High performance reactieve applicaties met Vert.x

Ervaringen uit de praktijk

Door de explosieve groei van internetgebruik zijn de performance-eisen voor webapplicaties tegenwoordig heel wat anders dan 5 of 10 jaar geleden. Neem bijvoorbeeld Facebook met ruim 750 miljoen actieve gebruikers per dag of Twitter dat zo'n 300.000 requests per seconde moet zien af te handelen. Het verwerken van zo'n belasting wordt ook wel het C10k probleem genoemd: de (inmiddels alweer verouderde) uitdaging van het tegelijkertijd afhandelen van 10.000 connecties of zelfs een veelvoud daarvan. Traditionele applicatieservers werken met een thread-gebaseerde aanpak, waarbij elke connectie exclusief door één OS thread wordt afgehandeld. Dat schaalt echter niet voldoende om tienduizenden connecties tegelijk aan te kunnen. Het reactieve applicatieplatform Vert.x pakt dat heel anders aan. In dit artikel lees je hoe en delen we onze praktijkervaringen met Vert.x.

Introductie

Vert.x is een lichtgewicht, asynchroon, event-driven, schaalbaar applicatieframework voor reactieve applicaties op de JVM. Het doel van Vert.x is om een 'eenvoudig, maar niet simplistisch' platform te bieden voor de ontwikkeling van moderne webapplicaties. Vert.x is 100% open source, valt onder de Apache 2.0 license en is één van de populairste Javaprojecten op Github van dit moment. Vert.x is *polyglot*: je kunt je applicaties schrijven in Java, Groovy, JavaScript, CoffeeScript, Ruby, Python en combinaties daarvan. Vert.x maakt gebruik van Netty voor het afhandelen van netwerk I/O en biedt support voor websockets en SockJS.
De uitvoerbare unit van Vert.x is een *verticle*: een klasse die reageert op berichten (events) en communiceert door berichten te sturen. Een verticle heeft twee kerneigenschappen: Events worden afgehandeld door een *event loop* (ook bekend als *run loop*). Communicatie tussen verticles vindt plaats via de *event bus*: een eenvoudig, maar krachtig messaging-systeem. De volgende secties gaan dieper op deze eigenschappen in.

Eventloop

Verticles worden uitgevoerd door één of meerdere eventloops in Vert.x. Een eventloop is een oneindige loop die continu controleert of er nieuwe events zijn die afgehandeld moeten worden. Voor elk event roept de eventloop een eventhandler aan, die de afhandeling van het event op zich neemt. Normaal gesproken gebruikt Vert.x één eventloop per CPU core. Door het werken met een eventloop en non-blocking I/O kan Vert.x flink wat connecties tegelijkertijd afhandelen met een minimum aantal threads. Vert.x implementeert hiermee het *multi-reactor* patroon. Uit deze aanpak volgt een eenvoudig actor-achtig concurrency-model: je kunt je applicaties schrijven alsof ze single-threaded zijn. Je hebt geen last van de traditionele valkuilen bij multi-threaded ontwikkeling. Een eenvoudige webserver in Vert.x ziet er uit als in **Listing 1**. In dit voorbeeld initialiseren we een webserver die op poort 80 luistert naar binnenkomende events. Hiervoor gebruiken we de Vert.x API die via *getVertx().createHttpServer()* een nieuwe instantie van een HttpServer aanmaakt. Voor

▸▸

het afhandelen van HTTP-requests maken we een anonymous inner class van een Handler met generic type HttpServerRequest. Deze handler zal op elk binnenkomend request reageren met een statuscode 200 en body “Hello Vert.x !”.
Tenslotte starten we de HttpServer. Elk binnenkomend HTTP-request zal vervolgens door dezelfde RequestHandler-instantie worden afgehandeld.Bij het werken met een eventloop is er één gouden regel waar je je altijd aan moet houden: *don't block the event loop!* Wanneer je blocking zaken als database-I/O in de eventloop uitvoert, staan alle connecties die door de eventloop worden bediend stil. Daar gaat je performance… De oplossing daarvoor is gelukkig erg eenvoudig: Vert.x biedt een hybride threading model waarbij niet persé alle verwerking via de eventloop hoeft plaats te vinden.
Naast verticles die op de eventloop draaien, heb je ook de mogelijkheid om zogenaamde *worker verticles* in te zetten: deze maken gebruik van een klassieke threadpool. Non-blocking verwerking zoals requestafhandeling vindt plaats op de eventloop en blocking zaken kun je via de eventbus delegeren naar workers. **Figuur 1** illustreert dit principe. Clients sturen HTTP-requests naar Vert.x, die worden afgehandeld door RequestHandlers op de eventloop. Wanneer de afhandeling blocking zaken bevat, delegeert de RequestHandler de verwerking via de eventbus naar een worker verticle.

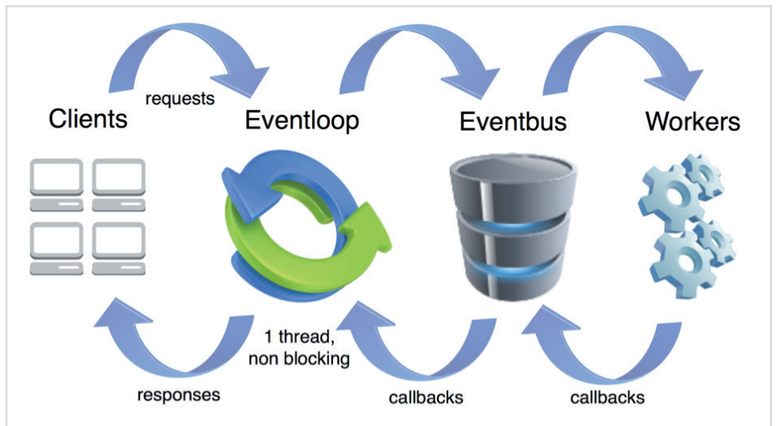
Eventbus

Vert.x bevat een ingebouwde gedistribueerde eventbus waarover verticles met elkaar kunnen communiceren. De implementatie van de eventbus leunt op Hazelcast, waarover je verderop in dit nummer meer kunt lezen. Door de clustering

```
public class VertxHttpServer extends Verticle {

  @Override
  public void start() {
    HttpServer httpServer = getVertx().createHttpServer();

    httpServer.requestHandler(new Handler<HttpServerRequest>() {
      public void handle(final HttpServerRequest request) {
        container.logger().info(“Received http event”);
        request.response().setStatusCode(200);
        request.response().end(“Hello Vert.x !”);
      }
    });

    httpServer.listen(80);
  }
}
```

Listing 1

mogelijkheden van Hazelcast kunnen verticles zowel binnen dezelfde JVM met elkaar communiceren als over meerdere JVM's heen. Dit zorgt voor *loosely coupled* componenten die je over een netwerk kunt distribueren.
Door Vert.x's polyglot mogelijkheden kun je op deze manier ook componenten in verschillende programmeertalen met elkaar laten communiceren: via JavaScript zelfs tot in de browser van de eindgebruiker!
Gebruik van de eventbus is relatief eenvoudig: je kunt een zogenaamde *message handler* registeren bij de eventbus op een bepaald adres (een string). Vervolgens kun je vanuit andere verticle berichten sturen naar dat adres. Vert.x regelt de routering en het bezorgen van berichten. Zo'n bericht is normaalgesproken een JSON-object of een string. **Listing 2** illustreert de werking van communicatie over de eventbus. Dit is een aangepaste versie van het eerdere voorbeeld. In plaats van direct een antwoord op het binnenkomende request te geven, sturen



Figuur 1: Hybride threading-model

VERT.X IS EEN LICHTGEWICHT, ASYNCHROON, POLYGLOT APPLICATIE-PLATFORM VOOR REACTIEVE APPLICATIES OP DE JVM.

we nu een bericht van het type String naar het eventbus-adres “bus.hello”. Omdat we een antwoord terug verwachten, registreren we ook een reply-handler die het antwoord aan de aanroepende partij (een browser) zal teruggeven.
Voor de verwerking van het eventbus-event maken we een nieuwe verticle, die tijdens het opstarten een message-handler registreert op het adres “bus.hello”. Wanneer de handler een bericht binnenkrijgt concateneren we ‘from eventbus!’ aan het bericht en sturen hem terug naar de aanroeper.
In dit voorbeeld handelen we een binnenkomend HTTP-request intern asynchroon af. Wanneer er ‘achter’ de eventbus iets misgaat (bijvoorbeeld een exceptie in de *handle()*-methode van de EventBusVerticle), komt geen event terug in de binnenste *handle()*-methode van de VertxHttpServer. Die stuurt dan ook geen response terug naar de client waardoor de browser daar blijft wachten op een antwoord.
Sinds versie 2.1 van Vert.x is het mogelijk om bij het verzenden van een bericht over de eventbus een time-out op te geven. Bij verstrijken van de timeout wordt de replyHandler dan alsnog (op een herkenbare manier) aangeroepen, zodat deze niet blijft wachten op een antwoord van de eventbus.
In dit voorbeeld komen twee uitdagingen van het werken met een asynchroon programmeermodel aan bod. Ten eerste de zogenaamde ‘callback hell’: voor een vrij simpel voorbeeld zijn al behoorlijk wat callbacks. Ten tweede is het omgaan met fouten: als er iets achter de eventbus misgaat, moet je erg scherp op de foutafhandeling letten om te voorkomen dat je aan de voorkant timeouts krijgt.

Schaalbaarheid

De eventbus met zowel intra-VM als inter-VM communicatie is ongelooflijk krachtig: je kunt op één machine meerdere Vert.x instanties draaien, maar ook tussen meerdere machines binnen een netwerk communiceren. Combineer dat met een loadbalancer die de load over meerdere machines verdeelt en je kunt vrijwel oneindig in de breedte schalen. Uiteraard met de kanttekening dat je je applicatie dan wel stateless moet opzetten. Wil je horizontaal kunnen schalen, dan mag je binnen een Vert.x-instantie geen state (zoals sessiegegevens) bewaren.
Vert.x bevat ingebouwde ondersteuning voor high availability van modules. Wanneer een instantie in een cluster faalt, worden de modules van die instantie automatisch op een andere instantie gedeployed. Dit gedrag is te sturen via een configuratieparameter.

```
public class VertxHttpServer extends Verticle {
  public void start() {

    HttpServer httpServer = getVertx().createHttpServer();

    httpServer.requestHandler(new Handler<HttpServerRequest>() {
      public void handle(final HttpServerRequest request) {
        container.logger().info(“Received http event”);
        getVertx().eventBus().send(“bus.hello”, “Hello”,
          new Handler<Message<String>>() {
            public void handle(Message<String> event) {
              request.response().setStatusCode(200);
              request.response().end(event.body());
            }
        });
      }
    });
    httpServer.listen(80);
  }
}



public class EventBusVerticle extends Verticle {
  public void start() {
    getVertx().eventBus().registerHandler(“bus.hello”,
      new Handler<Message<String>>() {
        public void handle(Message<String> event) {
          event.reply(event.body() + “ “ + “from eventbus!”);
        }
    });
  }
}
```

Listing 2

Modules

Vert.x bevat een krachtig module-systeem. De filosofie van de makers hierachter is dat een simpele kern met losse modules als uitbreidingen beter te onderhouden is dan een monolitische kolos. Een Vert.x module is een package (zip) met één of meerdere verticles. De enige vereiste is dat in de root van de module een bestand ‘mod.json’ staat met een aantal eigenschappen van de module (onder andere de ‘main’ verticle die bij starten van de modules wordt aangeroepen en een parameter die aangeeft of een verticle een worker is). Applicatiespecifieke configuratie van modules vindt plaats via een JSON configuratie-bestand. Er is vanuit de Vert.x community een flinke set modules beschikbaar voor bijvoorbeeld persistence in MongoDB, JDBC en session management.

Overige features

Vert.x bevat standaard ondersteuning voor unit- en integratietesten. Bij unittests kun je verticles als unit benaderen en geïsoleerd testen. Bij integratietesten kun je via de container API van Vert.x zelf verticles en modules deployen vanuit je test en bijvoorbeeld via HTTP benaderen. Wanneer je gebruik maakt van een embedded database als H2 kun je zelfs je complete ap-

plicatie in de lucht brengen en bijvoorbeeld integratietests op REST-endpoints uitvoeren. In het Github-repo in de referenties vind je hiervan een aantal voorbeelden.
Soms is het handig om data tussen meerdere verticles te kunnen delen zonder dat je daarvoor naar de database hoeft. Vert.x biedt daarvoor een 'shared data' mechanisme. Hierin kun je een aantal eenvoudige, immutable types kwijt zoals String en boolean. In de huidige Vert.x-versie werkt het delen van data alleen tussen verticles in dezelfde Vert.x-instantie. Er zijn plannen om dit in een latere versie uit te breiden naar alle Vert.x instanties in een cluster. Dan wordt het een stuk beter bruikbaar.
Wanneer je aan het ontwikkelen bent, is het vaak prettig om zo snel mogelijk het resultaat van je wijzigingen in je applicatie te kunnen zien. Vert.x helpt daarbij door ondersteuning te bieden voor het rechtstreeks starten van verticles vanuit IntelliJ en Eclipse. Het is zelfs mogelijk om (lokaal) gedeployde verticles automatisch te herladen zodra je een wijziging doorvoert.

Reactive Vert.x

Op pagina 6 van dit Java magazine heb je een introductie in reactieve applicaties en bijbehorende terminologie kunnen lezen. Vert.x vult de kenmerken van reactieve applicaties als volgt in:

- *Event-driven:* Vert.x reageert op events door het inzetten van een eventloop en eventbus;
- *Scalable:* Vert.x reageert op load door horizontaal te kunnen schalen met behulp van de clustermogelijkheden die Hazelcast biedt;
- *Resilient:* Vert.x reageert op storingen door modules op een falende instance automatisch op een andere instance te deployen;
- *Responsive:* Vert.x reageert op gebruikers door de combinatie van bovenstaande eigenschappen.

Vert.x in de praktijk

Voor een aantal projecten bij een klant gebruiken we Vert.x in combinatie met AngularJS en MongoDB. Vert.x dient dan als middenlaag met aan de voortkant REST-services voor de AngularJS frontend en aan de achterkant MongoDB voor data persistence. We gebruiken de polyglot-mogelijkheden van Vert.x niet, omdat de Java-variant ons voldoende houvast biedt. We maken ook geen gebruik van de mogelijkheid om de eventbus tot in de browser door te trekken, omdat we de bus intern willen houden en niet naar buiten willen blootleggen.
Alle test- en productieomgevingen draaien in de Amazon Cloud, waarbij we voor een aantal omgevingen meerdere applicatie-instanties achter een Elastic Loadbalancer draaien. Deployments van Vert.x modules doen we via een zelf ontwikkelde deployment module (ook weer op basis van Vert.x) die de deployment artifacts uit een afgeschermde Nexus-omgeving haalt. Het aansturen van de deployment module verloopt via een Maven plugin. Op deze manier kunnen we deployments vanuit een build pipeline in onze Jenkins CI-omgeving uitvoeren. Er zit een actieve community achter Vert.x: de makers (met name de project lead Tim Fox) reageren snel op vragen en een pull request met een fix die we nodig hadden voor de deploy-module werd binnen twee dagen upstream gemerged in de Vert.x source tree.

Conclusie

Vert.x is een lichtgewicht, asynchroon, polyglot applicatieplatform voor reactieve applicaties op de JVM. Door de eventloop, eventbus en clustering-mogelijkheden biedt Vert.x veel mogelijkheden op het gebied van schaalbaarheid en performance bij het afhandelen van vele gelijktijdige requests.
Is Vert.x het dan helemaal? Zijn er geen nadelen? Tja. Werken met callbacks en een eventbus is behoorlijk wennen. Wanneer je een architectuur met meerdere lagen hebt (view-logica, business logica en datalogica) is het traceren van het volledige pad dat een request doorloopt best een klus. Het woord 'callback hell' en de vergelijking met de syntax van jQuery.ajax() bekruipt je zeker in het begin regelmatig. Het uitvoeren van meerdere calls in een bepaalde volgorde, al dan niet parallel, is bij asynchrone verwerking ook een uitdaging. Wij gebruiken daar een eigen orkestratie-component voor, maar je kunt er ook een library als RxJava voor inzetten. Java 8 lambda's zouden hier mogelijk ook verlichting in kunnen brengen. Ons advies is om in dat geval erg terughoudend te zijn bij het gebruik van parallelle streams, omdat dat de efficiëntie van de eventloop makkelijk in de weg kan gaan zitten. Ondanks de nadelen op gebied van leesbaarheid levert het asynchrone model je ook veel op: juist deze structuur faciliteert concurrency en schaalbaarheid. Onze ervaringen met Vert.x zijn in ieder geval positief en we raden Vert.x dan ook van harte aan als platform voor het ontwikkelen van reactieve applicaties op de JVM. ■

Bert Jan Schrijver is Software craftsman bij JPoint. Hij is erg geïnteresseerd in Java, open source en continuous delivery.

Marcel Soute is Software craftsman bij JPoint.

REFERENTIES

http://vertx.io
https://github.com/msoute/javamagazine-vertx-examples
https://github.com/vert-x/vertx-examples