Finagle

Het cement tussen de stenen

In de wereld van microservices zijn er meerdere services, die met elkaar communiceren. De communicatie tussen de services moet snel zijn en om kunnen gaan met fouten. Finagle is een networking library, die hulp biedt bij deze uitdagingen voor diverse protocollen. Door het gebruik van deze library is het relatief eenvoudig om aanroepen tussen systemen resilient te laten werken. Finagle biedt hiervoor functionaliteit zoals retries, timeouts en circuit breaking.

Het Finagle project is in 2010 begonnen bij Twitter. Finagle wordt bij Twitter overal toegepast om verschillende services te verbinden. Nadat het door Twitter als open source beschikbaar is gesteld, is het ook door andere bekende bedrijven, zoals SoundCloud, ING en Tumblr opgepikt. Inmiddels is het het nummer 8 Scala-project op GitHub. De library is geschreven in Scala, maar ook uitstekend bruikbaar voor de andere JVM-talen. Twitter noemt het zelf: "*Finagle is an extensible RPC system for the JVM, used to construct high-concurrency servers.*" Door het onderliggende principe '*Your server as a function*' is de library modulair opgezet en is het eenvoudig om er gedrag aan toe te voegen.

RPC

Finagle wordt gepresenteerd als een RPC systeem. Hierbij wordt als losse definitie van RPC gebruikt:

- Stuur een request;
- Wacht (non-blocking);
- Response komt terug met *Succes* of *Failure*.

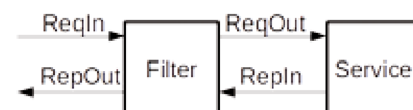
RPC is niet gebonden aan een specifiek protocol, maar is een principe. Finagle ondersteunt deze definitie van RPC via diverse protocollen, waaronder HTTP, Thrift en MySQL.
Deze definitie van RPC is vastgelegd in de centrale **Service** trait (zie **Listing 1**).

In essentie wordt een RPC call gedefinieerd als een functie van **Request** naar een **Future** van **Response**. Het hele Finagle principe is gebaseerd op functie-compositie, zodat een simpel model ontstaat, waarbij verantwoordelijkheden netjes gescheiden worden per functie.

De Twitter **Future** is een bekend concept en lijkt veel op de **scala.concurrent.Future** uit de Scala library. Eigenlijk was de Twitter **Future** er al voor de **Future** in Scala was opgenomen. Het belangrijkste verschil met de Scala **Future** is dat de **twitter.util** variant gecancelled kan worden. Dit cancellen is niet voor alle protocollen relevant. Voor HTTP bijvoorbeeld kan een eenmaal gedane call naar een remote systeem niet gecancelled worden. Maar bij Mux kan een cancellation nog wel worden gepropageerd. Om een service te verrijken, wordt gebruik gemaakt van **Filters**. **Listing 2** toont de definitie van deze **Filter** class.

Wat het eigenlijk zegt, is dat je met een **Filter** ook de input en output types kunt wijzigen (zie **Figuur 1**).

Dit kan bijvoorbeeld gebruikt worden als een request en response object moeten worden



Figuur 1: Filter

```
trait Service[Req, Rep] extends (Req => Future[Rep])
```

Listing 1: Service trait

```
abstract class Filter[-ReqIn, +RepOut, +ReqOut, -RepIn]
  extends ((ReqIn, Service[ReqOut, RepIn]) => Future[RepOut])
```

Listing 2: Filter class

Hinse ter Schuur is werkzaam als Senior Software Engineer bij SDB Java

```
class MarshallingFilter(objectMapper: ObjectMapper)
  extends Filter[QueryObject, ResponseObject, String, String] {
  def apply(req: QueryObject, service: Service[String, String]):
Future[ResponseObject] = {
    val queryString = objectMapper.writeValueAsString()
    service(queryString).map { resp =>
      objectMapper.readValue(resp, classOf[ResponseObject])
    }
  }
}
```

Listing 3: Voorbeeld filter

```
trait SimpleFilter[Req, Rep] extends Filter[Req, Rep, Req, Rep]
```

Listing 4: SimpleFilter trait

```
val loggingFilter = new SimpleFilter[Request, Response] {
  override def apply(request: Request, service: Service[Request, Response]):
Future[Response] = {
    debug(s”Logging request $request”)
    service(request)
  }
}
```

Listing 5: LoggingFilter

omgezet naar JSON, zie **Listing 3** voor een voorbeeld.

Voor simpele filters waarbij niets aan de types wordt gewijzigd, is een **SimpleFilter** trait beschikbaar. De definitie hiervan staat in **Listing 4**.

Op basis van de **SimpleFilter** kan bijvoorbeeld een **LoggingFilter** worden gemaakt, zoals in **Listing 5**.

Filters en services kunnen worden samengesteld door de **andThen** method van **Filter**. Filters worden samengesteld van links naar rechts. **Listing 6** laat zien hoe de logging- en timeout filter gecombineerd kunnen worden. **Figuur 2** laat deze opgebouwde stack visueel zien.

Deze basiselementen zijn voldoende om calls naar andere systemen doen. De uitdaging is om dit *resilient* te doen.

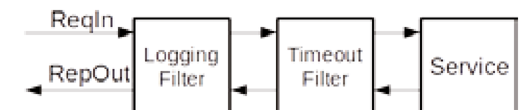
Resilience

Resilience is niet iets wat je in één service implementeert. Het is een manier van denken voor het hele systeem; over de verschillende services heen en tussen de verschillende services onderling. Finagle bevat een aantal resilience-modules voor zowel server- als clientkant, zoals onder andere: Retries, Timeouts, Service Discovery, Load balancing en Circuit breaking. De standaard generieke modules zitten in de core van Finagle en zijn toepasbaar op diverse protocollen.

Clients

Hoewel Finagle componenten voor zowel het implementeren van servers als clients aanbiedt, gaat dit artikel in op het gebruik van de client componenten.

Als gebruiker hoef je niet na te denken welke modules je in welke volgorde aan elkaar moet koppelen om een resilient systeem te bouwen. Voor de ondersteunde protocollen worden **Client** objecten aangeboden. Een **Client** is een complete stack van resilience modules. De modules zijn voor HTTP onderverdeeld in drie sub-stacks, zoals te zien is in **Figuur 3**. De client-stack zorgt voor *name-resolving*, *retries* en *load balancing*. De endpoint-stack zorgt voor *connection pooling* met daarbij *circuitbreaking* modules per endpoint. De connection-stack zorgt voor het opzetten van de fysieke connectie en het transportprotocol. Voor het



Figuur 2: Stacked service

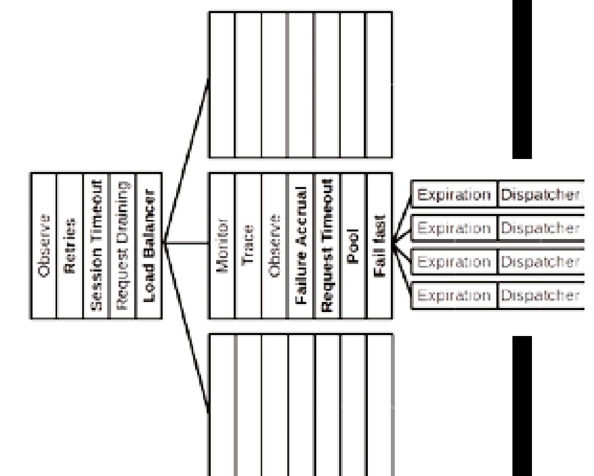
transport wordt gebruik gemaakt van de Netty library.

Voor de ondersteunde protocollen kan een configureerbare client worden geinstantieerd door **<protocol>.client** aan te roepen. Dit levert een client op, die opereert op een protocol specifiek request en response type. **Listing 7** laat een voorbeeld zien voor HTTP.

Configuratie

Alleen een client maken is niet voldoende. De configuratie van de client moet worden afgestemd op het gebruik. Configureren van Finagle is gebaseerd op twee principes: *common things are easy to do* en *uncommon things possible to do*.

Het eerste principe wordt ondersteund door de *conventional API*, waarbij **client.with...** methodes aangeboden worden, bijvoorbeeld **client.withRequestTimeout**.



Figuur 3: Client stack

```
val stackedService = loggingFilter andThen timeoutFilter andThen httpService
```

Listing 6: Samenstellen filters en services

```
val httpService = Http.client.newService(“example.com:80”, “MyService”)
val response: Future[Response] =
httpService(Request(Method.Get,”/reverse?text=Hello”))
// Do something useful with response
```

Listing 7: Http service

```
val httpService = Http.client
  .newService(“host1.example.com:80,host2.example.com:80”, “MyService”)
```

Listing 8: Static hosts

```
val httpService = Http.client
  .newService(“zk!zookeeper.example.com:2181!/someapi”, “MyService”)
```

Listing 9: Discovery via ZooKeeper

Het tweede principe wordt ondersteund door de *expert API*. Hiervoor wordt de `client.configured(param...)` aangeboden, waarbij dan zelf de juiste parameter class gezocht moet worden.

Service discovery

In typische omgevingen zijn er meerdere replica's van een aan te roepen systeem. Om deze replica's te kunnen vinden, biedt Finagle een service discovery module. Met deze module kunnen de replica's statisch of dynamisch ontdekt worden.
Bij het instantiëren van een service wordt niet een URL, maar een `name` opgegeven. Deze `name` wordt door een *resolver* geresolved naar één of meerdere adressen.

Resolvers kunnen worden opgegeven door de identifier van de resolver op te geven bij de name; bijvoorbeeld "inet!". Als geen specifieke *resolver* is opgegeven, dan wordt de `InetResolver` gebruikt. Deze gebruikt de kommagescheiden hostname:port combinaties als set van adressen waarnaar geconnect moet worden. Een voorbeeld hiervan is **Listing 8**.

In de praktijk zal het voorkomen dat vooraf niet bekend is welke replica's er voor een aan te roepen service zijn. Om deze dynamiek te ondersteunen, kan gebruik gemaakt worden van service discovery systemen. Finagle heeft standaard ondersteuning voor ZooKeeper. Door de module 'finagle-serversets' op het classpath te plaatsen, kan gebruik gemaakt worden van de 'zk'-resolver. Hierbij kunnen ZooKeeper hosts worden opgegeven en het pad van de gewenste service. De code in **Listing 9** laat zien hoe adressen van de 'someapi' worden opgevraagd via ZooKeeper. De service zal gebruik maken van de dynamische set van adressen. Als een replica zich afmeldt bij ZooKeeper, dan wordt deze niet meer door de Finagle client aangeroepen.

Load balancing

Om verkeer naar de aan te roepen replica's te verdelen, wordt gebruik gemaakt van een load balancing module. Deze module zit in de standaard client stacks en balanced over de bekende endpoints. Het mechanisme voor load balancing is instelbaar en bestaat uit twee delen: een *load metric* en een *distributor*. Standaard wordt gebruik gemaakt van de *Power of Two Choices (P2C)* distributor en de *Least Loaded* load metric. Dit is een goede default voor veel gevallen. Een goed alternatief voor als er met relatief lage load naar een groot aantal replica's wordt geconnect, is de *Aperture* distributor. Deze zorgt ervoor dat een aantal replica's "warm gehouden worden".

Circuit breaker

Om verkeer naar kapotte replica's te voorkomen en replica's de kans te geven zich te herstellen, heeft Finagle een *circuit breaking* module.
Circuit breakers zorgen ervoor dat er geen calls worden gedaan naar replica's, waarvan bekend is dat ze *down* zijn. Dit maakt het voor de load balancer mogelijk om een andere node te kiezen. Het circuit breaking mechanisme in de Finagle endpoint-stack bestaat uit twee delen: de `FailFast` en de `Failure Accrual` modules.
De `FailFast` is een 'session-driven' circuit breaker. Deze zal een node als down markeren als de *connection* niet gemaakt kan worden. De `FailFast` module probeert op de achtergrond herhaaldelijk de connectie opnieuw op te bouwen. Deze module kan niet geconfigureerd worden, alleen worden uitgeschakeld (zie **Listing 10**).

De `Failure Accrual` is een ‹request-driven' circuit breaker. Zodra een bepaald aantal requests gefaald is, of het succespercentage onder een drempelwaarde komt, zal het endpoint als *down* worden gemarkeerd voor een gedefinieerde tijd. Standaard zal deze module het endpoint na 5 opeenvolgende failures als *down* markeren. Het endpoint wordt dan als *dead* gemarkeerd. Zodra de `markDeadFor` tijd verstreken is, zal deze module weer requests doorlaten om te detecteren of het endpoint up is. Zo niet, dan wordt het endpoint voor een langere tijd als *dead* gemarkeerd. De `Failure Accrual` is instelbaar, maar valt onder de expert level API en is lastiger om in te stellen. De code in **Listing 11** vereist dat over 100 calls de succes rate 95% moet zijn. Zo niet, dan wordt het endpoint telkens voor 10 seconden als 'dead' gemarkeerd.

Retries

Met bovenstaande modules kan Finagle load op een slimme manier verdelen en replica's, die down zijn, vermijden. Echter, het kan nog steeds voorkomen dat er zaken misgaan. Om de gebruiker toch een resultaat te geven, kan een poging gedaan worden om dezelfde operatie op een andere replica uit te voeren door een automatische retry.

```
val httpService = Http.client
  .withSessionQualifier.noFailFast
  .newService(“zk!zookeeper.example.com:2181!/someapi”, “MyService”)
```

Listing 10: Disablen FailFast

```
val httpService = Http.client
  .configured(FailureAccrualFactory.Param(() => FailureAccrualPolicy.successRate(
      requiredSuccessRate = 0.95,
      window = 100,
      markDeadFor = Backoff.const(10.seconds)
  )))
.newService(“zk!zookeeper.example.com:2181!/someapi”, “MyService”)
```

Listing 11: Configureren Failure Accrual

In de client stack is hiervoor standaard een `Retries` module opgenomen. Doordat deze boven de load balancer module zit, kunnen gefaalde requests naar een andere node geretried worden. Het is niet altijd veilig om een retry uit te voeren. Denk bijvoorbeeld aan een replica, die een operatie wel heeft uitgevoerd, maar fout gaat bij het sturen van de response. Dan wil je geen automatische retry. De module retried alleen als er een `RetryableFailure` opgetreden is. Denk aan een exceptie, die optreedt voordat er bytes naar het endpoint zijn verzonden.

De `Retries` module wordt geconfigureerd met twee parameters: het `RetryBudget` en een backoff policy. Met het `RetryBudget` wordt gespecificeerd hoeveel retries gedaan kunnen worden in verhouding tot het totale aantal requests. Met de backoff policy kan worden opgegeven met hoeveel tussentijd de retries gedaan moeten worden.

De `Retries` vangt alle fouten uit onderliggende modules op, dit zijn echter niet-applicatieve fouten. Het kan ook wenselijk zijn om retries te doen voor applicatieve fouten, bijvoorbeeld bij specifieke HTTP foutcodes. Qua netwerk zijn dit geslaagde calls, maar er is toch iets mis. Om hiervoor dan retries te doen moet de `RetryFilter` voor de service worden geplaatst. **Listing 12** laat zien hoe een `RetryFilter` geconfigureerd kan worden.

Om retry-storm te voorkomen, wordt geadviseerd om de `RetryFilter` en de `Retries` module dezelfde `RetryBudget` instance te geven.

Timeouts

Timeouts kunnen op verschillende plaatsen optreden en kunnen ook apart geconfigureerd worden. De sessie timeout bepaalt hoe lang er gewacht wordt op een beschikbare sessie/service. De request timeout geeft aan hoe lang een request open mag staan. Standaard zijn beide timeouts ingesteld op oneindig (`Duration.Top`), omdat Finagle niet de specifieke kennis van de applicatie heeft om defaults in te stellen.

Als er een request timeout optreedt, dan wordt deze niet geretried, omdat niet zeker is of het request aangekomen is. **Listing 13** laat zien hoe sessie timeout en request timeout ingesteld worden.

```
val policy: RetryPolicy[(Request, Try[Response])] =
  RetryPolicy.backoff(Backoff.equalJittered(10.milliseconds, 10.seconds)) {
    case (_, Return(rep)) if rep.status == Status.InternalServerError => true
  }
val retryFilter = new RetryFilter[Request,Response] (
  retryPolicy = policy,
  timer = DefaultTimer.twitter,
  statsReceiver = statsReceiver,
  retryBudget = retryBudget
)
val stackedService = retryFilter andThen httpService
```

Listing 12: Retry filter

```
val httpService = Http.client
  .withSession.acquisitionTimeout(5.seconds)
  .withRequestTimeout(1.second)
  .newService(“zk!zookeeper.example.com:2181!/someapi”, “MyService”)
```

Listing 13: Timeouts

Mijn ervaringen

Finagle blijkt een krachtige library, die veel resilience maatregelen ondersteunt. Het is modulair opgezet en eenvoudig in gebruik wat betreft het aanroepen van services.

De manier van configureren is eenvoudig in Scala en in de meeste gevallen ook eenvoudig vanuit Java. De uitdaging bij het configureren, is het bepalen van de juiste waardes. Om tot een goed resilient systeem te komen, is kennis van de omgeving en kennis van de *internals* van Finagle nodig. Het is belangrijk om de invloed van de verschillende instellingen goed te begrijpen. Het is niet triviaal wat voor instellingen je moet kiezen.

De documentatie is voldoende qua concepten, maar er ontbreken praktische voorbeelden. Wel worden vragen beantwoord in de Finagle Google group en via Gitter. ■

REFERENTIES

Twitter Inc. Finagle User Guide: **http://twitter.github.io/finagle/guide/**
Vladimir Kostyukov. Finagle 101: **http://vkostyukov.net/posts/finagle-101/**
Marius Eriksen. Your Server as a Function: **https://monkey.org/~marius/funsrv.pdf**
Muki Seiler. A Beginners Guide for Twitter Finagle: **https://medium.com/@muuki88/a-beginners-guide-for-twitter-finagle-7ff7189541e5**
Twitter Engineering. Finagle: A Protocol-Agnostic RPC System: **https://blog.twitter.com/2011/finagle-a-protocol-agnostic-rpc-system**
Marius Eriksen. RPC Redux: https://www.youtube.com/watch?v=RN46xdzzECA
Vladimir Kostyukov. Finagle Under the Hood: **https://www.youtube.com/watch?v=kfs-dtbG0kY**
Marius Eriksen. Systems Programming at Twitter: **http://monkey.org/~marius/talks/twittersystems/**
Alessandro Vermeulen: Towards Finagle at ING Bank: **https://skillsmatter.com/skillscasts/6953-towards-finagle-at-ing-bank**
Argha Chattopadhyay. Lessons in resilience at SoundCloud: **https://developers.soundcloud.com/blog/lessons-in-resilience-at-SoundCloud**