OsgpLF (Osgp Light & Lightning Fast)

Dit is mijn POC voor een andere opzet van het OSGP project, om te kijken of met andere libraries de belangrijkste use case van OSGP, met minder code kan worden gerealiseerd en bovendien sneller is.

De (m.i.) belangrijkste use-case is om zo snel mogelijk heel veel (bundled) request naar heel veel verschillende devices te versturen.

De nadelen van de huidige opzet van OSGP zijn mijns inziens:

1) Alle requests worden (assynchroon) 1 voor 1 met het Soap protocol naar het platform gestuurd. Dit is traag en vraagt om heel veel bandbreedte, omdat de xml berichten groot zijn.
2) Er worden in verschillende lagen (platform / core / protocol adapter) heel veel transformaties heen- en terug uitgevoerd (var dto naar valueobject naar jsm-message etc) die nergens voor nodig zijn.
3) Omdat de communicatie tussen de verschillende lagen via ApacheMQ loopt, lijkt het me moeilijk om dynamisch (on-demand) extra machines in de 'onderste' protocol adapter laag toe te voegen.
4) Het duurt relatief lang (minuten) om als developer de applicatie te starten. Bovendien is men veel tijd kwijt aan 'maven-updaten , clean-en' etc.

Gewoon 'omdat het kan', ben ik daarom (in mijn vrije tijd) aan het experimenteren geslagen om bovengenoemde use case op een heel andere manier te implementeren. Daarbij heb ik de volgende tools gebruikt:

1) In plaats van SoapUI gebruik ik de strreaming api van gRpc (Google Remote procedure call). Grpc heeft de voordelen van SoapUI:
   - een contract, in de vorm van een proto file
   - ingebouwde security mbv certificaten etc
   - mogelijkheid om alle benodige code te genereren.
2) In plaats van inkomende requests (meerdere malen) te transformeren, stuur ik de protobuf files (waarmee gRpc werkt) zonder transformatie door het platform. Daarom is de Core laag volkomen agnostic voor welk protocol dan ook!
4) In plaats van ApacheMQ heb ik gebruik gemaakt van Akka actors. Deze zijn zowieso wat makkelijker voor developers (geen transformatie van/naar jms-message), maar nog belangrijker is dat, daar waar nodig ad-hoc actors worden aangemaakt om de schaalbaarheid te vergroten.
4) Omdat er toch geen Dto's en entity objecten en zo beschikbaar zijn, heb ik gebruik van een NoSQL database, om data te lezen en schrijven. In dit geval de object datbase Perst (maar ik ook met implementaties van Redis en Aerospike geexperimenteerd).

Globale opzet

De programma opzet is heel globale zin identiek aan die van OSGP. Er zijn drie lagen, die ieder corresponder met een Eclipse project :

- `Platform`
- `Core`
- `Dlms (Protocol adapter Dlms)`

Deze drie projecten hebben ieder een main() class die afzonderlijk kan worden gestart en gestopt.

Daarnaast is er nog een `Protobuf` project, hierin zitten met name de gegenereerde protobuf classes, en een `Shared` (en `Shared.dlms)` project waarin een paar hulpclasses zitten die in bovenstaande drie projecten hergebruikt kunnen worden.
Tenslotte is er nog een `Test-client` project, waarmee request(s) naar het systeem kunnen worden afgevuurd.

NB:
Omdat de berichten die naar het platform worden gestuurd, volkomen generiek zijn, is m.i. het Platform project (te vergelijk met protocol-adapter-ws-dlms) overbodig, en kan die functionaleit ook in de Core laag woden afgehandeld!.

Geimplementeerd use-case

Met de test-client van hierboven, kan de hoofdcase worden nageboots, waarbij een bundled request eerst via platform -> core -> protocol-adapter wordt gestuurd. Daar wordt afhankelijk van het request de correspondere device actie worden aangeroepen. Deze code as-is gekopieerd uit het *Protocol-Adapter-Dlms* project, en maakt dus ook gebruik van device simulator. Daarnaast is ook een stub waarmee wachttijden en fout condities kunnen worden gesimuleerd. De gebundelde antwoorden worden dan weer de omgekeerde weg terug gestuurd en opgeslagen in de database, waarna deze kan worden opgehaald.

Verschillen tussen OSGL en osgpLF

De verschillen tussen OSGP en osgp-lite zijn als volgt:

- De hoeveelheid code is veel kleiner. Op dit moment is deze (incomplete) codebase nog geen 15% ivm OSGP (5K versus 35K locs). Het idee is dat in core en platform geen aanpassingen nodig zijn als een actie wordt toegevoegd, en slecht minimale (configuratie) aanpassingen als een nieuwe adapter wordt toegevoegd.
  Ik denk dat zelfs de complete laag, waar de requests binnenkomen overbodig is!
- In plaats van minuten, duur het opstarten van de drie afzonderlijke projecten slechts enkele seconden!
- De snelheid om grote aantallen request door het systeem te halen, is veel groter, met name omdat het versturen van request en responses heen en weer tussen de verschillende lagen, vele malen

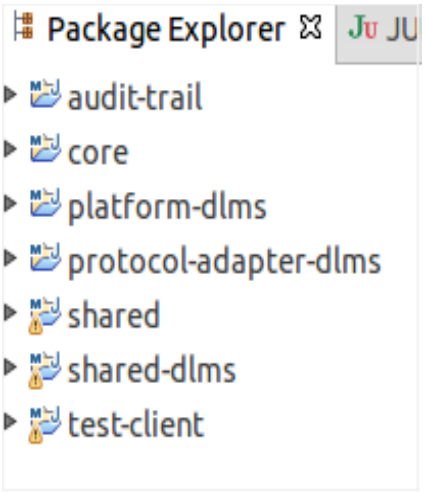
sneller gaat, dankzij de streaming api van gRpc.
Verder zijn er geen (overbodige) transformaties nodig, en zijn de database acties veel sneller.
In mijn virtual box kan ik 10K device-operations binnen een halve minuut verwerken.
(zie ook appendix voor resulaten van een 100K job)

Aantekeningen voor de ontwikkelaar

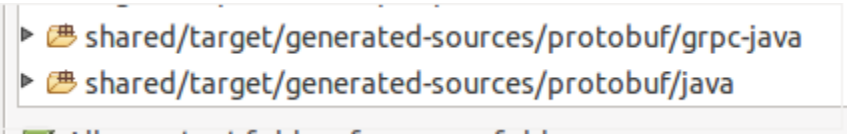
requirments : Java 8 ! (dus let op java -version en echo $JAVA_HOME)

Om zelf met dit project te experimenter, doe het volgende [1]:

1) Kopieer of clone het git project uit Github
   git clone https://github.com/robinbakkerus/OsgpLF.git
2) Compileer de .proto file(s)
   cd <path>/shared
   ./cmp.sh
   cd <path>/shared.dlms
   ./cmp.sh
3) Importeer de 5 projecten als maven project in (een nieuwe workspace) Eclipse.
   De layout zou er nu alsvolgt uit moeten zien:



Voeg in **shared** en **shared-dlms** de source folder: de **/target/generated-sources/protobuf/grpc-java** en **java** toe, zie:



Standaard werkt het systeem met de Perst object database. Daarnaast zijn er nog dao implementies gemaakt voor de Aerospike en Redis database. In de file: **common.conf**
wordt vastgelegd welke database op runtime gebruikt wordt.

3) (Optioneel als je Aerospike wilt gebruiken) Installeer de database, en start de database server:
   cd scripts

---

1 Deze procedure werkt alleen onder Linux, omdat Aeropspike onder Windows anders geinstalleerd moet worden.

./setup-aerospike.sh

./start-aerospike.sh

4) (Optioneel als je Redis wilt gebruiken)

Installeer Redis en start de server zie: **https://redis.io/documentation**

5) Vul eerst eenmalig de database met bijvoorbeeld 1 miljoen devices:

Run : `InsertDevices`.

6) Start (in willekeurige volgorde) de apps die horen bij resp platform/core/protocol-adapter-dlms:

`Platform`

`Core`

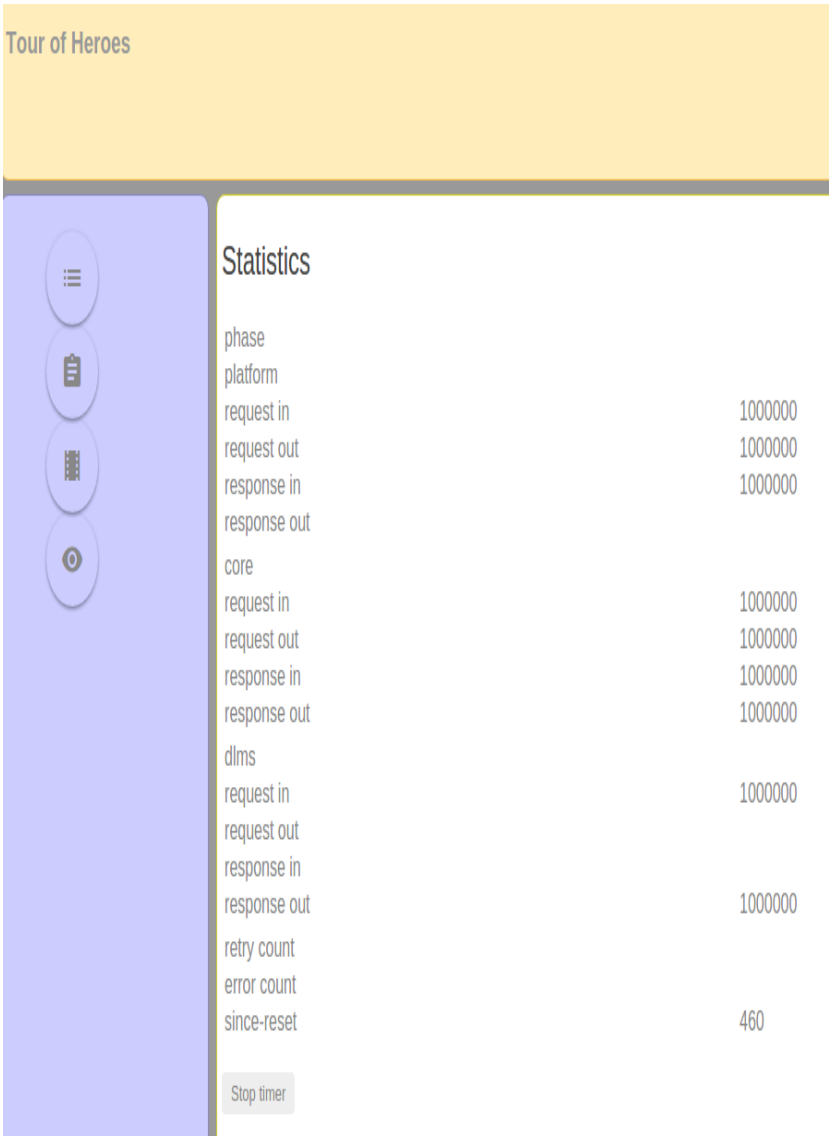
`Dlms`

(die moeten alle binnen enkele seconden starten)

7) Om request(s) te versturen en de flow door het systeem te monitoren:

run Dart applicatie : in folder he-gui : pub serve –-port 9000

submit een nieuwe job met 1 miljoen device operatiies, deze is binnen 5 minutes gedaan:



Appendix

Het aantal files dat nodig is om nieuwe operatie toe te voegen.

Als voorbeeld is de Jira story SLIM-854. Hierbij ging het om het aanmaken van een nieuwe LoadProfile operatie.
OSGP/Shared : 2 files (wsdl, xsd)
OSGP/Platform : 8 files (endpoint, value-object, converters, mappers etc)
OSGP/Protocol-A-Dlms : 6 files (converters, mappers, daadwerkelijke implementatie)
En dan moeten in het cucumber test project nog een builder worden gemaakt om deze nieuwe operatie te testen.

In OsgpLF, zouden alleen deze files moeten worden aangepast:
– dlms.proto
– In de protocol-adapter een nieuwe implementatie class, die je alleen maar als volgt hoeft te annoteren: @AnnotCommandExecutor(action=RequestType.XXXX)
Voor de test-client is de builder al gegenereerd.

Testrun van 100K job, met vertraging van 30 seconden.

Als ik een testrun draai van 100K device-operations (in mijn trage Virtualbox), waarbij een vertraging van 30 seconden wordt gesimuleerd, laat de console het volgende zien:

06:55:47.765 [main] WARN (InsertDevices.java:62) - inserted 1000000 core devices in 12417 msecs
06:56:18.819 [main] WARN (InsertDevices.java:81) - inserted 1000000 dlms devices in 31053 msecs
Insert deviceoperation took 17238 msecs
Bundling took 52660 msecs found 1000000 device operations
Send took 23303 msecs
…
since reset : 3360 finished: in 3147 seconds.

De de totale tijdsduur (3247 msecs), is dan ook practisch geheel toe te schrijven aan de tijd die het duur om met 1000 simulane actors, 30 seconden te wachten. (100K / 1000 * 30) = 3000 seconds.