov. Ugotovil sem, da je za hitre rešitve in preproste probleme knjižnica SignalR precej boljša od WCF. Za resnejše delo in bolj varen pristop ter naprednejše rešitve pa izberemo WCF.

Literatura

[1] Windows Communication Foundation Dostopno na:
http://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx

[2] SignalR Domača stran Dostopno na:
http://www.asp.net/signalr/overview/signalr-20/getting-started-with-signalr-20/introduction-to-signalr

[3] WebSocket.org Are you plugged in? Dostopno na:
https://www.websocket.org/

[4] The State of HTML5 Video Report Jeroen Wijering Dostopno na:
http://www.jwplayer.com/blog/the-state-of-html5-video-report-market-share-updates-text-tracks-mediasource-api-and-more/

[5] Server Sent Events One Way Messaging Dostopno na:
http://www.w3schools.com/html/html5_serversentevents.asp

[6] Comet - Model spletnih aplikacij Dostopno na:
http://en.wikipedia.org/wiki/Comet_(programming)

[7] Understanding Ajax Long-Polling Requests Dostopno na:
http://webcooker.net/ajax-polling-requests-php-jquery/

[8] The Official Microsoft IIS Site Dostopno na:
http://www.iis.net/

[9] Owin Dostopno na:
http://owin.org/

[10] GitSharp - Git for .NET and Mono Dostopno na:
http://www.eqqon.com/index.php/GitSharp

[11] LibGit2Sharp Jason "blackant"Long Dostopno na:
https://github.com/libgit2/libgit2sharp

[12] GitHub Dostopno na:
http://github.com/

[13] Service-oriented architecture Definition Dostopno na:
http://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html

[14] WCF podroben opis konfiguracijske datoteke Dostopno na:
http://msdn.microsoft.com/en-us/library/ms731734(v=vs.110).aspx

[15] Windows Presentation Foundation Dostopno na:
http://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx

[16] Hyper-V Dostopno na:
http://technet.microsoft.com/en-us/windowsserver/dd448604.aspx

[17] Data Execution Prevention Dostopno na:
http://en.wikipedia.org/wiki/Data_Execution_Prevention

[18] Omejitve povezovanja WCF za Windows Phone 8 - MSDN Dostopno na:
http://www.silverlightshow.net/items/Connecting-Windows-8-applications-with-services-Part-1-Using-services-to-get-data-in-our-Windows-8-applications.aspx

[19] Windows Phone Dostopno na:
http://www.windowsphone.com/en-us

[20] XAML Overview (WPF) Dostopno na:
http://msdn.microsoft.com/en-us/library/ms752059(v=vs.110).aspx