Kritische reflectie load balancer

Termen:

Term	Betekenis
client	Wanneer iemand een vraag voor de website doet wordt dit gezien als een cliënt voor de loadbalancer/server
Gewenste server	De server die door de algoritme gekozen is.
Health check	Een methode die kijken welke status een server heeft
Status: Normal	Er is niks mis met de server
Status: Busy	Server heeft het druk. Wordt gesimuleerd in de code door een timeout van 2000 milliseconden.(op server zelf)
Status: Down	Server is heel druk. Wordt gesimuleerd door een timout van 3000 miliseconden (op server zelf)
Status: Not running	Status voor wanneer een server offline is.

Kwaliteit code

Health check

De health check wordt om X aantal seconden gedaan. Dit betekent dat ieder X seconden gekeken wordt of en welke servers online zijn en welke status ze hebben. De servers die online zijn kunnen bereikt worden en de beste server handelt de request af.
Het volgende probleem kan zich voordoen: de gewenste server gaat offline voordat de health check is uitgevoerd/afgerond en dan kan de gewenste server niet bereikt worden en de request niet afhandelen.

Mocht deze situatie zich voordoen dan wordt een bericht naar de client gestuurd dat de server net offline is gegaan. Tegelijkertijd wordt de status van de gewenste server naar offline gezet. Als de client de website refresht zal er een nieuwe server gekozen worden die wel online is en de request afhandelen. Mocht alle servers offline zijn dan zal hier een bericht van getoont worden.

Een andere oplossing is dat, bij het falen van de request naar de gewenste server, dat er opnieuw gezocht moet worden naar de beste server en deze gebruiken voor het afhandelen van de request. Dan ontstaat het volgende probleem: als alle servers offline gaan dan zal er door de loadbalancer continu gezocht worden naar een server die bruikbaar is. Hierdoor zal de loadbalancer sloom worden of eventueel vast lopen.

Het is beter de client hierover te informeren met een bericht over de status van de servers zodat de client actief de browser refresht dan dat dit automatisch continu gebeurt. Er wordt een thread aangemaakt voor het afhandelen van een request. Als het zoeken naar de gewenste server mislukt en er continu gezocht wordt naar een vervangende server zal deze thread nooit afsluiten. In deze situatie zullen er telkens threads aangemaakt worden en niet afgesloten worden. Dit zal uiteindelijk te veel memorie in beslag nemen waardoor de loadbalancer zal vastlopen.

Bijhouden van sessies

De sessies worden bijgehouden bij de server klasse. Dit betekent dat elke server een lijst van sessies bij gaat houden op de loadbalancer zelf. Met een kleinschalig programma zal je hier niet tegen problemen aanlopen. Zodra je duizenden gebruikers wilt afhandelen op je loadbalancer wordt dit een probleem. Nu wordt er veel memory gebruikt dat eigenlijk onnodig is. Per client wordt een session bijgehouden waardoor de loadbalancer veel memorie moet reserveren voor de informatie van iedere sessie.

Sessies zouden beter op een database kunnen worden opgeslagen. Door dit te doen wordt deze informatie niet in de loadbalancer bewaart en reserveert de loadbalancer hier geen memory voor. Alleen op de momenten dat informatie nodig is kan deze uit de database gehaald worden. Hierdoor voorkom je dat deze informatie bewaard wordt als dit niet nodig is. Wanneer informatie van de sessie nodig is kan het worden opgehaald uit de database. Hierdoor wordt de loadbalancer optimaal benut. Doel van de loadbalancer is de werkdruk te verdelen tussen servers en niet het bewaren van data van sessies. Nu is het ook niet meer nodig om een lijst van sessies bij te houden in de klasse want nu kan er met de database gelijk gezocht worden naar een sessie en retourneert alleen 1 sessie mits die bestaat. Dit is net zo efficiënt op een grootschalige app (veel gebruikers tegelijkertijd) dan als op een kleinschalig (weinig gebruikers tegelijkertijd).

Hiermee los je gelijk ook het volgende probleem op. Mocht de situatie voorkomen dat de loadbalancer crasht dan zou je alle sessies kwijt zijn als deze bewaart wordt op de loadbalancer

zelf. Door de sessies op te slaan in een database zijn deze veilig mocht de loadbalancer crashen.

IAlgoritme

Deze interface moet gebruikt worden om een algoritme toe te voegen. Deze Interface bevat voor nu alleen de methode getBestServer. Deze methode heeft een lijst met servers nodig. het bevat tevens een parameter voor cookies. Deze is bij default null. Er is hiervoor gekozen zodat het altijd mogelijk is om een algoritme toe te voegen onafhankelijk van of er een session/cookie nodig is. Hierdoor is het mogelijk om een algoritme toe te voegen die cookie informatie wel of niet gebruikt.

Een issue hiermee is dat er nu wel elke keer een cookie en sessie geset wordt . Na het ophalen van de reponse van de server wordt er gekeken als de client al een cookie bevat en als deze wijst naar de gewenste server. Als dit het geval is dan zal er geen nieuwe cookie geset worden.. Het zou mogelijk zijn om de IAlgoritme zodanig aan te passen dat er alleen een cookie en sessie geset wordt zodra er gebruik wordt gemaakt van een cookie/session based algoritme.. Dit zou je dan in de algoritme zelf willen doen. Hiervoor moet de algoritme toegang hebben op de eventuele gesete cookie informatie en de response headers om hier eventueel een nieuwe header te setten. Hierdoor zou meer informatie heen en weer gestuurd worden en dit is niet efficiënt voor de loadbalancer.