Reflectieverslag

WIN Load Balancer

Naam: Yigit Baylan  
Studentnummer: 601714  
Docent: Martijn Driessen  
Semester: NOTS  
Course: WIN  
Plaats: Arnhem  
Datum: 6 april 2020

Inhoudsopgave

[1. Inleiding 2](#_Toc37246041)

[2. Ontwerpkeuzes 3](#_Toc37246042)

[2.1. Load verdeling 3](#_Toc37246043)

[2.1.2. Codevoorbeeld 3](#_Toc37246044)

[2.1.3. Alternatieven 4](#_Toc37246045)

[2.2. Persistentie 5](#_Toc37246046)

[2.2.2. Codevoorbeeld 5](#_Toc37246047)

[2.2.3. Alternatieven 6](#_Toc37246048)

[2.3. Health monitoring 7](#_Toc37246049)

[2.3.2. Codevoorbeeld 7](#_Toc37246050)

[2.3.3. Alternatieven 8](#_Toc37246051)

[3. Beroepsproduct 9](#_Toc37246052)

[3.1. Verbeteringen 9](#_Toc37246053)

[4. Conclusie 10](#_Toc37246054)

[Bronnen 11](#_Toc37246055)

[Bijlagen 12](#_Toc37246056)

[Bijlage A - Klassendiagram 12](#_Toc37246057)

# Inleiding

Voor de WIN-course moet een load balancer geschreven worden in C#. Deze load balancer moet op TCP-niveau communiceren met de gekoppelde servers. De communicatie die plaatst vind moet gebeuren als een client naar de webpagina van de load balancer navigeert.

Deze load balancer moet verschillende algoritmes kunnen gebruiken om een geschikte server te kiezen. Deze algoritmes moeten eenvoudig worden uitgebreid. De drie standaard meegeleverde algoritmes zijn; Round Robin, Random en Load. Daarnaast moet er in de instellingen van de load balancer cookie en session persistentie ingeschakeld kunnen worden. Als een van deze persistentie wordt ingeschakeld worden de algoritmes genegeerd en zal de juiste server gekozen moeten worden op basis van cookie of server.

In de afgelopen week heb ik zo een load balancer gebouwd. In dit document zal ik reflecteren op mijn ontwerpkeuzes en de beroepsproduct zelf.

In het hoofdstuk “Ontwerpkeuzes” is per paragraaf elk functionaliteit van de load balancer beschreven en de gemaakte ontwerpkeuzes toegelicht. In de volgende hoofdstuk is de reflectie op de complete beroepsproduct te vinden. Hierin is toelichting op de architectuur, design en implementatie te vinden. Daarnaast is er dit hoofdstuk een aantal verbeter punten voor de load balancer.

# Ontwerpkeuzes

Tijdens het bouwen van de load balancer heb ik bepaalde keuzes gemaakt. In dit hoofdstuk zal ik deze keuzes per onderdeel gaan benoemen en toelichten.

## Load verdeling

De load balancer moet in staat zijn om door middel van een algoritme een server de kiezen die de client verzoek afhandelt. De huidige load balancer ondersteund standaard de volgende drie algoritmes:

* Random
  + De load balancer selecteert willekeurig een beschikbare server
* Round Robin
  + De load balancer selecteert de beschikbare servers één voor één, totdat die bij de laatste server is. Hierna begint de load balancer weer bij de eerste server en herhaalt het proces zich.
* Load
  + De load balancer selecteer de beschikbare server die de minste verzoeken heeft behandeld.

### 2.1.2. Codevoorbeeld

**Random**

Ik heb voor deze algoritme de standaard Random klasse gebruikt. Het is belangrijk dat de geselecteerde server actief is, daarom filter ik eerst alle servers daarna stuur ik een willekeurige server uit de lijst met servers.

public Server GetBalancedServer(ObservableCollection<Server> servers)

{

Random rnd = new Random();

var serversAlive = servers.Where(server => server.isAlive == true);

return serversAlive.Skip(rnd.Next(serversAlive.Count())).FirstOrDefault(); ;

}

**Round Robin**

Hier worden de beschikbare servers gefilterd. Daarna wordt er met behulp van een if statement gecontroleerd of de huidige positie round robin nog goed is gepositioneerd. Is dit niet het geval dan wordt de positie gereset naar 0. Uit de beschikbare lijst met servers wordt de server waarvan de index de Round Robin positie is geselecteerd. De positie waarde wordt hierna meteen verhoogd met één.

public Server GetBalancedServer(ObservableCollection<Server> servers)

{

List<Server> serversAlive = servers.Where(server => server.isAlive == true).ToList();

if (Position >= serversAlive.Count)

Position = 0; // Position is a property of the RoundRobinAlgortihmStrategy

Server server = serversAlive[Position++];

return server;

}

**Load**

Uit de lijst met de beschikbare servers wordt er gekeken welke server de minste requests heeft behandeld. De server met de minste requests wordt terug gegeven.

public Server GetBalancedServer(ObservableCollection<Server> servers)

{

List<Server> serversAlive = servers.Where(server => server.isAlive == true).ToList();

int lowestRequest = -1;

Server serverWithLowestLoad = null;

foreach (Server server in serversAlive)

{

if (server.RequestHandledCount < lowestRequest || lowestRequest == -1)

{

lowestRequest = server.RequestHandledCount;

serverWithLowestLoad = server;

}

}

return serverWithLowestLoad;

}

**Reflection**

BalanceStrategyService zorgt ervoor dat alle loadbalance DLL’s runtime worden ingeladen

public ObservableCollection<IStrategy> GetLoadStrategies()

{

string[] files = Directory.GetFiles(@".\", "\*.dll"); // Defines the path to look for DLL files

foreach (var file in files)

{

Assembly assembly = Assembly.LoadFrom(file);

foreach (Type type in assembly.GetTypes()) // Gets the list of types, the parents are also included in this list

{

if (type.GetInterfaces().Contains(typeof(IStrategy))) // Checks if the type is a type of IStrategy, the balance interface

{

IStrategy balanceStrategy = Activator.CreateInstance(type) as IStrategy; // Creates a instance of the object as an IStrategy object.

\_loadBalanceStrategies.Add(balanceStrategy); // Adds the strategy to the list of strategies.

}

}

}

return \_loadBalanceStrategies;

}

### 2.1.3. Alternatieven

**Weighted Least Connection**

Elk server heeft een prioriteit waarde. De load balancer selecteert één voor één een server. Zodra er een server is met evenveel requests, selecteert de load balancer de server met een hoger prioriteit. (Kemp, z.d.)

**Chained Failover**

De load balancer blijft verzoeken sturen naar dezelfde server, totdat deze inactief is. Zodra deze inactief is gaat die door naar de volgende. De load balancer doet dit tot die bij de laatste server is. Na de laatste server begint die weer bij de eerste server in de lijst. (Kemp, z.d.)

## Persistentie

Een client kan cookies meesturen in zijn HTTP-verzoek. De load balancer kan er voor kiezen om de client een cookie mee te geven, zodat deze bij de eerst volgende verzoek naar dezelfde server wordt geleid. Naast een standaard cookie kan de server een session naar de client versturen.

Als de cookie persistentie is ingeschakeld, dan moet de load balancer controleren of er in het verzoek een cookie is meegestuurd dat de adres van de server bevat. Als die er is dan moet de load balancer deze server selecteren en de verzoek laten behandelen. Is dit niet het geval dan wordt er van op basis van de geselecteerde algoritme een beschikbare server gekozen. Hierbij wordt bij de client een cookie opgeslagen met de verwijzing naar deze server.

Een session wordt aangemaakt door de server. Deze houdt alle data van de client bij, zodat de client niet wordt overspoelt met cookies. De session bevat een ID en wordt verstuurd naar client. De client slaat deze sessionID op in zijn cookies. Bij session persistentie mag de client niet weten met welke server die gekoppeld is. De load balancer is dan verantwoordelijk om de sessionID te koppelen met de juiste server. Bij het ontbreken van een sessionID wordt een server gekozen op basis van de geselecteerde algoritme.

### 2.2.2. Codevoorbeeld

**Cookie based**

Cookie lezen uit een HttpRequest

public string GetCookie()

{

if (HasHeader("Cookie"))

{

List<string> connectedServerList = GetHeader("Cookie").Value.Split(";").Where(cookie => cookie.Contains("ConnectedServer")).ToList();

return connectedServerList.Count == 0 ? "NO\_COOKIE" : connectedServerList[0].Split("=")[1];

}

return "NO\_COOKIE";

}

Cookie plaatsen in een HttpResponse

public string SetServerCookie(string server)

{

string cookie = "ConnectedServer=" + server + "; Path=/; Expires=" + DateTime.Now.AddMinutes(1).ToString("ddd, dd MMM yyyy HH:mm:ss 'GMT'", CultureInfo.InvariantCulture.DateTimeFormat) + "; HttpOnly";

if (HasHeader("Set-Cookie"))

{

DeleteHeader("Set-Cookie");

Headers.Add(new HttpHeader("Set-Cookie", cookie));

}

else

Headers.Add(new HttpHeader("Set-Cookie", cookie));

return cookie;

}

**Session based**

Leest de session cookie uit de HttpResponse.

public string GetSessionCookie()

{

string receivedCookie = "";

if (HasHeader("Set-Cookie"))

receivedCookie = GetHeader("Set-Cookie").Value;

return receivedCookie == "" ? "NO\_SESSION" : receivedCookie;

}

Slaat de session op in het geheugen van de loadbalancer. Het geheugen wordt met behulp van een timer bijgehouden, zodat verlopen sessions hieruit worden verwijderd.

private void CheckForSession(string cookie, Server server)

{

if (cookie.Contains("connect.sid"))

{

List<string> sessionIdList = cookie.Split(";").Where(cookie => cookie.Contains("connect.sid")).ToList();

if(sessionIdList.Count != 0)

{

List<string> expiresList = cookie.Split(";").Where(cookie => cookie.Contains("Expires")).ToList();

string serverString = server.Host + ":" + server.Port;

string id = sessionIdList[0].Split("=")[1];

string expires = expiresList.Count != 0 ? expiresList[0].Split("=")[1] : "Wed, 01 Apr 1990 09:22:42 GMT";

Sessions.Add(new SessionModel(id, serverString, expires));

}

}

}

**Algemeen**

Eerst wordt er gekeken naar persistentie, als dit actief is selecteert de load balancer de server die hierbij hoort.

private Server GetServer(HttpRequest request)

{

if (activePersistanceModel.Type == "COOKIE\_BASED")

{

string cookie = request.GetCookie();

return cookie != "NO\_COOKIE" ? GetServerFromString(cookie) : activeBalanceModel.GetBalancedServer(Servers);

}

else if (activePersistanceModel.Type == "SESSION\_BASED")

{

string sessionID = request.GetServerSessionID();

return sessionID != "NO\_SESSION" ? GetServerFromSession(sessionID) : activeBalanceModel.GetBalancedServer(Servers);

}

else

return activeBalanceModel.GetBalancedServer(Servers);

}

### 2.2.3. Alternatieven

De persistentie is door mijn ontwerpkeuzes in de load balancer geweven. Dit zou als een apart class library geschreven kunnen worden, zodat het bij meerdere projecten gebruikt kan worden. Bovendien ben ik er achter gekomen dat het schrijven van zulke libraries de code veel overzichtelijker maken.

## Health monitoring

Een server kan vastlopen of zelfs uitgeschakeld zijn. De load balancer moet regelmatig controleren of een server nog verzoeken kan behandelen. Als een server geen verzoeken meer kan behandelen, dan mag de load balancer hier geen verzoeken naar sturen. Als een client een sessie of cookie heeft moet de load balancer deze client laten zien dat de server niet beschikbaar is. Hiervoor wordt er gebruik gemaakt van de HTTP-code 503.

### 2.3.2. Codevoorbeeld

De load balancer heeft een functie die door de lijst met servers gaat. Hierin wordt aan de server gevraagd of die nog gezond is of niet. Is dit niet het geval dan wordt de property isAlive aangepast naar false, anders wordt die op true gezet.

public void CheckServersHealth()

{

foreach (var server in Servers)

{

try

{

if (server.IsHealthy())

{

Dispatcher.Invoke(() =>

{

server.isAlive = true;

CollectionViewSource.GetDefaultView(Servers).Refresh();

});

}

else

{

Dispatcher.Invoke(() =>

{

server.isAlive = false;

CollectionViewSource.GetDefaultView(Servers).Refresh();

});

}

}

catch (Exception)

{

Dispatcher.Invoke(() =>

{

server.isAlive = false;

CollectionViewSource.GetDefaultView(Servers).Refresh();

});

AddLog(LogType.Server, "Server " + server.Host + " on port " + server.Port + " is unavailable");

}

}

}

De health check van de Server.

public bool IsHealthy()

{

Connect();

string requestString = "GET / HTTP/1.1\r\n" +

"Host: " + Host +

"Connection: close" +

"\r\n\r\n";

byte[] requestBuffer = Encoding.ASCII.GetBytes(requestString);

Send(requestBuffer);

byte[] responseBuffer = Receive(1024);

HttpResponse response = HttpResponse.Parse(responseBuffer);

Disconnect();

return response.FirstLine.Contains("200 OK");

}

### 2.3.3. Alternatieven

Er wordt nu alleen een verzoek gestuurd naar de server om te controleren of die bestaat of niet. Hierbij wordt er gebruik gemaakt van de HttpResponse klasse. Dit zou weg gelaten kunnen worden en de server zou op een ander manier moeten lezen of de ontvangen data correct is. De hele class library heeft hierdoor een dependency naar de HTTP class library.

De load balancer zou ook data bij kunnen houden om te zien wanneer een server uit gaat. Op basis van deze data zou die een voorspelling kunnen doen en zelfstandig besluiten of de server gezond is of niet. De data kan op basis van de tijd zijn dat een server uitgaat of op basis van het verzoek dat de server verwerkt. Als de load balancer met behulp van de self learning method alle servers ongeschikt verklaard zal die een fallback methode moeten uitvoeren. Deze methode gaat dan, zoals nu elk server een verzoek sturen om te analyseren of de servers gezond zijn of niet.

# Beroepsproduct

Tijdens het bouwen van dit beroepsproduct had ik een tunnelvisie over het gebruik van reflection. Maar dit veranderde al vrij snel toen ik mij hierin had verdiep. Als de load balancer uitgebreid moet worden terwijl die in productie is kan dit hier vrij eenvoudig mee. Tijdens het runnen kan je namelijk DLL’s toevoegen in de daarvoor bestemde folder. Hiervoor zou er nu wel een knop toegevoegd moeten worden met deze functionaliteit. Bovendien kan een extern programmeur zelf een balance strategy kunnen toevoegen. De strategy moet dan de interface IStrategy gaan implementeren.

Waar ik persoonlijk erg trots op ben is het verspreiden van alle functionaliteiten voor de loadbalancer. Elk belangrijk onderdeel uit de proxy is hierin terug gekomen als een eigen class library. Zie hiervoor de het ontleden van de HTTP requets en het communiceren met een server en client. Deze class libraries kunnen nu herbruikt worden voor een ander project dat draait op .NET Core 3.1. (Zie Bijlage A - klassendiagam)

De design keuze voor het implementeren van een strategy pattern ben ik het meest trots op. Dit heeft niet alleen het implementeren ervan makkelijker gemaakt, maar heeft ook voor een duidelijk overzicht gecreëerd.

## Verbeteringen

De model LoadBalancerModel is nu groot geworden. Maar dit komt omdat de functies niet meer dan vijftienregels code bevatten. Ik had dit graag nog opgesplitst willen hebben. De funcionaliteiten horen wel bij een typisch load balancer, maar hierin zou de LoadBalancerModel een controller functie kunnen hebben.

De health monitor controleert nu om de 2 seconde de servers. Het zou mooi zijn als hiervoor machine learning wordt toegepast. Dit zorgt voor minder stress bij de load balancer en heeft de client minder last van error pages. De load balancer weet immers al dat de server elk moment kan uitvallen en gaat geen verzoeken meer naar deze server sturen.

# Conclusie

De load balancer is in vergelijking tot de vorige beroepsproducten eenvoudiger te bouwen. Dit kwam voornamelijk omdat de voorkennis er al was. Ik kon veel functionaliteiten overnemen uit mijn eigen proxyserver. De HTTP en TCPCommunication class libraries komen hier ook vandaan. Dit heeft mij vrij veel geholpen bij het bouwen van dit product.

Voor het in productie nemen van de load balancer wil ik aanbevelen om de health monitoring uit te breiden met machine learning. De load balancer kan soms verzoeken sturen naar server die al is uit gevallen, maar doordat de health monitor check om de 800 milliseconde wordt gedaan kan het voorkomen dat de server al uit is voordat de load balancer dit weet.

# Bronnen

Hemant, R (2018, 23 juni). What is the difference between session and Cookies? Geraadpleegd op 4 april 2020, van <https://medium.com/@rawat.hemant27/what-is-the-difference-between-cookie-cache-and-session-d6f468a9b4b3>

Kemp (z.d.). Load Balancing Algorithms and Techniques. Geraadpleegd op 7 april 2020, van <https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques/>

# Bijlagen

## Bijlage A - Klassendiagram

