



Faculteit Bedrijf en Organisatie

Welke technologie is het meest geschikt om in een IoT-applicatie data van microservices te gebruiken?

Ruben Desmet

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Sonia Vandermeersch
Co-promotor:
Maarten Meersseman

Instelling: TVH

Academiejaar: 2018-2019

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Welke technologie is het meest geschikt om in een IoT-applicatie data van microservices te gebruiken?

Ruben Desmet

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Sonia Vandermeersch
Co-promotor:
Maarten Meersseman

Instelling: TVH

Academiejaar: 2018-2019

Tweede examenperiode

Woord vooraf

Tijdens dat ik dit onderzoek voerde, liep ik ook mijn stage in het bedrijf TVH. Al snel dit academiejaar kwam ik te weten dat ik hier mijn eerste werkervaring mocht opdoen. Toen er gevraagd werd voor een onderwerp te zoeken voor deze bachelorproef, leek het mij handig om een onderzoek te voeren die interessant was voor TVH. Na een brainstorm-sessie met mijn stagementor en de product-owner van mijn stage kwam ik tot dit onderwerp.

Eerst wist ik nog helemaal niets over dit onderwerp. Een ideaal moment dus om zaken bij te leren. Op mijn stage werken we ook met een event-bus. Ons team van de stage gebruikt die van Google, namelijk Google Pub/Sub. Andere teams gebruiken Kafka en ik vernam dat vroeger ook nog RabbitMq gebruikt werd. Dit leek mij ideaal om te onderzoeken welke technologie nu eigenlijk beter is.

Gelukkig waren er een tal van mensen die mij steunden tijdens deze periode. Eerst en vooral wil ik mijn ouders bedanken, die altijd klaar stonden indien ze met iets konden helpen. Verder wil ik ook mijn collega's van mijn stage bedanken voor ideeën aan te reiken. Verder zijn er nog twee personen die voor dit onderzoek een dikke pluim verdienen. Als eerste wil ik mevrouw Vandermeersch heel erg bedanken om mij steeds vooruithelpende feedback te geven en altijd bereid te zijn om alles eens na te lezen. Als laatste wil ik ook een hele grote dankjewel zeggen aan Maarten Meersseman die mij inhoudelijk bijstond voor de technologieën van Kafka en RabbitMq, en altijd klaar stond om extra uitleg te geven. Door deze personen heb ik ook meer inzicht gekregen in het onderzochte thema.

Samenvatting

Dit onderzoek vergelijkt drie verschillende technologieën die data verzenden en ontvangen via event-bussen. Bij TVH zijn ze al reeds in contact gekomen met de drie technologieën die hier onderzocht worden. Het is belangrijk dat dit onderzocht werd omdat men dan eens kon te weten komen als de meeste teams effectief met de beste technologie bezig zijn. Momenteel gebruiken de meeste teams Kafka, maar bijvoorbeeld in het team waar de onderzoeker (Ruben Desmet) zijn stage gelopen heeft, werd er Google Pub/Sub gebruikt. Het komt dus niet ongelegen om te weten te komen welke nu eigenlijk het beste gebruikt moet worden.

Dit zou kunnen tot positieve gevolgen leiden. Indien ze een meer efficiëntere technologie vinden, kunnen ze sneller of met minder resources hun data verzenden. Hierdoor kunnen er mogelijk andere voordelen naar boven komen, zoals bijvoorbeeld meer data verwerken binnen een tijdspanne. Indien de conclusie van dit onderzoek zou zijn dat ze al reeds met de beste technologie bezig zijn, dan is dit ook een positieve uitkomst. Hierdoor weten de teams dat ze nog geen tijd verloren zijn.

Voor dit onderzoek zijn Google Pub/Sub, Kafka en RabbitMq met elkaar vergeleken. Eerst werd de snelheid gemeten tussen het verzenden naar de event-bus en het ontvangen ervan in een andere applicatie. Dit werd voor drie verschillende hoeveelheden van objecten gedaan. Als eerste werden er 10 000 objecten genomen, daarna 100 000 en als laatste 1 000 000. Hierna werd ook nog het memory gebruik vergeleken. Dit werd maar voor één hoeveelheid gedaan, namelijk 1 000 objecten, omdat dit veel meer tijd in beslag nam. De resultaten van de memory en de snelheid werden niet op het zelfde moment berekend.

Dit document beschrijft hoe dit onderzoek in zijn werk is gegaan. De resultaten uit dit onderzoek waren bij grotere hoeveelheden moeilijk te vergelijken met elkaar omdat

er verschillende hoeveelheden objecten effectief aankwamen. Maar uit het onderzoek kunnen we toch concluderen dat Kafka het beste is om mee te werken voor TVH. Als verder onderzoek zou je dit onderzoek meerdere keren kunnen uitvoeren. Een andere mogelijkheid is om de applicaties niet lokaal maar in de cloud te laten draaien, op die manier ben je niet afhankelijk van een pc of laptop.

Inhoudsopgave

1	Inleiding	15
1.1	Probleemstelling	16
1.2	Onderzoeksvraag	16
1.3	Onderzoeksdoelstelling	16
1.4	Opzet van deze bachelorproef	16
2	Stand van zaken	17
2.1	Microservices	17
2.2	Kafka	19
2.3	RabbitMq	21
2.4	Kafka vs RabbitMq	22
2.5	Google Pub/Sub	23

2.6	IoT-applicatie	24
2.6.1	Informatie verzamelen en verzenden	26
2.6.2	Informatie verzamelen en reageren	26
2.6.3	Combinatie	26
2.7	TVH	27
3	Methodologie	29
3.1	Producer project	30
3.1.1	Creëren van de data	31
3.2	Consumer project	33
3.3	Onderzoek	34
4	Proof of Concept	37
4.1	Producer project	37
4.1.1	Kafka	37
4.1.2	RabbitMq	41
4.1.3	Google Pub/Sub	43
4.2	Consumer project	45
4.2.1	Kafka	47
4.2.2	RabbitMq	49
4.2.3	Google Pub/Sub	51
5	Conclusie	55
5.1	Snelheid	55
5.1.1	10 000 objecten	55
5.1.2	100 000 objecten	57

5.1.3	1 000 000 objecten	60
5.1.4	Conclusie snelheid	62
5.2	Memory	62
5.2.1	1 000 objecten	62
5.2.2	Conclusie memory	62
6	Besluit	65
A	Onderzoeksvoorstel	67
A.1	Introductie	67
A.2	Literatuurstudie	67
A.3	Methodologie	68
A.4	Verwachte resultaten	68
A.5	Verwachte conclusies	69
	Bibliografie	71

Lijst van figuren

2.1	Monolitische architectuur	18
2.2	Wisselwerking tussen partities en consumer-groepen, (Sookocheff, 2015) 20	
2.3	Voorbeeld van een Kafka cluster, (Johansson, 2016)	21
2.4	Simpele voorstelling van de werking van een offset, (Sookocheff, 2015) 21	
2.5	Werkings drie verschillende exchanges bij <i>RabbitMq</i> , (Johansson, 2015) 22	
2.6	Populariteit <i>Kafka</i> en <i>RabbitMq</i> in het afgelopen jaar, (Trends, 2019) 23	
2.7	Populariteit <i>Kafka</i> en <i>RabbitMq</i> in de afgelopen vijf jaar, (Trends, 2019) 23	
2.8	De flow die een message aflegt bij <i>Google Pub/Sub</i> , (Google, 2019) 24	
2.9	De gecreëerde waarde van IoT, (i-scoop, 2019)	25
4.1	Configuratie van de topic	38
4.2	Configuratie van de topic voor de gebruikte subscription	44
5.1	Vergelijking bij 10 000 objecten	57

5.2	Vergelijking bij 100 000 objecten	59
5.3	Vergelijking bij 1 000 000 objecten	61
5.4	Vergelijking memory	63

Lijst van tabellen

5.1	Verschil tussen ontvangen en verzenden (in ms) - Google Pub/Sub	55
5.2	Verschil tussen ontvangen en verzenden (in ms) - Kafka	56
5.3	Verschil tussen ontvangen en verzenden (in ms) - RabbitMq	56
5.4	Verschil tussen ontvangen en verzenden (in ms) - Google Pub/Sub	58
5.5	Verschil tussen ontvangen en verzenden (in ms) - Kafka	58
5.6	Verschil tussen ontvangen en verzenden (in ms) - RabbitMq	58
5.7	Verschil tussen ontvangen en verzenden (in ms) - Google Pub/Sub	60
5.8	Verschil tussen ontvangen en verzenden (in ms) - Kafka	60
5.9	Verschil tussen ontvangen en verzenden (in ms) - RabbitMq	61

1. Inleiding

Dit onderzoek wordt uitgevoerd op vraag van het bedrijf TVH. Dit is een familiebedrijf dat oorspronkelijk opgericht is door Paul Thermote en Paul Vanhalst in 1969. Ze startten in een schuur waar ze kapotte landbouwmachines en forklifts repareerden. Later verkochten ze ook deze tweedehands machines. Vervolgens startten ze snel met het verhuren van deze werktuigen. Enkele jaren later was het begin van de verkoop van vervangonderdelen voor machines. Nadien werden er nog een paar bedrijven overgenomen waardoor het bedrijf uitgroeide tot het grote bedrijf dat we vandaag de dag kennen.

Maar wat is het nut van dit onderzoek voor TVH? Zoals je overal ziet, is internet bijna nergens meer weg te denken. Ook voor dit bedrijf kan internet enkele voordelen bieden. TVH verzamelt en verzendt voornamelijk informatie. Dit wordt later verder uitgelegd in sectie 2.6.

Al deze informatie die verstuurd wordt van de machines, wordt samengebracht en verwerkt. Een voorbeeld van informatie is een machine die stuurt dat zijn sleutel aan staat. De IoT-applicatie die deze informatie daadwerkelijk zal verwerken heeft een grote verantwoordelijkheid. Het is namelijk belangrijk dat er geen data verloren gaat bij het verzenden en dat de data heel snel verwerkt kan worden. Er zijn enkele software mogelijkheden om met deze informatie om te gaan. Dit onderzoek zal nagaan welke technologie het beste is om te gebruiken voor TVH met zijn specifieke data.

Doordat dit onderzoek zal aantonen welke technologie de beste zal zijn, ontstaan er enkele voordelen voor het bedrijf TVH. Data kan op de meest efficiënte wijze verzonden worden waardoor alles ook sneller verwerkt kan worden. Hierdoor kan het bedrijf meer winst maken omdat het sneller conclusies kan trekken uit de verkregen data.

1.1 Probleemstelling

Als technologie werd er vroeger binnen het bedrijf *RabbitMQ* gebruikt. Ondertussen zijn alle teams overgeschakeld naar een andere technologie namelijk *Kafka*. Oorspronkelijk was het dus de bedoeling van dit onderzoek om deze twee met elkaar te vergelijken en te concluderen dat TVH al dan niet de juiste beslissing had gemaakt om over te schakelen naar *Kafka*. Ondertussen ben ik ook al gestart met mijn stage binnen TVH waarbij ik deel uitmaak van een student-squad. Binnen ons team gebruiken wij *Kafka* noch *RabbitMQ*. Wij gebruiken een derde mogelijkheid, namelijk *Google Pub/Sub*, alle andere teams hebben nog steeds *Kafka* als technologie. Er zijn dus verschillende technologieën die gebruikt worden binnen dit bedrijf, waarbij niemand weet welke nu eigenlijk het beste is om te gebruiken.

1.2 Onderzoeksvraag

De bedoeling van dit onderzoek is deze drie technologieën vergelijken en concluderen welke ervan de beste blijkt te zijn. Zo komt het bedrijf te weten of het handig zou zijn om in elk team te blijven werken met *Kafka* of om terug over te schakelen naar een andere technologie.

1.3 Onderzoeksdoelstelling

Dit onderzoek zal geslaagd zijn indien we voor TVH met hun specifieke data kunnen concluderen welke technologie het beste kan gebruikt worden voor alle teams. Op deze manier werkt elk team met dezelfde technologie en kan er ook meer kennis gedeeld worden.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie. Enkele begrippen worden grondig uitgelegd alsook de werking van de verschillende technologieën.

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 4 wordt het onderzoek gevoerd op een bepaald voorbeeld. Hier zullen er resultaten uit komen die besproken zullen worden in de conclusie.

In Hoofdstuk 5, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen.

2. Stand van zaken

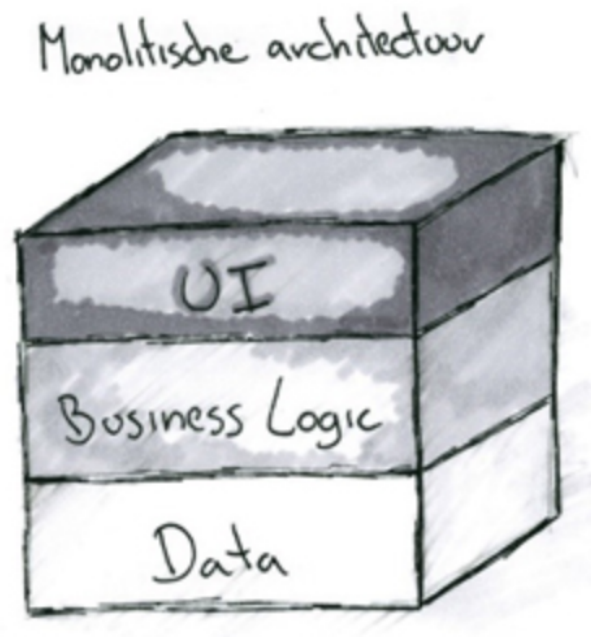
In dit hoofdstuk gaan we Kafka en RabbitMq bespreken. Want natuurlijk zijn de begrippen zoals *Kafka* en *RabbitMq* niet voor iedereen even duidelijk. Ook de term microservices zal bij sommigen de wenkbrauwen wel eens doen fronsen. In dit hoofdstuk is het de bedoeling uit te leggen wat al deze begrippen betekenen. Na dit hoofdstuk zul je dankzij deze uitleg in staat zijn om om alles goed te begrijpen.

2.1 Microservices

Microservices is een software architectuur. De applicatie bestaat uit meerdere kleinere microservices die samen één groot geheel vormen. Deze kleinere microservices zijn onafhankelijk van elkaar en hebben elk hun eigen proces. Deze software architectuur is vrij recent en is de laatste jaren een echte hype aan het worden.

Een belangrijke vraag is: waarom zijn microservices ontstaan? Server-side applicaties gebruiken meestal object-georiënteerde programmeertalen. Deze programmeertalen hebben abstracties om de complexiteit van hun programma's te behandelen in modules. Dit wordt ook wel eens 'the monolith' genoemd. Dit is één groot geheel dat meestal uit drie lagen bestaat. De presentatielaag, de businesslaag en de data laag. Deze lagen kunnen niet apart van elkaar gebruikt worden omdat ze verschillende resources met elkaar delen. Dit zorgt ervoor dat bij iedere wijziging in de applicatie, alle lagen opnieuw gereleased worden in een nieuwere versie. Je kan natuurlijk al raden dat dit in een grote applicatie niet de beste oplossing is.

Daarom is er een andere architectuur die een heel andere aanpak heeft, namelijk microservices. Dit bestaat uit verschillende kleinere componenten die indien nodig onafhankelijk



Figuur 2.1: Monolitische architectuur

van elkaar uitgevoerd kunnen worden. Bij deze aanpak is het ook belangrijk dat je uw services klein houdt. Hierdoor kunnen je services gemakkelijk hergebruikt, begrepen en opnieuw gebuild worden. Iedere microservice heeft dus maar 1 verantwoordelijkheid.

Omdat microservices op verschillende machines moeten kunnen draaien, bijvoorbeeld op verschillende besturingssystemen, is het beter om ze in te pakken samen met hun dependencies in een container. *Docker* is een voorbeeld van een technologie die deze containers aanbiedt. Door microservices in containers te plaatsen, zorg je ervoor dat de uitvoering van deze services onafhankelijk gebeurt van andere applicaties op dezelfde machine. Containers zijn dus onafhankelijk van een besturingssysteem, hierdoor kunnen microservices op verschillende locaties gedraaid worden. Een van de voordelen hiervan is dat ze door containers gemakkelijker met de cloud kunnen werken. Een ander voordeel is dat iedere microservice zijn eigen verantwoordelijkheid heeft. Omdat ze maar één verantwoordelijkheid hebben, kunnen ze beter garanderen dat dit ook goed uitgevoerd wordt. Een nadeel van microservices is dat het steeds over het netwerk loopt. Er kan hierdoor vertraging zijn, maar er kan ook iets fout lopen over het netwerk waardoor er mogelijk informatie verloren gaat. Het is nooit de bedoeling dat informatie verloren gaat natuurlijk. Je kan hiervoor maatregelen nemen zodat de kans dat dit gebeurt minimaal is. Een ander nadeel is dat als je veel microservices hebt en die moeten allemaal met elkaar communiceren, dan mag er niets mis zijn met het netwerk om elkaar te vinden. Dit probleem kan opgelost worden door service discovery. Dit is het vinden van de netwerk locatie van een ander microservice. Indien een service een dynamisch ip-adres geeft, is het niet de bedoeling dat twee services niet meer met elkaar kunnen communiceren indien een ip-adres verandert. Indien je meer wilt weten over de werking van service discovery dan verwijs ik u graag door naar het artikel van Xu (2019).

De vier grootste voordelen van microservices zijn:

- Schaalbaarheid
- Beperken complexiteit
- Verkorten time-to-market
- Autonomie van ontwikkelteams: je kan met verschillende teams werken, je kan de beste technologie kiezen, afhankelijk van het probleem dat je wilt oplossen.

Kafka, *RabbitMq* en *Google Pub/Sub* zijn technologieën die gebruikt kunnen worden om informatie te verwerken in microservices. Hieronder verklaren we hoe deze verschillende technologieën werken en waar de verschillen zitten.

(Guidi, Lanese, Mazzara & Montesi, 2017), (Velthoven, 2016) en (Xu, 2019)

2.2 Kafka

Wanneer je als bedrijf veel berichten binnen krijgt op een korte tijdsspanne, dan heb je natuurlijk een goed functionerende technologie nodig die al deze berichten kan verwerken. *Kafka* is een voorbeeld van zo een technologie. Het is dus een berichtensysteem waarbij schaalbaarheid en redundantie een grote troef zijn. Bepaalde kernwoorden zijn belangrijk om de architectuur van *Kafka* te begrijpen. Deze kernwoorden zijn: topics, producers, consumers en brokers.

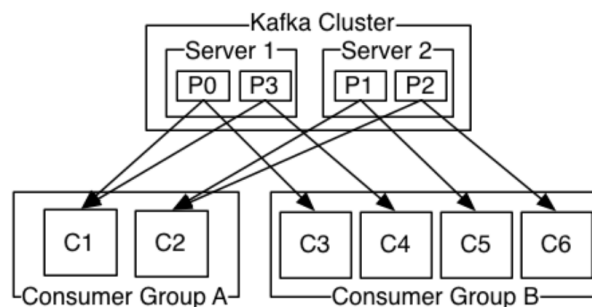
Als producer kun je een bericht verzenden naar uw gewenste topic. Topics zijn, zoals de naam al doet vermoeden, verschillende onderwerpen. Alle berichten zijn gegroepeerd in een van deze topics. Een topic kan verschillende nodes hebben, maar het kan ook zijn dat er maar één node is. Als er verschillende nodes zijn dan kan je de inkomende data repliceren. Een node kan opgedeeld zijn in verschillende partities. Als een bericht binnenkomt, dan wordt deze opgeslagen in een partitie van je topic die op zijn beurt weer op een verschillende node kan zitten.

Een topic kan je dus verdelen in verschillende partities. Dit wil zeggen dat je uw data kunt verdelen in ongeveer gelijke groepen. Iedere partitie kan dan staan voor een specifieke groep data binnen een topic zodat je niet alle berichten altijd moet overlopen. Dit principe van verschillende partities wordt ook gebruikt bij traditionele databanken. Het is dus geen garantie dat het effectief gelijke groepen zijn. De bestemming van een bericht hangt af van de message-key. Als je bijvoorbeeld woorden wilt groeperen en je verdeelt je groepen volgens de eerste letter van ieder woord. Dan staan alle woorden die beginnen met een a samen, ieder woord dat begint met een b, ... Als je dan toevallig veel woorden binnen krijgt die beginnen met een c, dan kan het gebeuren dat je groepen niet gelijk verdeeld zijn.

Een bericht heeft een bepaalde retention. Dit is de tijd dat het bericht maximum beschikbaar is. Als bijvoorbeeld de retention op zeven dagen staat, en je binnen deze zeven dagen het bericht niet gelezen hebt, dan wordt het bericht verwijderd. Dan is het niet meer mogelijk om de inhoud van het bericht nog ooit op te vragen. Het formaat van een bericht kan verschillend zijn. Het type kan een gewone tekst zijn, kan van een Json-formaat zijn, of

kan iets helemaal anders zijn. Dit waren allemaal voorbeelden van leesbare data, maar het is ook mogelijk dat het niet leesbaar is. Het formaat kan ook binair zijn, Avro is hier een voorbeeld van. Het is mogelijk om zowel naar een specifieke topic een bericht te verzenden als te ontvangen.

Dit brengt ons naadloos bij het volgende begrip: consumers. Een consumer kan zelf bepalen van welke topic hij berichten wilt ontvangen. Consumers binnen een consumer-groep worden automatisch enkele partities toegewezen. Door topics en consumers op deze manier op te delen, zorg je ervoor dat het mogelijk is dat meerdere consumers kunnen lezen van meerdere partities in een topic. Dit heeft als positief gevolg dat je meer berichten kunt verwerken binnen een bepaalde tijd. Het is al even vermeld, maar wat is nu eigenlijk een consumer-groep? Dit is een verzameling van verschillende consumers. Iedere consumer op zich leest van één specifieke partitie waardoor je het aantal verwerkte berichten binnen één tijdseenheid kunt verhogen. Alle consumers binnen één groep lezen samen alle berichten die op een topic staan. Het opdelen van je topic in verschillende partities zorgt er dus niet voor dat je een deel van je data verliest. Mochten er meer consumers zijn dan dat er partities zijn, dan zitten er sommigen zonder werk. Omgekeerd, als er meer partities zijn dan consumers, dan krijgen consumers van verschillende partities berichten binnen. De hoeveelheid consumers binnen een groep is afhankelijk van de hoeveelheid partities die er zijn.

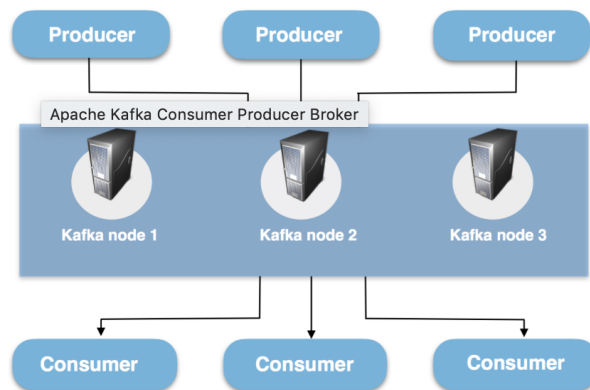


Figuur 2.2: Wisselwerking tussen partities en consumer-groepen, (Sookocheff, 2015)

Figuur 2.2 is een voorbeeld hoe een topic in verschillende servers kan opgedeeld worden en hoe consumers in consumer-groepen kunnen onderverdeeld worden. Je ziet dat de Kafka cluster in twee servers onderverdeeld is. Iedere server heeft twee partities. Er zijn 2 consumer-groepen, groep A bestaat uit twee consumers, groep B bestaat uit 4 consumers. Iedere partitie kan dus naar verschillende consumer-groepen berichten versturen, maar kan niet binnen dezelfde consumer-groep naar een andere consumer iets verzenden.

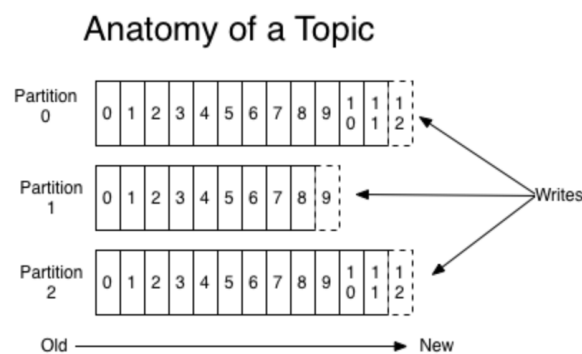
Het laatste woord dat nog moet verduidelijkt worden is een broker. *Kafka* draait op een cluster. Iedere cluster bestaat uit één of meerdere nodes(servers). Per Kafka cluster kunnen er verschillende producers en consumers zijn, zoals te zien is op figuur 2.3.

Ieder bericht binnen een partitie heeft een offset. Dit is een identifier, waardoor het mogelijk is om de berichten te ordenen. Normaal gezien als je als consumer een subscriptie maakt op een topic, dan krijg je vanaf dit moment alle nieuwe berichten die binnen komen op de partitie waarop je een subscriptie hebt. Door een offset is het mogelijk om iets oudere



Figuur 2.3: Voorbeeld van een Kafka cluster, (Johansson, 2016)

berichten die op een partitie staan dan het moment dat je een subscriptie aangemaakt hebt, ook op te vragen. Op figuur 2.4 zie je een simpele voorstelling van een producer die berichten op een partitie van een topic plaatst. De cijfertjes stellen de offset voor van een bericht.



Figuur 2.4: Simpele voorstelling van de werking van een offset, (Sookocheff, 2015)

(Sookocheff, 2015) en (Johansson, 2016)

2.3 RabbitMq

Een alternatief voor *Kafka* is *RabbitMq*. Dit is software waar men berichten kan plaatsen en ophalen op één of meerdere 'queues'. Ook hier heb je verschillende mogelijkheden van wat het formaat is van de berichten. Het kan zowel een gewone tekst zijn, als een Json-file.

Er zijn hier ook producers die data op de queue zetten, alsook consumers die een binding kunnen maken op een queue. Wanneer een bericht van een queue wordt gelezen, dan wordt deze verwijderd van de queue. Het is dus niet mogelijk om later opnieuw hetzelfde bericht op te gaan vragen.

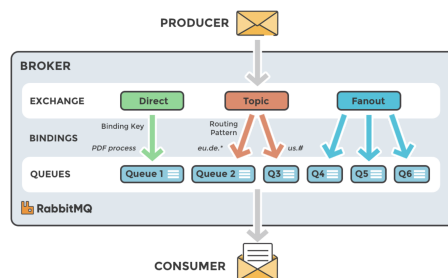
RabbitMq kan een voordeel zijn om te gebruiken in verschillende situaties. Als je bijvoor-

beeld als gebruiker de data die je opslaat in een queue wilt distribueren naar verschillende consumers, dan is ook *RabbitMQ* een ideale technologie om dit te doen. Het is dus mogelijk dat verschillende consumers de berichten lezen. Een ander voordeel is dat het mogelijk is om je berichten te verdelen over verschillende consumers. Op deze manier wordt dan de hoeveelheid mooi gebalanceerd verdeeld over de verschillende consumers.

Een *RabbitMQ* broker bevat exchanges en queues. De exchange is verantwoordelijk voor het verzenden van de berichten naar de verschillende queues. Indien een bericht in de broker toekomt, moet deze dus eerst langs de exchange voordat hij in een queue terecht kan. De link tussen de queue en de exchange wordt ook wel eens een binding genoemd.

Er zijn vier verschillende soorten van exchanges. Dit zijn: direct, fanout, topic en headers exchanges. Drie van deze worden uitgebeeld in figuur 2.5

- De direct exchanges sturen berichten naar een queue op basis van een routing key. Deze key is hetzelfde als de binding key van de queue waarnaar het naartoe moet verzenden.
- Fanout exchanges sturen berichten naar alle queues die verbonden zijn met de exchange
- Bij Topic exchanges worden wildcard matches gebruikt. De match wordt gedaan tussen de routing key en de routing pattern die in de binding gespecificeerd wordt.
- Header exchanges gebruiken de attributen uit de header om te bepalen welke queue de bestemming is van een bericht.



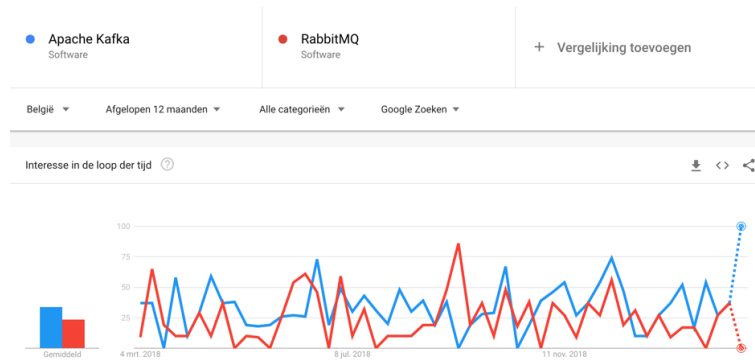
Figuur 2.5: Werking drie verschillende exchanges bij *RabbitMQ*, (Johansson, 2015)

(Johansson, 2015)

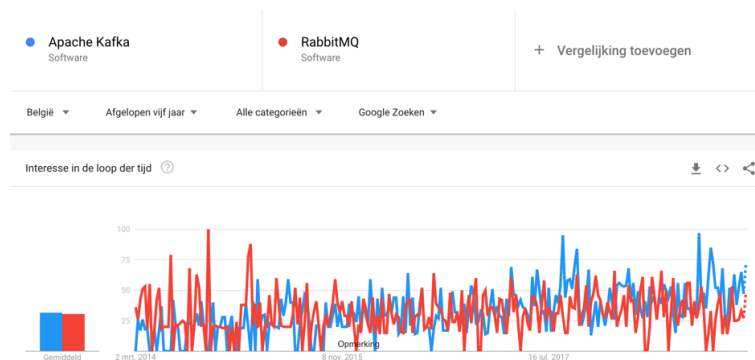
2.4 Kafka vs RabbitMQ

Als we kijken naar Google Trends om te bepalen welke van deze twee technologieën nu het populairste is, dan krijgen we in figuur 2.6 een grafiek te zien van het afgelopen jaar.

In deze grafiek is duidelijk te zien dat *Kafka* iets populairder is dan *RabbitMQ*. Er zit tussen juli en november wel een piek in waarbij *RabbitMQ* veel populairder is, maar over het algemeen gezien kunnen we concluderen dat in het afgelopen jaar *Kafka* toch iet wat populairder is.



Figuur 2.6: Populariteit *Kafka* en *RabbitMq* in het afgelopen jaar, (Trends, 2019)



Figuur 2.7: Populariteit *Kafka* en *RabbitMq* in de afgelopen vijf jaar, (Trends, 2019)

Als je figuur 2.7 bekijkt, dan zie je een algemene trend. Namelijk dat vijf jaar geleden *RabbitMq* veel populairder was, maar dat deze trend systematisch omgedraaid is. Uit deze figuur valt ook af te leiden dat in het algemeen de interesse naar technologieën voor microservices lichtjes toegenomen is in de laatste vijf jaar.

(Trends, 2019)

2.5 Google Pub/Sub

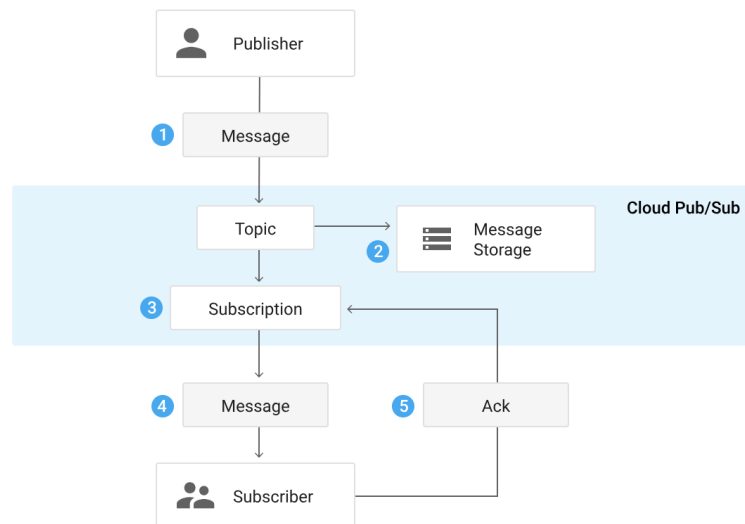
Tijdens het academiejaar dat dit onderzoek is uitgevoerd, zijn de gebruikte technologieën die TVH gebruikt ook wat veranderd. Er is momenteel een nieuwe technologie bij gekomen, namelijk *Google Pub/Sub*. Zoals de naam doet vermoeden is Google eigenaar van deze technologie.

Ook hier zijn het ongeveer dezelfde kernwoorden die belangrijk zijn:

- Topic
- Subscription
- Message

- Publisher
- Subscriber

De termen topic en message zijn hetzelfde als bij de andere technologieën. Subscription is de link tussen de subscriber (deze wordt straks uitgelegd), en de topic. Dit is dus een soort van abonnement die een subscriber aangaat met een topic. Een subscriber is iemand of iets die de verschillende messages van een specifieke topic leest. Een publisher is dan degene die nieuwe messages op een topic plaatst.



Figuur 2.8: De flow die een message aflegt bij *Google Pub/Sub*, (Google, 2019)

Op figuur 2.8 is te zien hoe een message tot bij een subscriber geraakt, iemand of iets die de message leest. Eerste stap is de publisher die een message plaatst (published) naar een topic. Een message bevat een payload, dit is de inhoud van de message. Ook kunnen er eventueel attributen toegevoegd worden die iets meer vertellen over de inhoud van de payload. Vervolgens wordt in de tweede stap de message opgeslagen in de Message Storage totdat een subscriber de message leest en acknowledged. Dit wil zeggen dat de subscriber een bevestiging stuurt naar de subscription om te melden dat hij de message goed ontvangen heeft. Hierna zet de Pub/Sub deze message klaar in al zijn subscriptions. Deze message wordt dan gelezen als bijvoorbeeld de subscriber deze message binnen trekt. In de vierde stap zien we dat alle berichten die nog niet gelezen zijn door de subscriber, bij de subscriber binnen komen. Als een message goed is aangekomen, dan wordt er een acknowledge gestuurd naar de subscription. Dan wordt deze message ook verwijderd en kan deze niet meer opnieuw gelezen worden. Dit is de vijfde stap.

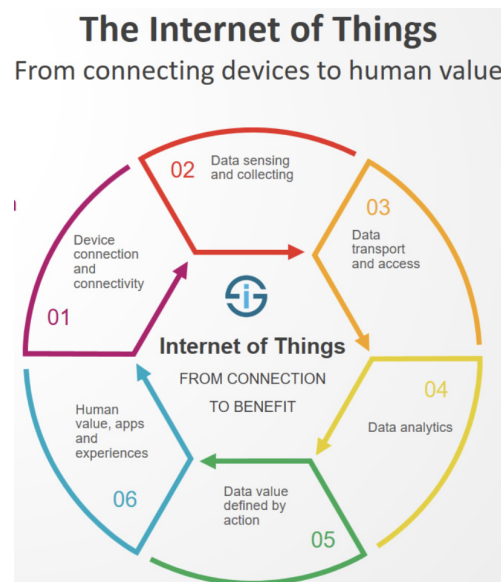
(Google, 2019)

2.6 IoT-applicatie

Microservices architectuur wordt meestal gebruikt in een IoT-applicatie. Bij microservices is het meestal zo dat er veel data is. En bij IoT wordt er veel data verzonden. Voluit staat

IoT voor Internet of Things. Onder deze noemer verstaan we machines en alledaagse objecten die verbonden staan met het internet en eventueel ook met elkaar.

Iedereen weet wat de voordelen zijn van internet op toestellen zoals een gsm, tablet, ... Laten we naar het voorbeeld van een gsm bekijken. Als je vergelijkt wat een gsm allemaal kon toen er nog geen internet mogelijk was en nu, dan zie je dat het eigenlijk een ander toestel geworden is met veel meer functionaliteiten. Dit is ook zo bij alledaagse objecten. Met het toevoegen van internet op deze objecten verhoog je de functionaliteit ervan. We zullen dit even illustreren met een voorbeeldje. Neem nu een lamp in je woonkamer. Dankzij IoT ben je niet meer verplicht om manueel een schakelaar die fysiek aanwezig is in je woonkamer om te draaien. Door de lamp te verbinden met het internet kan je die ook vanop afstand aan of uit schakelen via een app op je smartphone of iets dergelijks. Dit geeft extra voordelen aan het product. Als we ons eigen voorbeeldje nog eens bekijken, kan dit handig zijn als je vergeten het licht te doven bent indien je bijvoorbeeld naar je werk vertrokken bent. Dan neem je gewoon je smartphone en doof je het licht vanop afstand. Zonder IoT was dit allemaal niet mogelijk geweest.



Figuur 2.9: De gecreëerde waarde van IoT, (i-scoop, 2019)

Op figuur 2.9 kan je de flow zien hoe Internet of Things extra waarde oplevert voor een bedrijf. In de eerste stap zorg je ervoor dat je object waar je informatie uit wilt halen verbonden is met het internet. Vervolgens worden er metingen en waarnemingen gedaan zodat er iets is dat het toestel kan verzenden. Nadien wordt de data natuurlijk effectief verzonden. Als vierde stap kan je eventueel analyses doen op je verkregen informatie zodat je nog extra zaken kunt concluderen. In de volgende stap moet je natuurlijk waarde hechten aan de informatie die binnen komt. Want natuurlijk weet het apparaat dat informatie verstuurt niet waarvoor een bepaald cijfer staat. De bedoeling in deze stap is dus bepaalde waarde voor het bedrijf toevoegen aan de verkregen getallen of waarden. Als laatste stap kan je conclusies trekken en eventueel zelf als bedrijf mogelijke acties ondernemen. Deze 6 stappen illustreren hoe je door het toevoegen van connectiviteit extra waarde voor een bedrijf kunt realiseren.

Het is nu niet zo dat alleen alles wat je van op afstand aan of uit kunt schakelen onder de noemer IoT valt. Er zijn voornamelijk drie categorieën waarin we Internet of Things objecten onderscheiden.

- Zaken die informatie verzamelen en verzenden.
- Zaken die informatie verzamelen en daaropvolgend een actie uitvoeren.
- Zaken die de twee bovenstaande combineren.

Het bedrijf TVH gebruikt voornamelijk de eerste categorie. Dit wordt in 2.7 uitgelegd.

2.6.1 Informatie verzamelen en verzenden

Bij deze categorie komen er sensoren bij kijken. Sensoren kunnen van verschillende type zijn: bewegings-, temperatuur- of lichtsensoren bijvoorbeeld. Alle mogelijke types van sensoren kunnen hiervoor gebruikt worden. Deze sensoren zijn verbonden met het internet en nemen bepaalde waarnemingen aan van de omgeving. Deze waarnemingen zijn afhankelijk van het type sensor. Bijvoorbeeld bij een temperatuursensor zal deze om een bepaalde tijd de temperatuur van zijn locatie verzenden. De informatie die sensoren verzenden kunnen van grote waarde zijn voor bedrijven. Hierdoor kunnen ze bepaalde beslissingen nemen die mogelijk tot meer winst of andere voordelen leiden.

2.6.2 Informatie verzamelen en reageren

De voorbeelden van deze categorie liggen dicht bij de hand dan je zou denken. Er zijn tal van voorwerpen die dagelijks gebruikt worden die informatie verzamelen en erop reageren. Enkele voorbeelden zijn: een printer die een document ontvangt en uitprint of een autosleutel die je auto open maakt of sluit wanneer je op het juiste knopje duwt.

Deze voorbeelden zorgen niet meteen voor een business-waarde om bijvoorbeeld de winst van een bedrijf te verhogen. Dit wilt niet zeggen dat deze zaken daarom minder interessant zijn.

2.6.3 Combinatie

Door deze twee categorieën te combineren, krijgen we nieuwe mogelijkheden. Door deze combinatie kun je een voortdurende wisselwerking creëren tussen acties die iemand neemt en de reacties die het object maakt. Laten we een fictief voorbeeldje verzinnen. Stel je wilt een cilindervormig voorwerp vullen met water. Het voorwerp kan 100 liter water bevatten. De bedoeling is dat het water altijd tussen de 80 en 90 liter moet blijven. Dan kan je een sensor installeren die detecteert of het water onder de 80 liter zit. Indien de sensor dit detecteert, moet hij automatisch het voorwerp bijvullen. De machine stopt met het object te vullen wanneer de sensor detecteert dat hij aan 89 liter zit. Hierop voert hij een actie uit om geen water meer bij te vullen. Op deze manier is er een continue werking van het systeem door het verzamelen van informatie en een actie uit te voeren aan de hand

van de ontvangen informatie. Voor veel bedrijven kan dit handig zijn om processen te optimaliseren, want er moeten in principe geen mensen meer aan de pas komen. Vroeger zou in dit voorbeeldje een mens voortdurend handmatig controleren of het water niet onder de 80 liter zat, daarna zelf het voorwerp bijvullen. Hierbij was er nog een extra risico dat de persoon te laat ontdekt dat er meer dan 90 liter water zou zijn. Dit zou erge gevolgen kunnen hebben voor een bedrijf.

(McClelland, 2019) en (i-scoop, 2019)

2.7 TVH

Aan ieder werktuig hangt een tracking-device. Met dit toestel is het mogelijk om de locatie van al hun toestellen die ze verhuren te lokaliseren. Maar deze apparaten verzenden niet alleen hun locatie, ze verzenden ook veel andere nuttige informatie. Ze houden bijvoorbeeld bij hoeveel werkuren een machine al bezig is. Of ze sturen door als de motor van de machine aanligt, de machine aan het bewegen is, ... Kort samengevat: er komt heel veel informatie binnen uit deze toestellen en daar zit heel veel handige informatie in voor het bedrijf. Zo kunnen ze bijvoorbeeld na een tijdje te weten komen wat de tijd is tot dat een onderdeel van een machine vervangen moet worden. Of ze kunnen bedrijven die machines huren controleren dat ze effectief alleen de werktuigen gebruiken tijdens de werkweek, en niet tijdens het weekend.

Laten we nog even figuur 2.9 erbij nemen. We hebben al algemeen uitgelegd wat deze afbeelding voorstelt. Nu gaan we dit even toepassen op het bedrijf TVH. In de eerste stap worden de tracking-devices aan de werktuigen gekoppeld. Nadien verzamelen deze devices informatie over de machines. Deze informatie wordt dan in stap drie verzonden. Een voorbeeld van analyse op de informatie die binnenkomt binnen het bedrijf is controleren als het tracking-device dat informatie verzendt wel gekoppeld is aan een werktuig. In stap vijf koppel je de cijfers aan specifieke waarden. Bijvoorbeeld input 2 geeft 0 of 1 terug. Dit kan betekenen dat een motor van een machine al dan niet aanligt. In de laatste stap kan het bedrijf effectief acties ondernemen uit de conclusies die getrokken kunnen worden van de verkregen informatie.

Dankzij de tracking-devices zijn alle machines gekoppeld aan het internet. Voor een IoT-applicatie die al deze informatie verzamelt heb je natuurlijk een goede technologie nodig. *Kafka*, *RabbitMQ* en *Google Pub/Sub* zijn mogelijke technologieën en zijn al reeds uitgelegd. Dit onderzoek moet uitwijzen welke technologie het beste is voor TVH om deze informatie te verzamelen.

3. Methodologie

Om te testen welke van de technologieën die in dit onderzoek aan bod komen nu eigenlijk het beste is, zou je met veel zaken rekening moeten houden. Wat het beste is hangt vooral af van welke soort data je gebruikt. Ook de hoeveelheid transformaties die je data ondergaan voor dat ze uitgelezen worden of verzonden speelt een grote rol. Natuurlijk is het moeilijk om binnen de tijdspanne die er is voor dit onderzoek al deze factoren te gaan onderzoeken. Daarom zal dit onderzoek zich toespitsen op een bepaald scenario en daar conclusies uit trekken. Voor het scenario beslissen we op voorhand voor welk formaat van de data dat we dit onderzoek zullen voeren, dit zal een Json zijn. Ook zullen we het onderzoek drie keer uitvoeren, voor verschillende groottes van data. Een keer met 10 000, 100 000 en 1 000 000.

Voor dit onderzoek zijn er twee grote applicaties nodig. Een producer en een consumer applicatie. De producer zorgt ervoor dat er per technologie een implementatie is waardoor een technologie data kan verzenden. In dit onderzoek zal dit voor Kafka, RabbitMq en Google Pub/Sub gebeuren. Elke keer deze applicatie opgestart wordt, zal er meegegeven worden voor welke technologie er data moet verzonden worden.

Voor de tweede applicatie is er ook een implementatie nodig per technologie die ervoor zal zorgen dat er geluisterd wordt naar de topic/queue indien er nieuwe messages zijn. Deze verschillende consumers zullen tegelijkertijd luisteren naar hun eigen topic/queue. Deze applicatie moet dus niet apart opgestart worden.

Nu gaan we de implementatie overlopen van objecten van klassen die onafhankelijk zijn van een technologie.

3.1 Producer project

Natuurlijk moet er iets gelijkaardigs zijn om te kunnen vergelijken. Het data object dat we zullen verzenden en ontvangen bij zowel *Kafka*, *RabbitMq* en *Google Pub/Sub* zal hetzelfde zijn. Op deze manier is het toekomstige resultaat niet afhankelijk van het type data. Dit onderzoek zal zich dus toespitsen op data met het type Json. Deze klasse (*Data.class*) hergebruiken we in elke technologie. Deze klasse stelt het type voor van het bericht dat we verzenden.

```
1 @Getter
2 @AllArgsConstructor
3 @NoArgsConstructor
4 @ToString
5 public class Data {
6     private int id;
7     private String name;
8     private String description;
9     private Date sendedDate;
10 }
```

Deze klasse bevat vier attributen. Dit is om een Json object na te bootsen. Het is vanzelfsprekend dat deze klasse niet zo maar automatisch kan omgezet worden naar een Json. De manier waarop dit gedaan wordt is opnieuw per technologie verschillend. De uitleg zal gegeven worden in de subsectie van de technologie zelf. De attributen die hier gebruikt worden zijn: een id van het type int, dit is om elk data object van elkaar te kunnen onderscheiden tijdens het verzenden. Als tweede zie je name van het type String. Dit heeft als bedoeling opnieuw om verschillende objecten van elkaar te kunnen onderscheiden en om een realistisch attribuut te geven aan het Data-object. Deze doelstelling is dezelfde voor het attribuut description van het type String. Als laatste hebben we de sendedDate van het type Date. Deze zal de datum en het tijdstip bijhouden wanneer het object verzonden is. Dit zal later nog eens aangehaald worden wanneer we deze objecten genereren.

Als u boven de klasse kijkt dan ziet u nog vier annotaties staan. Deze kunnen we gebruiken door de library van Lombok. Dit werd mogelijk gemaakt door deze dependency toe te voegen aan onze pom.xml want het gebruikte project voor dit onderzoek maakt gebruik van Maven om libraries toe te voegen.

```
1 <dependency>
2     <groupId>org.projectlombok</groupId>
3     <artifactId>lombok</artifactId>
4     <optional>true</optional>
5 </dependency>
```

De library Lombok geeft op voorhand al enkele handige implementaties. De gebruikte annotaties zullen we uitleggen, want natuurlijk bestaan er meer dan enkel deze die dit onderzoek gebruikt. Voor de andere mogelijkheden verwijs ik u graag door naar de site van Project Lombok waar u alle mogelijkheden op een rijtje ziet: <https://projectlombok.org/features/all>.

De eerste annotatie is `@Getter`, deze zorgt ervoor zoals de naam al doet vermoeden, dat er voor ieder attribuut een getter aanwezig is. Op deze manier staat uw code niet overvol met getters van al uw attributen. In deze klasse zijn er niet zo veel attributen dus zou het aantal getter nog meevallen. Maar u kan zich wel inbeelden dat in een groter project in een klasse met veel attributen en andere methodes dat dit een handige annotatie is. Je ziet dus niet dat de getters aanwezig zijn maar die zijn er wel effectief. Laten we de tweede en de derde samen bekijken: `@AllArgsConstructor` en `@NoArgsConstructor`. Deze zorgen ervoor dat er een constructor gegenereerd wordt met respectievelijk al de attributen en zonder de attributen. Dit zijn vaak voorkomende constructors en het is handig om ook hiervoor een annotatie te hebben. De laatste is de `@ToString`. Deze werd gebruikt tijdens het testen van het project. Dit zorgt voor een String-representatie van het object. Dit was handig om lokaal te testen. Het data object werd dan uitgeprint tijdens het verzenden om te controleren of het object wel de juiste waardes bevatte.

3.1.1 Creëren van de data

De implementatie van de verschillende technologieën komt in het volgende hoofdstuk aan bod. Het enige werk dat in dit hoofdstuk voor de Producer applicatie overblijft, is het creëren van de data en het verzenden ervan. Er zijn nog drie cruciale elementen die besproken moeten worden en die zijn: de `ProducerApplication`-, de `RandomDataProvider`- en de `RandomDataPublisher`.

ProducerApplication

Dit is de eerste klasse die gebruikt wordt om de applicatie te starten. In de methode `runner` geven we via de parameters alle klassen mee die data publishen. Dit is dus een object van de klasse van `GooglePubSubPublisher`, `KafkaPublisher` en `RabbitMqPublisher`. De implementatie van deze klassen worden ook in volgend hoofdstuk besproken. Ook moeten we de klasse `RandomDataPublisher` meegeven. In deze methode roep je de methode `doStuff` op van de `RandomDataPublisher` klasse. Wat deze methode exact doet bespreken we later. Maar je moet meegeven welke publisher je wilt gebruiken. Deze lijn code is het enige dat je moet aanpassen om data te verzenden voor één van de drie technologieën.

```
1 @SpringBootApplication
2 @Slf4j
3 public class ProducerApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(ProducerApplication.class, args)
7         ;
8     }
9
10    @Bean
11    public CommandLineRunner runner(final RandomDataPublisher
12        randomDataPublisher,
13        final GooglePubSubPublisher googlePubSubPublisher,
14        final KafkaPublisher kafkaPublisher,
```

```

13         final RabbitMqPublisher rabbitMqPublisher) {
14             return args -> {
15                 randomDataPublisher.doStuff(
16                     googlePubSubPublisher);
17             //    OF:         randomDataPublisher.doStuff(
18             kafkaPublisher);
19             //    OF;         randomDataPublisher.doStuff(
20             rabbitMqPublisher);
18                 };
19             }
20     }

```

RandomDataProvider

Deze klasse is een hulpklasse die de lijst van Data objecten genereert. Het aantal objecten wordt meegegeven als parameter en wordt ingesteld in de RandomDataPublisher. De naam van het object en de beschrijving is telkens de naam van het attribuut, respectievelijk name en description, plus het id eraan geplakt. Hier wordt ook een nieuwe datum gecreëerd die dient als sendedDate.

```

1 @Component
2 public class RandomDataProvider {
3     public List<Data> create(int numberOfEntries) {
4         final List<Data> dataList = new ArrayList<>();
5         for (int i = 0; i < numberOfEntries; i++) {
6             dataList.add(new Data(i, "name" + i, "description"
7                 + i, new Date()));
8         }
9         return dataList;
10    }
11 }

```

RandomDataPublisher

Deze klasse heeft de functionaliteit om de methode create uit de klasse RandomDataProvider op te roepen om de Data objecten te genereren en door te geven aan de juiste publisher. De constructor heeft opnieuw de publisher klassen als parameter die gecreëerd worden door Spring Boot. Het heeft ook nog een extra parameter namelijk een instantie van de RandomDataProvider om later de create methode te kunnen gebruiken.

We verdiepen ons even in de methode doStuff. Hierin wordt aan de randomDataProvider gevraagd om een aantal Data objecten te maken. In dit codevoorbeeld is dit 10000 objecten. Deze slaat hij op een lijst van het type Data. In de methode wordt er gecontroleerd van welke instantie de publisher is die je mee krijgt. Indien de publisher van Kafka of RabbitMq is dan wordt deze lijst verzonden naar de Kafka topic of de RabbitMq exchange. Is het

een instantie van de publisher van Google Pub/Sub dan wordt deze lijst overlopen en per object verstuurd. De publish methode van GooglePubSubPublisher wenst maar één object en geen lijst van objecten.

```
1 @Component
2 public class RandomDataPublisher {
3
4     private final RandomDataProvider randomDataProvider;
5     private final KafkaPublisher kafkaPublisher;
6     private final GooglePubSubPublisher googlePubSubPublisher;
7     private final RabbitMqPublisher rabbitMqPublisher;
8
9     public RandomDataPublisher(RandomDataProvider
10        randomDataProvider,
11        KafkaPublisher kafkaPublisher,
12        GooglePubSubPublisher googlePubSubPublisher,
13        RabbitMqPublisher rabbitMqPublisher) {
14         this.randomDataProvider = randomDataProvider;
15         this.kafkaPublisher = kafkaPublisher;
16         this.googlePubSubPublisher = googlePubSubPublisher
17         ;
18         this.rabbitMqPublisher = rabbitMqPublisher;
19     }
20
21     public void doStuff(Object object) {
22         final List<Data> dataList = randomDataProvider.
23             create(10_000);
24         if(object instanceof GooglePubSubPublisher) {
25             for (Data data : dataList) {
26                 googlePubSubPublisher.publishData(data);
27                 System.out.printf("publishing data: %s",
28                     data);
29             }
30         } else if (object instanceof KafkaPublisher){
31             kafkaPublisher.publish(dataList);
32         } else {
33             rabbitMqPublisher.publish(dataList);
34         }
35     }
36 }
```

3.2 Consumer project

De bedoeling van deze applicatie is dat ze wanneer er iets op een topic of queue geplaatst wordt, ze deze informatie weer omzet en opslaat in een lokale databank. Bij het opslaan van de data wordt er meer opgeslagen dan dat er informatie binnenkomt om deze te kunnen vergelijken, maar dit wordt straks duidelijker. Wanneer deze applicatie opgestart is, luistert

deze voortdurend naar de topics, of naar de queue, of er messages op geplaatst worden. Indien er een message op een topic of de queue toegekomen is, dan leest hij deze meteen uit. Deze applicatie luistert dus tegelijkertijd naar zowel de topic van Kafka en Google Pub/Sub als naar de queue van RabbitMq. Om dit niet met elkaar te laten beïnvloeden wordt tijdens dit onderzoek enkel data verstuurd van een bepaalde technologie, bijvoorbeeld Kafka, indien de consumerapplicatie niets meer leest van een andere technologie. Deze aanpak zorgt er dus voor dat data van verschillende technologieën niet door elkaar gelezen worden. We hebben in deze applicatie hetzelfde domein object van de klasse Data, want we willen het object dat verstuurd werd op dezelfde manier terugkrijgen. Door later de juiste deserializers mee te geven, moeten we ons geen zorgen maken en wordt het inkomende object automatisch gemapt naar hetzelfde domeinobject.

```
1 @Getter
2 @AllArgsConstructor
3 @NoArgsConstructor
4 @ToString
5 public class Data {
6     private int id;
7     private String name;
8     private String description;
9     private Date sendedDate;
10 }
```

Naast deze Data klasse hebben we ook per technologie een entity klasse en repository. De entity klasse zorgt ervoor dat per technologie een tabel gecreëerd wordt in de databank. De repository zorgt ervoor dat een object van deze klasse effectief opgeslagen kan worden. De namen van deze entity klassen zijn: DataGPS, DataKafka en DataRMQ. Voor de repository zijn de namen van de interfaces: DataGPSRepository, DataKafkaRepository en DataRMQRepository. De implementatie van deze klassen is telkens dezelfde, maar het is nodig om deze apart te houden verschillende tabellen te hebben in de databank per technologie. In het volgende hoofdstuk zullen we één entity klasse en repository uitleggen want voor de andere is het analoog.

3.3 Onderzoek

Zoals al eerder vermeld worden de resultaten opgeslagen in een databank. De databank draait lokaal in een Docker container. Dit is de eenvoudige configuratie die nodig is om te databank op te stellen:

```
1 spring:
2     datasource:
3         url: "jdbc:postgresql://localhost:5432/postgres"
4         username: postgres
5         password: postgres
6     jpa:
7         hibernate:
8             ddl-auto: create
```

```

9         jdbc:
10         lob:
11         non-contextual-creation: true

```

De URL van de databank wordt meegegeven met daarachter de naam van de databank. Daaronder geef je de gebruikersnaam en wachtwoord mee. Het is niet erg dat het wachtwoord bloot gegeven wordt, want dit is toch een lokale databank. Verder zie je ook dat indien de applicatie opnieuw opstart, de database opnieuw gecreëerd wordt.

Verder werd dit onderzoek drie keer uitgevoerd. Een keer met 10000, 100000 en 1000000 Data objecten die verzonden werden. Zo kunnen we ook vergelijken hoe sterk het tijdsverschil wordt. De bedoeling was om de methodes die de objecten ontvangen zo klein mogelijk te houden. Want hoe groter deze methodes worden, hoe langer het duurt per object om deze objecten te ontvangen en dit zou een verkeerd beeld kunnen geven.

De resultaten werden per technologie opgeslagen in een CSV-file om daarna berekeningen op te kunnen doen. Dit werd gedaan met een Python script.

Eerst wordt de CSV uitgelezen via de pandas library. Hierna wordt er wat aan data exploratie gedaan om te controleren of de waardes die uitgelezen werden wel effectief mogelijk zijn.

```

1 import pandas as pd
2 result = pd.read_csv('/Users/rubendesmet/Desktop/gps10_000.csv', sep=',')
3 print(result.head())
4 print(result.columns)
5 print(result.index)
6 print(result.count())
7 print(result.shape)

```

Vervolgens worden de kolommen die niet van toepassing zijn verwijderd. Enkel de kolom timeDiff zou moeten overblijven die het verschil van de ontvangen en verzonden tijd voorstelt. Daarna kijken we hoeveel rijen er in de tabel zitten, worden eventuele NaN verwijderd, en kijken we nog eens hoeveel rijen er in de tabel zitten. Als laatste controlemiddel printen we nog eens de eerste vijf rijen af en zo kunnen we ook zien of we kolommen vergeten verwijderen zijn.

```

8 print(result.count())
9 result = result.dropna()
10 print(result.count())
11 print(result.head())

```

Dan rest ons nog één ding, namelijk het berekenen van het gemiddelde van deze kolom, samen met het minimum en maximum.

```

12 print(result['timeDiff'].mean())
13 print(result['timeDiff'].max())

```

```
14 | print(result['timeDiff'].min())
```

Al deze stappen worden dus negen keer uitgevoerd. Drie keer per technologie en drie keer per hoeveelheid verzonden data.

Tweede deel van het onderzoek

Nadat dit allemaal onderzocht was, werd er door beslist door het bedrijf TVH dat we ook nog de gebruikte memory zouden onderzoeken. Hiervoor moest er niets aangepast worden aan de producer applicatie, maar wel aan de consumer applicatie.

```
15 | // Get the Java runtime
16 | Runtime runtime = Runtime.getRuntime();
17 | // Run the garbage collector
18 | runtime.gc();
19 | // Calculate the used memory
20 | long memory = runtime.totalMemory() - runtime.freeMemory();
21 |
22 | memory = bytesToMegabytes(memory);
```

Per receiver werd deze code toegevoegd in de methode die de messages ontvangt. Deze code achterhaalt de gebruikte memory. De hulpmethode `bytesToMegabytes(memory)` zet de memory om naar megabytes. Dit resultaat werd dan ook meegegeven bij het aanmaken van een object van een entity en zo mee opgeslagen. Dit wil zeggen dat er in `DataGPS`, `DataKafka` en `DataRMQ` een attribuut memory bijkwam. Bij de resultaten van dit gedeelte van het onderzoek werd enkel rekening gehouden met de memory. Kijken naar het verschil tussen het verzenden en ontvangen in tijd was niet meer nodig. Het ophalen van het gebruik van memory neemt namelijk zo veel tijd in beslag dat het tijdsverschil niet meer relevant was om te bekijken.

Door dit nieuwe gedeelte van het onderzoek kwamen er 3 CSV bestanden bij. Eén bestand per technologie die de memory berekent. Er werd gekozen om telkens maar 1000 objecten te verzenden. Er kwamen dus ook drie nieuwe Python-scripts bij. Ook één per technologie die het gemiddelde, maximum en minimum van de memory berekende per technologie. Het opbouwen van deze scripts is gelijkaardig met het berekenen van het gemiddelde tijdsverschil. Maar in plaats van het tijdsverschil werd enkel de kolom met de memory bijgehouden.

4. Proof of Concept

De algemene opzet van de twee nodige applicaties om dit onderzoek te kunnen voeren, hebben we al even gezien. Nu tonen we de implementaties die specifiek nodig zijn voor dit onderzoek. Dit zijn de implementaties van de technologieën die in dit onderzoek gekozen werden om te vergelijken: Kafka, RabbitMq en Google Pub/Sub. Eerst bekijken we de producer applicatie.

4.1 Producer project

4.1.1 Kafka

In figuur 4.1 ziet u welke configuratie de topic heeft. Er wordt gebruik gemaakt van zes partities en de data wordt vier weken bijgehouden. Met andere woorden: als de data op de topic komt, en deze niet binnen de vier weken door een consumer gelezen wordt, dan wordt de data verwijderd en kan deze data niet meer opnieuw uitgelezen worden.

KafkaConfig

Zo ziet de implementatie van de KafkaConfig.class eruit.

```
1 @Configuration
2 public class KafkaConfig {
3
4     @Bean
5     @Primary
6     public KafkaProperties kafkaProperties()
```

MANAGEMENT > TOPICS >
New topic

Availability

Setting ⓘ
Balanced availability/durability

Cleanup policy

Cleanup policy ⓘ
Delete

Time to retain data ⓘ
4 weeks

Max size on disk in GB ⓘ
1 GB

Message size

Maximum message size in MB ⓘ
1 MB

Save & create

Back

TOPIC SUMMARY

name	test-bachelor
topic	proof-rube
partitions	6
replication factor	3
cluster	-
min.insync.replicas	2
cleanup.policy	delete
retention.ms	2419200000
retention.bytes	1000000000
max.message.bytes	1000000
delete.retention.ms	86400000

Figuur 4.1: Configuratie van de topic

```

7      @Value("${kafka_key}") final String kafkaKey,
8      @Value("${kafka_secret}") final String kafkaSecret,
9      @Value("${kafka_bootstrap}") final String
        kafkaBootstrap
10     ) {
11         KafkaProperties kafkaProperties = new
            KafkaProperties();
12
13         KafkaProperties.Producer producer =
            kafkaProperties.getProducer();
14
15         producer.getProperties().put(ProducerConfig.
            KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.
            common.serialization.StringSerializer");
16         producer.getProperties().put(ProducerConfig.
            VALUE_SERIALIZER_CLASS_CONFIG, "org.
            springframework.kafka.support.serializer.
            JsonSerializer");
17
18         producer.getProperties().put(ProducerConfig.
            BOOTSTRAP_SERVERS_CONFIG, kafkaBootstrap);
19         producer.getProperties().put(ProducerConfig.
            RETRIES_CONFIG, "0");
20         producer.getProperties().put(ProducerConfig.
            ACKS_CONFIG, "all");
21         producer.getProperties().put(ProducerConfig.
            RETRY_BACKOFF_MS_CONFIG, "1000");
22         producer.getProperties().put(ProducerConfig.
            REQUEST_TIMEOUT_MS_CONFIG, "30000");
23         producer.getProperties().put(ProducerConfig.
            LINGER_MS_CONFIG, "200");
24         kafkaProperties.getProperties().put(SslConfigs.
            SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG, "
            https");
25         kafkaProperties.getProperties().put(SaslConfigs.
            SASL_MECHANISM, "PLAIN");

```



```
26         kafkaProperties.getProperties().put("request.  
           timeout.ms", "20000");  
27         kafkaProperties.getProperties().put("retry.backoff  
           .ms", "500");  
28         kafkaProperties.getProperties().put(SaslConfigs.  
           SASL_JAAS_CONFIG, "org.apache.kafka.common.  
           security.plain.PlainLoginModule required  
           username=\" \"  
29         + kafkaKey + "\" password=\" \" + kafkaSecret + "\";  
           \");  
30         kafkaProperties.getProperties().put("security.  
           protocol", "SASL_SSL");  
31  
32         kafkaProperties.getProperties().put(JsonSerializer  
           .ADD_TYPE_INFO_HEADERS, "false");  
33  
34         return kafkaProperties;  
35     }  
36 }
```

In deze klasse ziet u dat het instellen van de nodige properties in de `kafkaProperties` methode van deze klasse gebeurt. Deze zouden ook in de `application.yml` file kunnen ingesteld worden. In dit project hebben we besloten om dit in een config klasse te doen. Tijdens het opzetten van het project kon de applicatie de properties niet uitlezen vanuit de `application.yml` file. Om geen tijd te verliezen hebben we besloten om deze properties in een config klasse in te stellen. Dit verandert niets aan de functionaliteit of aan de resultaten van dit onderzoek. Maar als onderzoeker was het mogelijk om sneller aan de slag te gaan met het belangrijkere werk.

Boven de klasse ziet u de annotatie `@Configuration`. Dit is een annotatie van het Spring framework. Deze zorgt ervoor dat je aangeeft dat deze klasse een configuratie instelt. In de klasse ziet u boven de methode `kafkaProperties` twee andere annotaties staan. `@Bean` zorgt ervoor dat er een bean aangemaakt wordt voor deze methode waardoor het Spring framework voor deze methode eerder een instantie aanmaakt. De applicatie kan daardoor bij het opstarten deze properties instellen. Daaronder zie je een andere annotatie die hiermee te maken heeft: `@Primary`. Deze zorgt ervoor dat het Spring framework meerdere beans heeft van `KafkaProperties`, hij deze voor al de andere neemt. Op deze manier zijn we zeker dat deze properties gebruikt worden en geen andere indien er meerdere beans zouden zijn.

U ziet dat er tamelijk wat properties geconfigureerd zijn. Er zullen slechts enkele properties besproken worden. Uiteraard zijn dit de belangrijkste voor dit onderzoek. In dit hoofdstuk werd al eens vermeld dat het object van de klasse `Data` niet zomaar omgezet wordt naar een `Json`. Hiervoor is er een serializer nodig die dit doet. Op lijn 16 van de `KafkaConfig` klasse is er te zien dat de gekozen serializer ingesteld wordt. In dit geval wordt de `JsonSerializer` gebruikt uit de package `springframework.kafka.support.serializer`. Dit is dus een specifieke `JsonSerializer` voor Kafka. De key wordt dan weer geserializeerd naar een `String` door de

StringSerializer uit de package `org.apache.kafka.common.serialization`. Dit is te zien op de regel er boven. Wat als het niet lukt voor de producer om de data te verzenden? In ons geval mag hij dit niet opnieuw proberen. Dit zie je op lijn 19. Indien je meer zekerheid wilt dat je een data object zeker verzonden wordt, dan kan je deze property in plaats van 0 op bijvoorbeeld 3 zetten. Je mag zeker ook niet vergeten om de username en password mee te geven van waar je Kafka topic zich bevindt. Op lijn 28 en 29 zie je dat dit meegegeven wordt. De username en wachtwoord worden niet in plaintext in de code geplaatst. Het is niet de bedoeling dat iedereen weet wat het wachtwoord en username is. Deze worden ingesteld in uw `application.yml` en worden meegegeven via de parameters van de methode door het Spring framework. Door de annotatie `@Value` en de naam van de key van uw waarden in de `application.yml`, weet Spring perfect waar hij het wachtwoord en username moet gaan halen.

KafkaPublisher

De `KafkaPublisher.class` is de effectieve producer van Data objecten. Veel moet er niet ingesteld worden om deze producer te laten werken. Wat je zeker moet hebben is een `KafkaTemplate` en de naam van de topic waarnaar je verzendt. Laat ons even deze klasse stap voor stap bekijken.

Boven de klasse staat de annotatie `@Component`. Dit is opnieuw een annotatie van het Spring framework. Deze zorgt ervoor dat je via constructor- of setterinjectie in andere klassen telkens dezelfde instantie van de klasse `KafkaPublisher` kunt gebruiken. Op die manier moet je niet telkens een nieuwe instantie maken van de klasse. Er zijn twee attributen: de `kafkaTemplate` van het type `KafkaTemplate` en een `String` `topicName`; De `KafkaTemplate` is een template die je gebruikt om je message naar een Kafka topic te verzenden.

In de constructor worden `kafkaTemplate` en `topicName` meegegeven als parameters. Spring Boot zorgt ervoor dat de `kafkaTemplate` automatisch gecreëerd wordt. De `topicName` willen we niet zo maar in onze code schrijven, dus injecteren we die vanuit de `application.yml`. Hiervoor wordt opnieuw `@Value` gebruikt.

Als laatste bevat deze klasse ook nog de `publish` methode. Deze verzendt een message effectief naar de topic. In onze implementatie krijgt deze methode een lijst van Data objecten mee. In dit onderzoek is het niet van toepassing om één enkel Data object te verzenden, dus om praktische redenen wordt er een lijst meegegeven. In deze methode wordt deze lijst overlopen en telkens worden dezelfde operaties uitgevoerd. Er wordt een object aangemaakt van de klasse `Message`, zoals de naam al doet vermoeden wordt dit onze message. In de payload van deze message stoppen we ons Data object. In de header geven we twee zaken mee. Als message key geven we het id mee, omdat in deze implementatie het id toch telkens uniek zal zijn. Als tweede moeten we nog de naam van onze topic meegeven zodat de `kafkaTemplate` weet naar waar hij moet versturen. Het enige dat deze methode dan nog moet doen is de `kafkaTemplate` gebruiken om de message te verzenden. Het omzetten van de message in een json formaat wordt automatisch gedaan omdat we in de properties een serializer opgegeven hebben. Ook de message key wordt voor dezelfde

reden automatisch geserialiseerd.

```
1 @Component
2 public class KafkaPublisher {
3
4     private KafkaTemplate kafkaTemplate;
5     private String topicName;
6
7     public KafkaPublisher(KafkaTemplate kafkaTemplate,
8         @Value("${kafka.topic.name}") String topicName) {
9         this.kafkaTemplate = kafkaTemplate;
10        this.topicName = topicName;
11    }
12
13    public void publish(List<Data> dataList) {
14        for (Data data : dataList) {
15            Message<Data> message = MessageBuilder.withPayload
16                (data)
17                .setHeader(KafkaHeaders.
18                    MESSAGE_KEY, String.valueOf(
19                        data.getId()))
20                .setHeader(KafkaHeaders.TOPIC,
21                    topicName)
22                .build();
23
24            kafkaTemplate.send(message);
25        }
26    }
27 }
```

4.1.2 RabbitMq

RabbitMqConfig

In dit onderzoek zijn alle configuraties van RabbitMq gedaan in application.yml.

```
1 spring:
2     rabbitmq:
3         host: ${rabbitmq_hostadress}
4         port: 16379
5         username: ${rabbitmq_username}
6         password: ${rabbitmq_password}
7         virtual-host: "rabbitmq"
8         ssl:
9             enabled: true
10            algorithm: "TLSv1.2"
11        listener:
12            simple:
13                default-requeue-rejected: false
```

```

14         direct:
15         default-requeue-rejected: false

```

Veel configuraties zijn er niet. De belangrijkste zijn host, port, username en password. De host is het adres waar uw RabbitMq exchange op draait met de juiste poort die opgegeven wordt in port. Username en password worden net als het host-adres bijgehouden in een variabele maar deze staan natuurlijk niet op het codevoorbeeld. Hiernaast wordt ook nog de naam van de exchange bijgehouden die we later in de implementatie nodig zullen hebben. Uiteraard zit deze property ook niet in het codevoorbeeld.

RabbitMqPublisher

Over het algemeen gezien kun je deze klasse goed vergelijken met de KafkaPublisher klasse. Het grootste verschil zit in de publish methode. De annotatie @Component boven de klasse heeft hetzelfde doel als bij de KafkaPublisher. Er is hier één attribuut meer en dat is de ObjectMapper. Deze wordt gebruikt om later in de private hulpmethode onze data om te zetten naar een String. Uiteraard is de KafkaTemplate hier vervangen door een RabbitTemplate. Deze hebben dezelfde functionaliteit maar horen bij een verschillende technologie. Het is dus ook een template die je gebruikt om de data te verzenden naar de exchange. Hier heb je niet de naam van de topic nodig zoals bij de KafkaPublisher maar de naam van de exchange.

In de constructor worden de rabbitTemplate en de objectMapper opnieuw automatisch gecreëerd door Spring Boot. De naam van de exchange wordt terug via de application.yml file geïnjecteerd. In de publish methode overlopen we opnieuw een lijst van objecten van Data. Bij iedere elementje uit de lijst voeren we dezelfde operaties uit. We gebruiken de hulpmethode 'createJsonString' om ons object om te zetten naar een String via de methode 'writeValueAsString(String value)' van de objectMapper. Het resultaat van deze String is in een Json formaat. Verder gebruiken we de rabbitTemplate om ons bericht te verzenden naar de exchange die we meegeven samen met ons bericht.

```

1  @Component
2  public class RabbitMqPublisher {
3      private RabbitTemplate rabbitTemplate;
4      private ObjectMapper objectMapper;
5      private String exchangeName;
6
7      public RabbitMqPublisher(final RabbitTemplate
8          rabbitTemplate,
9          final ObjectMapper objectMapper,
10         @Value("${rabbitmq.topic.exchange.name}") final
11         String exchangeName) {
12         this.rabbitTemplate = rabbitTemplate;
13         this.objectMapper = objectMapper;
14         this.exchangeName = exchangeName;
15     }
16
17     public void publish(final List<Data> dataList) {

```

```
16         for (Data data : dataList) {
17             String jsonData = createJsonString(data);
18
19             rabbitTemplate.convertAndSend(exchangeName, "",
20                                         jsonData);
21         }
22     }
```

4.1.3 Google Pub/Sub

Op figuur 4.2 kunt u zien hoe de configuratie van de topic is voor de gebruikte subscription in dit onderzoek. We zien dat bij het lezen van een message die op de Pub/Sub staat er een tijdslimiet van 60 seconden staat. Dit wil zeggen dat indien de message binnen de 60 seconden niet gelezen kan worden wanneer de subscriber messages aan het lezen is, dat de actie NACK uitgevoerd wordt op deze message. Dit wil zeggen dat de subscriber laat weten aan de topic dat hij deze niet kunnen ontvangen heeft. Dit wil zeggen dat de message nog altijd niet verwijderd is van de topic en dat deze nog altijd gelezen kan worden door de subscriber. Bij Google Pub/Sub kun je deze tijdslimiet instellen tussen 60 en 600 seconden. Onder de "Acknowledgement" Deadline ziet u "Message retention duration" staan. Dit is hoe lang een message op de topic blijft staan. In dit geval staat deze waarde op het maximum: zeven dagen. Met andere woorden: indien een message binnen de zeven dagen niet gelezen wordt, dan zal deze verwijderd worden en is het niet meer mogelijk om deze message terug te vinden.

GooglePubSubConfig

Voor Google Pub/Sub is de configuratie iets afwijkend in vergelijking met Kafka en RabbitMq. Hier gebruiken we zowel een configuration klasse als de application.yml file. De configuratie klasse wordt aangeduid met de annotatie van Spring: @Configuration. Ook de application.yml file wordt gebruikt voor drie properties. Laat ons eerst de application.yml file overlopen. Deze wordt gebruikt om de naam van de topic in op te slaan, zodat deze niet voorkomt in onze code. Dan zijn er nog twee zaken die hier geconfigureerd worden: spring.cloud.gcp.project.id en spring.cloud.gcp.credentials.location. Het eerste is het project id van het cloud platform waar uw Pub/Sub zich bevindt. Het tweede is het path naar de credentials file om toegang te krijgen tot het cloud platform. De rest van het configuratiewerk wordt in de GooglePubSubConfig klasse gedaan. De eerste methode createBachelorproefChannelOutput geeft een MessageChannel terug. Deze heeft een annotatie @Bean die de naam 'bachelorproef-channel-output' krijgt. Dit is letterlijk het 'kanaal' dat we gaan gebruiken om onze messages te verzenden. De volgende methode geeft een MessageHandler terug en meer bepaald in dit geval een PubSubMessageHandler. Deze heeft een pubsubTemplate die je kan vergelijken met de kafkaTemplate of rabbitTemplate. Ook wordt er verwacht dat je de naam van uw topic meegeeft. Naar deze topic zal hij de messages sturen. Zoals te zien is gebruiken we weer de @Value annotatie van het Spring framework om de naam van onze topic niet in onze code te moeten plaatsen. Boven deze

projects/happyfleetruleengine/subscriptions/bp-ruben-subscription

Topic

projects/happyfleetruleengine/topics/ruben-bp

Delivery Type

Pull

Subscription expiry ?

The subscription will never expire, no matter the activity.

Acknowledgement Deadline ?

60 Seconds

Message retention duration

7 Days 0 Hours 0 Minutes

Retain acknowledged messages

No

Figuur 4.2: Configuratie van de topic voor de gebruikte subscription

methode zie je niet alleen de annotatie `@Bean` maar ook `@ServiceActivator` met de naam van de bean uit de eerste methode. Dit zal ik zo meteen uitleggen wanneer we aan het stukje komen van de `GooglePubSubPublisher` interface. De derde methode returned een `JacksonPubSubMessageConverter`. Door deze bean aan te maken weet Spring automatisch dat hij deze klasse moet gebruiken om onze objecten om te zetten naar het Json-formaat.

```

1 @Configuration
2 public class GooglePubSubConfig {
3
4     @Bean("bachelorproef-channel-output")
5     public MessageChannel createBachelorproefChannelOutput() {
6         return new DirectChannel();
7     }
8
9     @Bean
10    @ServiceActivator(inputChannel = "bachelorproef-channel-output")
11    public MessageHandler messageSender(
12        PubSubOperations pubsubTemplate,
13        @Value("${pubsub.topic}") String topicName
14    ) {
15        return new PubSubMessageHandler(pubsubTemplate, topicName);
16    }
17
18    @Bean

```

```
19     public JacksonPubSubMessageConverter
        createJacksonMessageConverter(final ObjectMapper
        objectMapper) {
20         return new JacksonPubSubMessageConverter(objectMapper)
            ;
21     }
22
23 }
```

GooglePubSubPublisher

Deze klasse heeft maar één methode, namelijk `publishData`. Deze methode verstuurt het Data-object naar de topic via het kanaal *bachelorproef-channel-output*. Er hoeft geen implementatie te zijn van deze methode, het Spring-framework regelt al het andere werk in jouw plaats. Maar toch is het handig om even te verduidelijken hoe dit allemaal in zijn werk gaat. Wanneer de methode `publishData` met een Data object opgeroepen wordt, dan verzendt hij deze data via het kanaal dat beschreven staat in de annotatie `@MessagingGateway`. Maar deze interface kan dit werk niet op zijn eentje doen en zoekt daarom hulp in de configuratie. Als we dan teruggaan naar de `GooglePubSubConfig` klasse, dan zien we boven de methode die een `MessageHandler` teruggeeft de `@ServiceActivator` staan en dezelfde naam van het kanaal. Dus om kort samen te vatten: als de methode `publishData` opgeroepen wordt, dan kijkt hij welk kanaal hij nodig heeft om te verzenden. Dan zoekt hij een `MessageHandler` die geactiveerd wordt wanneer zijn kanaal gebruikt wordt en deze handler gebruikt spring om de message dan effectief te verzenden. Omdat de bean die de `MessageChannel` teruggeeft de naam "bachelorproef-channel-output" heeft, en deze naam van het kanaal ook in de annotatie `@ServiceActivator` staat, weet Spring dat dit kanaal moet gebruikt worden.

```
1 @MessagingGateway(defaultRequestChannel = "bachelorproef -
    channel-output")
2 @Component
3 public interface GooglePubSubPublisher {
4     void publishData(Data data);
5
6 }
```

4.2 Consumer project

Nu overlopen we eens alle specifieke code die nodig is voor dit onderzoek in de consumer applicatie. Dit is de implementatie van de verschillende technologieën.

Entity klasse

Zoals vermeld in het vorige hoofdstuk hebben we per technologie één entity klasse. We bekijken het voorbeeld van de DataGPS klasse, want voor de andere technologieën is de implementatie analoog.

```
1 @Entity
2 public class DataGPS {
3
4     public DataGPS( int dataId, String name, String
        description, Date sendedDate, Date receivedDate, long
        timeDiff) {
5         this.dataId = dataId;
6         this.name = name;
7         this.description = description;
8         this.sendedDate = sendedDate;
9         this.receivedDate = receivedDate;
10        this.timeDiff = timeDiff;
11    }
12
13    @Id
14    @GeneratedValue(strategy = GenerationType.IDENTITY)
15    private long id;
16
17    private int dataId;
18    private String name;
19    private String description;
20    private Date sendedDate;
21    private Date receivedDate;
22    private long timeDiff;
23 }
```

Boven de klasse staat er een annotatie uit de package javax.persistence. De @Entity zorgt ervoor dat van deze klasse automatisch een tabel aangemaakt wordt in de databank. De attributen van deze klasse zullen automatisch ook de velden zijn van deze tabel.

In de constructor zien we dat alle attributen van deze klasse aanwezig zijn als parameter, behalve het attribuut id. Dit is niet nodig om in de constructor mee te geven aangezien deze waarde automatisch gecreëerd wordt. Door de annotatie @Id zeg je dat dit veld het id is van de tabel en dus uniek moet zijn. Dit is eveneens de primary key van de tabel. Door de tweede annotatie @GeneratedValue zorg je ervoor dat er automatisch een waarde aangemaakt wordt die uniek is. Je moet je dus voor de rest niets meer aantrekken van dit attribuut.

Repository

Ook hier bekijken we alleen het voorbeeld van de DataGPSRepository, voor de andere technologieën is dit opnieuw analoog.


```
1 public interface DataGPSRepository extends CrudRepository<
   DataGPS, Long> {
2 }
```

Op het eerste zicht lijkt het alsof deze klasse geen implementatie heeft. Toch kan je bij deze repository bijvoorbeeld de methode `save(DataGPS data)` oproepen om een object van de klasse `DataGPS` op te slaan. Dit komt omdat deze interface extends van de interface `CrudRepository<T, ID>`. Deze interface zorgt ervoor dat je op een object van de klasse, die je meegeeft in de plaats van `T`, de basisoperaties kan uitvoeren zoals `save`.

4.2.1 Kafka

KafkaConfig

Eerst en vooral zijn er opnieuw enkele waardes die we liever niet weergeven. In dit geval zijn er vier waardes: de naam van de topic, het bootstrapadres (het adres waar de server zich bevindt), een key en een secret. Sommige van deze variabelen zullen gebruikt worden in de `application.yml` file waar ook deze vier waardes zich bevinden. Anderen, zoals de topic, worden dan weer gebruikt in de implementatie zelf. De namen die we aan deze variabelen gegeven hebben in dit project zijn: `kafka.topic.name` (voor de naam van de topic), `kafka.bootstrapAddress` (voor het bootstrapadres), `kafka_key` en `kafka_secret` (voor de key en de secret).

Verder wordt er voor de configuratie enkel de `application.yml` gebruikt en dus geen aparte `KafkaConfig` klasse. Dit is hoe de configuratie er uit ziet voor Kafka:

```
1 spring:
2   kafka:
3     consumer:
4       bootstrap-servers: ${kafka.bootstrapAddress}
5       group-id: foo
6       auto-offset-reset: latest
7       key-deserializer: org.apache.kafka.common.
8         serialization.StringDeserializer
9       value-deserializer: org.apache.kafka.common.
10        serialization.StringDeserializer
11
12     properties:
13       ssl.endpoint.identification.algorithm: https
14       sasl.mechanism: PLAIN
15       request.timeout.ms: 20000
16       retry.backoff.ms: 500
17       sasl.jaas.config: org.apache.kafka.common.security
18         .plain.PlainLoginModule required username="${
19           kafka_key}" password="${kafka_secret}";
20       security.protocol: SASL_SSL
```

We zien dat het bootstrapadres meegegeven wordt. Op deze manier weet de applicatie op welke server hij moet gaan zoeken. Hier worden er geen serializers gebruikt maar deserializers, wat voor zich spreekt. Om de waarde van de Json te deserializeren wordt de `StringDeserializer` gebruikt uit de package `org.apache.kafka.common.serialization`. Om de key te deserializeren wordt de `StringDeserializer` gebruikt uit dezelfde package. Verder hoeven we ons geen zorgen te maken over het deserializeren. Het Spring framework regelt al dit werk voor ons. We kunnen er dus van uit gaan dat als er een message binnen komt dit automatisch op een goede manier omgezet wordt. Verder zien we ook dat de `kafka_key` en `kafka_secret` variabelen gebruikt worden om het wachtwoord en gebruikersnaam mee te geven.

KafkaReceiver

Deze klasse is verantwoordelijk voor het ontvangen van objecten die van de Kafka topic komen. Deze klasse heeft twee attributen, een `ObjectMapper` en de `DataKafkaRepository` om de objecten op te slaan in de databank. Door deze mee te geven als parameter in de constructor zorgt Spring Boot er opnieuw voor dat deze automatisch gecreëerd worden.

Deze klasse bevat slechts één methode: `listen(String message)`. Hier zien we dat er boven de annotatie `@KafkaListener` te lezen valt. Deze annotatie krijgt de naam van de topic mee en het `groupId`. Hierdoor weet de applicatie van waar hij berichten moet lezen. Door deze annotatie zeg je dat als er een bericht uitgelezen moet worden uit deze topic, dat je deze methode moet uitvoeren. De String die de methode meekrijgt als parameter werd al automatisch gedeserialiseerd door de configuratie in `application.yml`.

In deze methode lezen we via de `objectMapper` de message uit en vormen deze om naar ons domeinobject van het type `Data`, dat dezelfde klasse is als bij de producer applicatie. Hierna maken we een nieuwe datum aan om te registreren wat de tijd was van het uitlezen van de message. Hierna vragen we aan ons object wat de tijd was van verzenden. Dit trekken we van elkaar af en op deze manier komen we te weten hoeveel milliseconden het verschild tussen het uitlezen van de message en het verzenden ervan. Vervolgens slaan we deze gegevens op als een object van de klasse `DataKafka`. Veel van deze informatie halen we uit de oorspronkelijke message van het type `Data`. Maar voor de analyse van dit onderzoek is het ook handig om extra attributen mee te geven zoals het verschil. Daarna wordt de methode `save` uit de repository uitgevoerd. Op deze manier wordt dit object opgeslagen in de databank.

```
1 @Component
2 public class KafkaReceiver {
3
4     private ObjectMapper objectMapper;
5     private DataKafkaRepository dataKafkaRepository;
6
7     @public KafkaReceiver(final ObjectMapper objectMapper,
8                           final DataKafkaRepository dataKafkaRepository) {
9         this.objectMapper = objectMapper;
10        this.dataKafkaRepository = dataKafkaRepository
11        ;
12    }
```

```

11     }
12
13     @KafkaListener(topics = "${kafka.topic.name}", groupId = "
14         foo")
15     public void listen(String message) throws Exception {
16
17         try {
18             Data data = objectMapper.readValue(message,
19                 Data.class);
20             System.out.println(data);
21
22             Date received = new Date();
23             Date sendd = data.getSendedDate();
24
25             long difference = received.getTime() - sendd.
26                 getTime();
27
28             dataKafkaRepository.save(new DataKafka(data.
29                 getId(), data.getName(), data.
30                 getDescription(), data.getSendedDate(),
31                 received, difference));
32         } catch (IOException e) {
33             e.printStackTrace();
34         }
35
36         System.out.println("Received Messasge in group foo: "
37             + message);
38     }
39 }

```

4.2.2 RabbitMq

RabbitMqConfig

Ook hier wordt er enkel gebruikt gemaakt van de application.yml om RabbitMq te configureren. Eerst en vooral zijn er ook hier enkele variabelen die we liever niet blootstellen, zoals de naam van de queue. Deze zal gebruikt worden tijdens de implementatie in de klasse RabbitMqReceiver. Hiernaast willen we ook de username, password en het hostadres niet tonen. Deze zullen gebruikt worden in deze application.yml file. De namen van deze variabelen zijn respectievelijk: rabbitmq_queue_name, rabbitmq_username, rabbitmq_password en rabbitmq_host.

De rest van de configuratie ziet er als volgt uit:

```

1 spring:
2     rabbitmq:
3         host: ${rabbitmq_host}
4         port: 16379

```

```

5      username: ${rabbitmq_username}
6      password: ${rabbitmq_password}
7      virtual-host: "rabbitmq"
8      ssl:
9          enabled: true
10         algorithm: "TLSv1.2"
11     listener:
12         simple:
13             default-requeue-rejected: false
14         direct:
15             default-requeue-rejected: false

```

Je ziet dat het hostadres meegegeven wordt, alsook de username en password.

RabbitMqReceiver

Deze klasse is zo goed als gelijk aan de KafkaReceiver. Het is evident dat de DataKafkaRepository vervangen is door de DataRMQRepository. Boven de methode receiveMessage staat ook een andere notatie namelijk @RabbitListener. Door deze annotatie boven de methode te zetten, weet de applicatie dat hij deze methode moet uitvoeren als er op de opgegeven queue een bericht komt. In deze methode wordt de message eerst weer omgezet in een Data object en dan opgeslagen als een DataRMQ object met de waarden van de message en het verschil tussen het verzenden en het ontvangen.

```

1  @Component
2  public class RabbitMqReceiver {
3
4      private ObjectMapper objectMapper;
5
6      private DataRMQRepository dataRMQRepository;
7
8      public RabbitMqReceiver(final ObjectMapper objectMapper,
9                             final DataRMQRepository dataRMQRepository) {
10         this.objectMapper = objectMapper;
11         this.dataRMQRepository = dataRMQRepository;
12     }
13
14     @RabbitListener(queues = "${rabbitmq_queue_name}")
15     public void receiveMessage(String message) throws
16         Exception {
17         try {
18             Data data = objectMapper.readValue(message, Data.
19                 class);
20             System.out.println(data);
21
22             Date received = new Date();
23             Date sended = data.getSendedDate();

```

```

23         long difference = received.getTime() - sended.
           getTime();
24
25         dataRMQRepository.save(new DataRMQ(data.getId(),
           data.getName(), data.getDescription(), data.
           getSendedDate(), received, difference));
26     } catch (IOException e) {
27         e.printStackTrace();
28     }
29     System.out.println("Receiving message RabbitMq: " +
           message);
30 }
31 }

```

4.2.3 Google Pub/Sub

GooglePubSubConfig

Voor Google Pub/Sub werken we met een config klasse en ook met de application.yml. Er zijn vier zaken die meegegeven worden in deze laatste file. Twee ervan worden gebruikt in de implementatie, de andere twee moet je gewoon meegeven in deze file. De twee die gebruikt worden in de implementatie zijn pubsub.subscription en pubsub.autostartup. De eerste is de naam van de subscription, de tweede is een boolean om aan te duiden of de subscriber moet starten met luisteren als de applicatie wordt opgestart of niet. De andere twee configuraties die je moet meegeven zijn: spring.cloud.gcp.project-id en spring.cloud.gcp.credentials.location. De eerste is het id van het project, de tweede is de plaats waar de credentials staan om te kunnen inloggen op het Google Cloud Platform.

De GooglePubSubConfig klasse ziet er als volgt uit:

```

1 @Configuration
2 public class GooglePubSubConfig {
3
4     @Bean("bachelorproef-channel-input")
5     public MessageChannel createDeviceEventChannelInput() {
6         return new DirectChannel();
7     }
8
9     @Bean
10    public PubSubInboundChannelAdapter
        createDeviceChannelAdapter(
11        PubSubOperations pubSubTemplate,
12        @Value("${pubsub.subscription}") String
            subscriptionName,
13        @Qualifier("bachelorproef-channel-input")
            MessageChannel inputChannel,
14        @Value("${pubsub.autostartup}") Boolean autoStart) {
15

```

```

16         final PubSubInboundChannelAdapter adapter = new
            PubSubInboundChannelAdapter(pubSubTemplate,
                subscriptionName);
17         adapter.setOutputChannel(inputChannel);
18         adapter.setAckMode(AckMode.AUTO);
19         adapter.setPayloadType(Data.class);
20         adapter.setAutoStartup(autoStart);
21         return adapter;
22     }
23
24     @Bean
25     public JacksonPubSubMessageConverter
        createJacksonMessageConverter(final ObjectMapper
            objectMapper) {
26         return new JacksonPubSubMessageConverter(objectMapper)
            ;
27     }
28 }

```

Boven de klasse ziet u opnieuw de annotatie `@Configuration` staan om aan te geven aan het Spring framework dat dit een configuratie is. Verder wordt er een nieuwe `MessageChannel` aangemaakt. Dit is het 'kanaal' die gebruikt wordt om messages te ontvangen. Er wordt ook een naam gegeven aan dit kanaal namelijk "bachelorproef-channel-input". Vervolgens moet er een `PubSubInboundChannelAdapter` aangemaakt worden die enkele configuraties bevat. Als parameter van deze methode moet je een `PubSubOperations` meegeven, dit is te vergelijken met de `KafkaTemplate` of `RabbitTemplate`. Verder moet je de naam van de subscription meegeven en ook welke `MessageChannel` hij moet gebruiken om messages te ontvangen. Via de `@Qualifier` annotatie zeg je dat hij de `MessageChannel` moet gebruiken die we daarnet aangemaakt hebben. Als laatste geef je dan nog mee of het starten met luisteren naar de topic automatisch moet gebeuren of niet. In deze methode stel je dan nog in dat de payload van de message, die een `Json` is, van het type `Data` is. Door de laatste methode, die een `JacksonPubSubMessageConverter` aanmaakt, kan deze `Json` automatisch omgezet worden in een instantie van `Data`.

GooglePubSubReceiver

Dit doet ongeveer hetzelfde als de vorige receivers. Alleen hoeft je hier geen `ObjectMapper` te gebruiken omdat deze al is meegegeven aan de `JacksonPubSubMessageConverter`. Verder wordt er gebruik gemaakt van de `DataGPSRepository` om objecten van het type `DataGPS` op te slaan.

De methode `storeData` krijgt geen message mee in de vorm van een `String`, want het omzetten van de message naar een object van `Data` is al gebeurd door de `JacksonPubSubMessageConverter`. Je hebt dus meteen het `Data` object dat binnenkomt. Je hoeft dus enkel maar de juiste waardes op te vragen en mee te geven aan een nieuw object van de klasse `DataGPS`. Ook hier werd het verschil berekend tussen het verzenden en het ontvangen. Boven de methode staat de annotatie `@ServiceActivator` die als parameter het kanaal heeft

waar messages op toekomen. Dit wil zeggen als een message langs dit kanaal binnenkomt, dat de applicatie deze methode moet uitvoeren.

```
1 @Component
2 public class GooglePubSubReceiver {
3
4     private DataGPSRepository dataGPSRepository;
5
6     public GooglePubSubReceiver(final DataGPSRepository
7         dataGPSRepository) {
8         this.dataGPSRepository = dataGPSRepository;
9     }
10
11     @ServiceActivator(inputChannel = "bachelorproef-channel-
12         input")
13     public void storeData(final Data data) throws Exception {
14         System.out.println("--- Getting data: name: " + data.
15             getName() + ", description: " + data.getDescription
16             () +
17             ", id: " + data.getId() + ", sended: " + data.
18             getSendedDate().toString() + ", received: " + new
19             Date().toString() + " ---");
20
21         Date received = new Date();
22         Date sended = data.getSendedDate();
23
24         long difference = received.getTime() - sended.getTime
25             ();
26
27         dataGPSRepository.save(new DataGPS(data.getId(), data.
28             getName(), data.getDescription(), data.
29             getSendedDate(), received, difference));
30     }
31 }
```


5. Conclusie

5.1 Snelheid

5.1.1 10 000 objecten

Het eerste wat we gaan analyseren en een conclusie uit trekken is het verzenden van 10 000 objecten. Eerst kijken we naar de waardes apart per technologie, daarna gaan we alle technologieën gaan vergelijken met elkaar.

Google Pub/Sub

De resultaten van de snelheden bij 10 000 objecten van Google Pub/Sub zien er als volgt uit:

	Tijd (ms)
min.	1366
gem.	11665.7694
max.	19365
# ontvangen.	10 000

Tabel 5.1: Verschil tussen ontvangen en verzenden (in ms) - Google Pub/Sub

We zien dus dat het kleinste verschil iets meer dan één seconde is, en het grootste verschil bijna 20 seconden. Het gemiddeld aantal seconden ligt rond de 11,5 seconden.

Kafka

De resultaten van de snelheden bij 10 000 objecten van Kafka zien er als volgt uit:

	Tijd (ms)
min.	1008
gem.	20339.5169
max.	38537
# ontvangen.	10 000

Tabel 5.2: Verschil tussen ontvangen en verzenden (in ms) - Kafka

We zien dus dat het kleinste verschil nog dichterbij de één seconde ligt dan bij Google Pub/Sub. Het grootste verschil is bijna 40 seconden. Het gemiddeld aantal seconden ligt rond de 20 seconden.

RabbitMq

De resultaten van de snelheden bij 10 000 objecten van RabbitMq zien er als volgt uit:

	Tijd (ms)
min.	598
gem.	21780.9111
max.	41805
# ontvangen.	10 000

Tabel 5.3: Verschil tussen ontvangen en verzenden (in ms) - RabbitMq

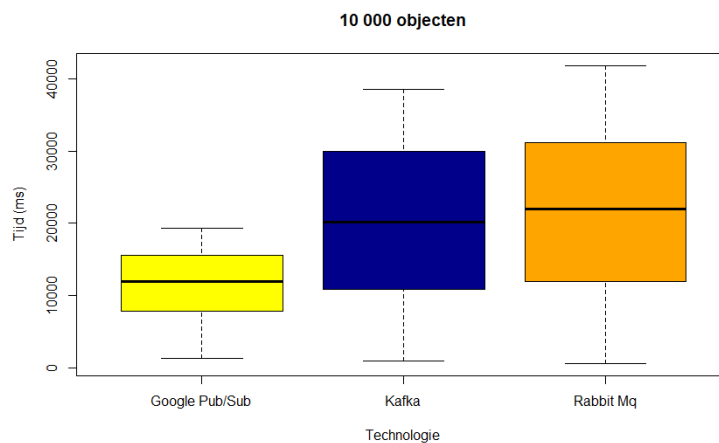
Het kleinste tijdverschil is iets meer dan een halve seconde. Het grootste verschil is meer dan 40 seconden. Het gemiddeld aantal seconden is bijna 42 seconden.

Vergelijking

Om een goede vergelijking te kunnen maken tussen de drie technologieën zetten we de resultaten bij 10 000 objecten van de verschillende technologieën even naast elkaar met behulp van drie boxplots die te zien is op figuur 5.1.

Bij zowel Google Pub/Sub, Kafka en RabbitMq zijn alle 10 000 objecten die verzonden zijn geweest, ook aangekomen. Dit wil zeggen dat bij 10 000 objecten geen enkel object mislukt is om te ontvangen. We kunnen bij deze hoeveelheid dus een duidelijke conclusie trekken als we kijken naar de snelheid tussen ontvangen en verzenden.

Eerst bekijken we de minimum waarden. Daarin is te zien dat RabbitMq de kortste tijd heeft van de drie minimum waarden. Dit is dubbel zo snel als we vergelijken met Kafka, die de tweede kortste tijd heeft. RabbitMq slaagt er in om in ongeveer een halve seconde een object te verzenden en te ontvangen. Kafka doet dit slechts in één seconde. Google



Figuur 5.1: Vergelijking bij 10 000 objecten

Pub/Sub is de slechtste leerling van de klas op het gebied van minimum tijd en doet er ongeveer 1,3 seconden over.

Laat ons nu kijken naar de maximum tijd. Hier doet Google Pub/Sub het veruit beter of de twee andere technologieën. Het doet er nooit langer dan 20 seconden over om een bericht te versturen en weer te lezen. Hier doen Kafka en RabbitMq het veel slechter. Zij doen er maximum 3,8 en 4,1 seconde over, en dit is veel meer of bij Google Pub/Sub.

Het resultaat van de maxima weerspiegelt zich ook in de gemiddeldes. Gemiddeld doet Google Pub/Sub er 11 seconden over. Het gemiddelde van Kafka en RabbitMq ligt dan weer wat hoger met 20 en 21 seconden.

Conclusie bij 10 000 objecten: Alhoewel de minimum tijd bij Google Pub/Sub hoger ligt of bij Kafka en RabbitMq, slaagt Google Pub/Sub er toch in om behoorlijk wat sneller al de data te verwerken.

5.1.2 100 000 objecten

Het lijkt ons ook interessant om eens te bekijken wat te resultaten zijn indien we het aantal objecten wat gaan verhogen. We nemen tien keer zo veel objecten en kijken wat we hiermee kunnen uit concluderen.

Google Pub/Sub

De resultaten van de snelheden bij 100 000 objecten van Google Pub/Sub zien er als volgt uit:

We zien dus dat het kleinste verschil iets minder is dan twee seconden, en het grootste verschil bijna 170 seconden, wat bijna drie minuten is. Het gemiddeld aantal seconden ligt rond de 88 seconden, wat bijna 1,5 minuten is.

	Tijd (ms)
min.	1753
gem.	88603.80396
max.	169580
# ontvangen.	100 000

Tabel 5.4: Verschil tussen ontvangen en verzenden (in ms) - Google Pub/Sub

Kafka

De resultaten van de snelheden bij 100 000 objecten van Kafka zien er als volgt uit:

	Tijd (ms)
min.	1055
gem.	175526.39283
max.	346431
# ontvangen.	100 000

Tabel 5.5: Verschil tussen ontvangen en verzenden (in ms) - Kafka

Het kleinste verschil blijft nog steeds dicht bij de één seconde liggen, alhoewel het verschil toch lichtjes gestegen is. Het grootste verschil ligt rond de 346 seconden, wat bijna zes minuten is. Het gemiddeld aantal seconden ligt rond de 175 seconden, omgerekend is dit bijna drie minuten.

RabbitMq

De resultaten van de snelheden bij 100 000 objecten van RabbitMq zien er als volgt uit:

	Tijd (ms)
min.	458
gem.	114365.12135045651
max.	224467
# ontvangen.	59 802

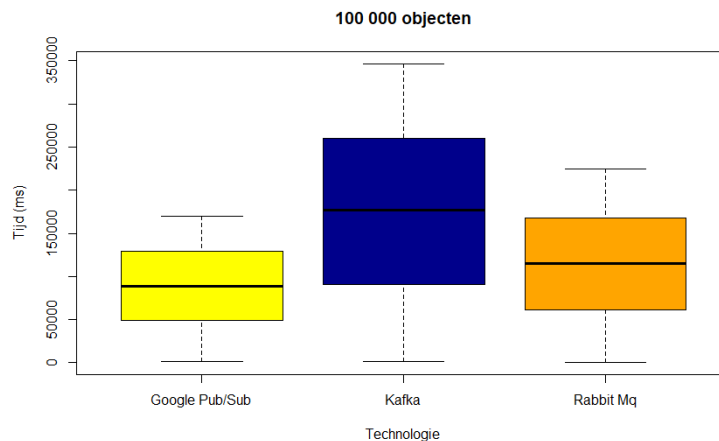
Tabel 5.6: Verschil tussen ontvangen en verzenden (in ms) - RabbitMq

Wat meteen opvalt, is dat bij deze hoeveelheid RabbitMq er niet in geslaagd is om tijdens dezelfde run alle 100 000 objecten binnen te halen. Natuurlijk haalt hij deze wel nog binnen indien de applicatie opnieuw opgestart wordt, maar het is toch opvallend dat maar iets meer of de helft meteen ontvangen wordt.

De minimum tijd is nog gedaald in vergelijking met 10 000 objecten. Hij heeft nu slechts minder dan een halve seconde nodig om te verzenden en te ontvangen. Het gemiddelde ligt rond de 114 seconden en het maximum iets meer dan 224 seconden. Natuurlijk zijn het gemiddelde en maximum minder belangrijk om te vergelijken aangezien niet alle 100 000 objecten zijn ontvangen.

Vergelijking

Ook bij deze hoeveelheid aan objecten zetten we de drie technologieën eens naast elkaar. Deze boxplots zijn te zien op figuur 5.2



Figuur 5.2: Vergelijking bij 100 000 objecten

Hierbij is het minder relevant om RabbitMq helemaal te gaan vergelijken met de twee andere technologieën aangezien maar iets meer dan de helft van de objecten ook ontvangen zijn geweest.

Eerst gaan we de minimum waarden bekijken, hier kunnen we ook nog vergelijken met RabbitMq. Daarin is te zien dat RabbitMq de kortste tijd heeft van de drie minimum waarden. Deze waarde is zelf nog minder dan bij de vorige vergelijking van 10 000 objecten. Kafka blijft redelijk stabiel van de kortste tijd en schommelt nog steeds rond de seconde. Bij Google Pub/Sub zien we toch een stijging en deze waarde is nu 1,7 seconden geworden.

Laat ons nu kijken naar de maximum tijd. Hier doet opnieuw Google Pub/Sub het veruit beter of de twee andere technologieën. We zouden kunnen zeggen dat de waarde bij RabbitMq ook nog goed meevalt, maar dit zou een vertekend beeld geven omdat niet alle objecten ontvangen zijn geweest. Dus het heeft enkel zin om voor de maximum waarde Google Pub/Sub te gaan vergelijken met Kafka. Bij Kafka ligt de maximum waarde veel hoger. Het heeft er maximum ongeveer 346 seconden voor nodig, wat bijna dubbel zo veel is dan bij Google Pub/Sub.

Het resultaat van de maxima weerspiegelt zich ook opnieuw in de gemiddeldes. Ook hier houden we geen rekening met RabbitMq want niet alle objecten zijn ontvangen. We kunnen eigenlijk al zeker concluderen dat bij 100 000 objecten Google Pub/Sub sowieso sneller zou geweest zijn dan RabbitMq aangezien het gemiddelde voor minder ontvangen waarden al hoger ligt dan bij Google Pub/Sub. Gemiddeld doet Google Pub/Sub er 88 seconden over. Het gemiddelde van Kafka ligt dan weer wat hoger met 175 seconden.

Conclusie bij 100 000 objecten: Alhoewel de minimum tijd bij Google Pub/Sub opnieuw

hoger ligt of bij Kafka en RabbitMq, slaagt Google Pub/Sub er toch in om behoorlijk wat sneller al de data te verwerken.

5.1.3 1 000 000 objecten

De laatste hoeveelheid objecten die we onderzoeken is nog eens tien keer zo veel. We kijken ook eens wat het resultaat is bij 1 000 000 objecten.

Google Pub/Sub

De resultaten van de snelheden bij 1 000 000 objecten van Google Pub/Sub zien er als volgt uit:

	Tijd (ms)
min.	3327
gem.	577596.5258098603
max.	1037761
# ontvangen.	556 454

Tabel 5.7: Verschil tussen ontvangen en verzenden (in ms) - Google Pub/Sub

We zien dus dat het kleinste verschil iets meer is dan drie seconden, en het grootste verschil bijna 1037 seconden, wat bijna 17 minuten is. Het gemiddeld aantal seconden ligt rond de 577 seconden, wat iets meer dan 9,5 minuten is. Hier is Google Pub/Sub voor het eerst niet in geslaagd om alle objecten te ontvangen. Het zal dus ook hier wat moeilijker zijn om het gemiddelde en het maximum te vergelijken.

Kafka

De resultaten van de snelheden bij 1 000 000 objecten van Kafka zien er als volgt uit:

	Tijd (ms)
min.	3930
gem.	1719014.245389
max.	3424980
# ontvangen.	1 000 000

Tabel 5.8: Verschil tussen ontvangen en verzenden (in ms) - Kafka

Het kleinste verschil ligt nu niet meer rond de seconde, maar is nu gestegen naar bijna vier seconden. Het grootste verschil ligt rond de 3424 seconden, wat bijna een uur is. Het gemiddeld aantal seconden ligt rond de 1719 seconden, omgerekend is dit bijna een half uur.

RabbitMq

De resultaten van de snelheden bij 1 000 000 objecten van RabbitMq zien er als volgt uit:

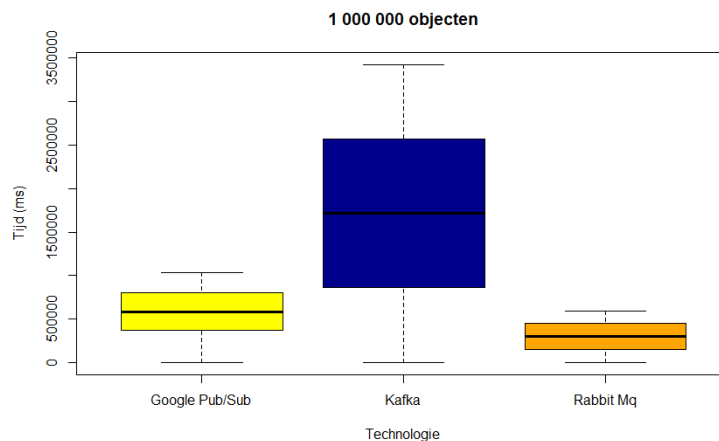
	Tijd (ms)
min.	1345
gem.	300640.46033216227
max.	590104
# ontvangen.	59 802

Tabel 5.9: Verschil tussen ontvangen en verzenden (in ms) - RabbitMq

RabbitMq is ook hier niet in geslaagd om alle objecten te ontvangen. De minimum tijd is voor het eerst gestegen naar iets meer dan een seconde. Het gemiddelde ligt op 300 seconden. En het duurt nooit langer dan 590 seconden of een kleine tien minuten om een object te ontvangen.

Vergelijking

Ook bij deze hoeveelheid aan objecten zetten we de drie technologieën eens naast elkaar. Deze boxplots zijn te zien op figuur 5.3



Figuur 5.3: Vergelijking bij 1 000 000 objecten

Nu is het wat moeilijker om de snelheden met elkaar te gaan vergelijken aangezien de drie verschillende technologieën verschillend aantal objecten hebben ontvangen. Toch overlopen we de technologieën eens.

Eerst gaan we de minimum waarden bekijken, hier kunnen we nog alle drie met elkaar vergelijken. Daarin is te zien dat RabbitMq nog steeds de kortste tijd heeft van de drie minimum waarden. Deze waarde is wel gestegen naar iets boven de seconde, maar blijft nog steeds de beste. Kafka is voor het eerst ook opvallend gestegen. De waarde is bijna vier keer zo groot geworden. Bij Google Pub/Sub zien we ook een stijging, het minimum

ligt nu ook rond de drie seconden. Kafka heeft dus bij deze hoeveelheid het meeste tijd nodig als minimum waarde.

Laat ons nu kijken naar de maximum tijd. Dit is wat moeilijker om te vergelijken aangezien al de technologieën verschillend aantal objecten ontvangen hebben. Kafka heeft het meeste tijd nodig, maar heeft ook het meeste objecten ontvangen. RabbitMq en Google Pub/Sub hebben ongeveer evenveel ontvangen, en daar zien we RabbitMq toch duidelijk beter zijn voor de maximum waarde.

Het resultaat van de maxima weerspiegelt zich ook opnieuw in de gemiddeldes. De verhoudingen zijn hetzelfde als bij de maximum waardes

Conclusie bij 1 000 000 objecten: RabbitMq doet het beter dan Google Pub/Sub. Kafka is moeilijk te vergelijken met de twee andere.

5.1.4 Conclusie snelheid

Op het gebied van snelheid is Kafka over het algemeen gezien de slechtste. Google Pub/Sub doet het over het algemeen gezien het beste, behalve bij super veel objecten (hier: 1 000 000) doet RabbitMq beter dan Google Pub/Sub.

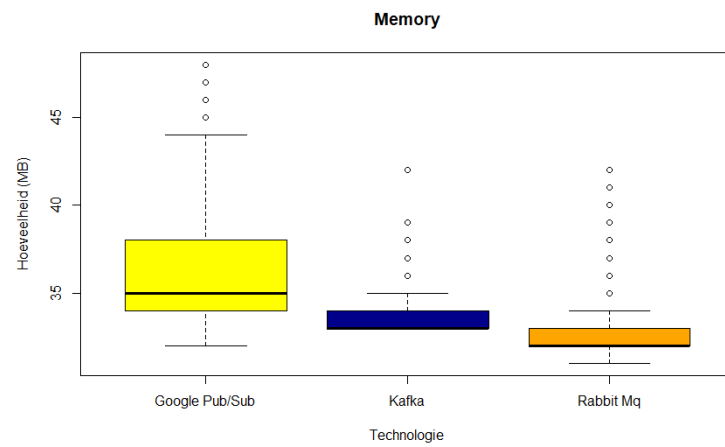
5.2 Memory

5.2.1 1 000 objecten

Omdat de operatie om aan de Java runtime de gebruikte memory te vragen te lang duurt, wordt er enkel gekeken bij 1 000 objecten. Er wordt hier geen rekening gehouden met de snelheden.

5.2.2 Conclusie memory

Op figuur 5.4 ziet u de boxplots van het resultaat van het memory gebruik. Het is duidelijk te zien dat er bij alle technologieën enkele uitschieters zijn. Maar het is ook duidelijk dat over het algemeen voor het memory gebruik RabbitMq beter is. Het gemiddelde, maximum en het minimum liggen lager in vergelijking met de andere. Gemiddeld heeft RabbitMq 33.138 MB nodig. Google Pub/Sub en Kafka hebben gemiddeld 36.31 en 33.601 nodig. RabbitMq is dus beter maar het verschil met Kafka is klein. Het is enkel Google Pub/Sub die veel meer memory nodig heeft dan de rest.



Figuur 5.4: Vergelijking memory

6. Besluit

Wanneer snelheid voor een bedrijf het belangrijkste is, dan lijkt Google Pub/Sub het beste. Tenzij je heel veel objecten hebt (meer dan 1 000 000) dan lijkt RabbitMq sneller te werken. Het memory gebruik is het best bij RabbitMq maar Kafka doet het niet veel slechter. Bij Kafka ben je het meest zeker dat je effectief al je verzonden data meteen ontvangt. Hier slaagt RabbitMq het slechtste op.

Voor TVH specifiek lijkt het belangrijkste dat alle data zeker meteen ontvangen wordt. Omdat Kafka hierop het beste scoort en bijna de beste is op basis van memory gebruik, lijkt Kafka de beste technologie om te gebruiken binnen dit bedrijf.

Een verder onderzoek zou dit onderzoek enkele keren kunnen herhalen om te bepalen hoe frequent verzonden data niet meteen aankomt. Ook zouden we de applicaties in een ander onderzoek in de cloud kunnen laten draaien zodat deze niet afhankelijk zijn van een lokale pc of laptop.

A. Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Introductie

Binnen TVH is er dus heel wat input van data die via microservices naar de juiste componenten verstuurd worden. Het spreekt voor zich dat niet ieder component, alle data nodig heeft. Daarom maakt dit bedrijf voornamelijk gebruik van de technologie *Kafka* om met deze microservices te werken. Dit onderzoek zal nagaan of *Kafka* inderdaad wel de beste technologie is om al deze data te verwerken in dit bedrijf. Aan de hand van deze onderzoeksvraag en deelvragen komt dit onderzoek hopelijk tot een besluit welke technologie het meest geschikt is:

- Welke technologie is het best om met microservices te werken voor het bedrijf TVH?
 - Bestaan er nog alternatieven voor *Kafka* en *RabbitMQ*?
 - Wat zijn de bevindingen van gebruikers?
 - Welke technologie is het snelst?

A.2 Literatuurstudie

In het onderzoek van Shadija, Rezai en Hill (2017) staat te lezen dat microservices de business analysts helpen om grote schaalbare applicaties te maken. Het grote voordeel hiervan is flexibiliteit. Als er nieuwe functionaliteiten moeten worden gemaakt dan is het

door de microservices gemakkelijk te implementeren. Vooral in het Internet of Things (IoT) domein kan dit het werk versoepelen. Dus voor het bedrijf TVH lijkt het de meest geschikte manier om de input van al de verzamelde data te verwerken.

Ook in andere onderzoeken naar microservices wordt *Kafka* gebruikt. Zoals in het onderzoek van Khazaei, Bannazadeh en Leon-Garcia (2017). Als we kijken naar de conclusie uit dit onderzoek, blijkt dat *Kafka* in grote lijnen het best scoort. De andere technologieën die in dit onderzoek gebruikt werden zijn *Spark* en *Cassandra*.

Het verschil van deze bachelorproef-onderzoek met het onderzoek van Shadija e.a. (2017) en het dat van Khazaei e.a. (2017) is dat dit onderzoek nagaat welke technologie het beste is voor het bedrijf TVH. Het besluit van dit onderzoek is dus niet noodzakelijk een algemeen besluit voor alle bedrijven die met microservices werken. Het onderzoek van Khazaei e.a. (2017) sluit hier het dichtst bij aan omdat het ook *Kafka* en andere technologieën vergelijkt, maar het onderzoek legt meer de nadruk hoe flexibel een programmeerbaar, zelf-besturend IoT-platform is, gebruik makend van microservices. Het vergelijken van verschillende technologieën bij Khazaei e.a. (2017) is dus maar een klein onderdeel van het onderzoek en wordt bovendien in een andere architectuur toegepast zoals de titel meedeelt: ‘*SAVI-IoT: A Self-Managing Containerized IoT Platform*’. De meerwaarde van deze vergelijking voor de conclusie is dus niet zo groot voor Khazaei e.a. (2017).

Ook Nycander (2015) en Cherradi, El Bouziri en Boulmakoul (2017) behandelen microservices in hun onderzoek.

A.3 Methodologie

Om te bepalen welke technologie het beste is bij het gebruiken van microservices, zal dit onderzoek de verschillende technologieën vergelijken. Eerst zal er een rondvraag gehouden worden over de bevindingen en de voor- en nadelen van *Kafka* en *RabbitMq*. Er wordt ook gepolst of medewerkers met nog andere technologieën reeds gewerkt hebben en wat daar de bevindingen zijn. Er zal voornamelijk gewerkt worden met open vragen waardoor er veel nieuwe nuttige informatie zal ontstaan voor dit onderzoek.

Dan zal er onderzocht worden of er effectief nog alternatieven bestaan voor *Kafka* en *RabbitMq*.

Als laatste zal op een virtuele machine de realiteit nagebootst worden. Dit wordt gedaan door elke technologie op deze virtuele machine te zetten en daarna te gaan meten wat de snelheden zijn bij het opvragen, verwerken, ... van voorbeelddata.

A.4 Verwachte resultaten

Het resultaat van dit onderzoek zal hopelijk aanwijzen welke technologie het meest geschikt is om deze hoeveelheid van data aan te kunnen. We hopen dit te zien door cijfergegevens

van de snelheden van uitvoering.

A.5 Verwachte conclusies

Aangezien TVH al gebruik maakt van *Kafka*, en ook in andere onderzoeken *Kafka* gebruikt werd of bestempeld werd als beste oplossing, kunnen we in dit onderzoek hopelijk ook concluderen dat *Kafka* de beste oplossing is om met microservices om te gaan binnen TVH met hun specifieke data.

Bibliografie

- Cherradi, G., El Bouziri, A. & Boulmakoul, A. (2017, november 1). A Generic microservice-based architecture for Smart HazMat Transportation Ecosystem.
- Google. (2019). What Is Cloud Pub/Sub?
- Guidi, C., Lanese, I., Mazzara, M. & Montesi, F. (2017). *Microservices: A Language-Based Approach*. Springer International Publishing.
- i-scoop. (2019). What is the Internet of Things?
- Johansson, L. (2015). Part 1: RabbitMQ for beginners - What is RabbitMQ?
- Johansson, L. (2016). Part 1: Apache Kafka for beginners - What is Apache Kafka?
- Khazaei, H., Bannazadeh, H. & Leon-Garcia, A. (2017, augustus 1). *SAVI-IoT: A Self-Managing Containerized IoT Platform* (masterscriptie, University of Toronto, Ontario, Canada).
- McClelland, C. (2019). What Is IoT? – A Simple Explanation of the Internet of Things.
- Nycander, P. (2015, juni 1). *Learning-Based Testing of Microservices* (masterscriptie, KTH Royal Institute of Technology, School of Computer Science en Communication (CSC)).
- Shadija, D., Rezai, M. & Hill, R. (2017, september 7). *Towards an Understanding of Microservices* (masterscriptie, Department of Computing Sheffield Hallam University, UK, School of Computing en Engineering University of Huddersfield, UK).
- Sookocheff, K. (2015). Kafka in a Nutshell.
- Trends, G. (2019). Kafka vs RabbitMq.
- Velthoven, J. K. (2016). 10 dingen die je moet weten over microservices.
- Xu, J. (2019). Service discovery in microservice architecture.