Assignment lab 02

YHL-18 CMT

Namn: Henrik Ählström

Task 1:

- **Continuous Integration vs. Continuous Delivery:**
  Continuous Integration – Integrerar(bygger och testar) förändringar I mjukvara/applikationer kontinuerligt samt testar så att förändringarna inte introducerar nya buggar/fel. Detta minskar risken för svåra intergrationsfel samt ger respons till utvecklaren. Väldigt användbart om flera utvecklare jobbar mot samma kodbas/repository.
  Continuous Delivery - Kontinuerlig mjukvaruleverans(tänk packa ihop till ett RPM paket som läggs I ett repo). Levererar någonstans (tex ett repo) men deploya inte till produktion per automatik, deployment får göras manuellt.
- **Jenkins vs Gitlab**
  Jenkins – En automationsserver för applikationsbyggnad, testning och CI/CD. Jenkis är en egen server som kan deployas på en VM eller I en container. Kan använda andra Version Control System än git. Ett stort urval av moduler för att utöka Jenkins kapabiliteter finns att installera.
  Gitlab – En tjänst för att hantera Git repos.
  Bägge alternativ har hand om en hel DevOps life-cycle, dvs allt från kod granskning till CI/CD.
- **SDN –** Software defined network. Syftar till att abrstrahera ett nätverks logic(control plane)från dess data plane(hårdvara) för att kunna styra nätverket via programvara och för att automatisera ett nätverk I högre grad. Ett SDN är mycket mer dynamiskt dvs det är “enklare” att konfigurera om ett nätverk baserat på behov och speciella events. Konfigurering kan I högre grad ske on-the-fly.
- **Openflow –** Ett protokoll ofta använt I SDN implementationer som möjliggör interfacing mellan dataplanet(nätverkshårdvara) och kontrollplanet(en server kallad (SDN)controller). Istället för att tex en switch använder ARP protokollet för att forwarda trafik används I OpenFlow används ett så kallat flowtable för att bestämma hur switchen ska forwarda paket, I detta flowtablet sätts de regler som switchen kollar på när den ska bestämma hur den ska bete sig. Dessa regler sätts från controllern.
  Om en table miss uppstår, dvs det finns ingen regel som matchar just det här paketet, forwardas paketet till controllern som tar ett beslut om bästa sätt att forwarda det, varje flowtable längs vägen blir då tilldelade den här regeln för nästkommande paket.
- **Control Plane vs Data Plane:**
  Control Plane – Nätverkets logic, bestämmer hur toplogin ska se ut samt hur paket ska forwardas.
  Data Plane – Skulle kunna syfta på hårdvara för enkelhetsskull. Har hand om hantering av paket baserat på hur control planet har bestämmt att den ska göra det.

- **Microservices:**
  - **API Gateway –** Används för att exponera APIs mot klienter. Fungerar som en entrypoint för klienter så de kommer åt bakomliggande APIs och routar trafiken dit den ska med hjälp av service discovery.
  - **What's a service –** En isolerad och gärna så självständig som möjligt funktion som tillsammans är löst sammankopplade med andra services på ett distrubierat vis för att tillsammans utgöra en applikation.
  - **Advantages –** Agilitet och flexibilitet vad det gäller utveckling, val av programmeringsspråk, CI/CD(Continuous Intetration/Deployment) samt flexibel skalning.
  - **Limitations/Pitfalls –** Komplexitet, distrubierade system har en viss komplexitet I sin natur. Allt från hur tjänster ska kommunicera, hur de ska hitta varandra och hur persistent data ska hanteras.
  - **Service Discovery –** Syftar till att garantera kommunikation mellan microservices I en ever-changing miljö.
- **Inter-Service Communication:**
  - **REST –** Representational state transfer – En mjukvaruarkitektur för design av webbapplikationer. Är stateless och använder klient-server modellen. Använder oftast HTTP.
  - **RPI/RPC –** Remote Procedure Call – En teknik för IPC som används av applikationer byggda enligt klient-server arkitektur för att executa något I ett annat namespace tex en annan nod eller en annan container. Vad jag förstår är syftet att förenkla utveckling av distrubierade applikationer genom att agera som om det vore ett lokalt call fast istället I en remote tappning.
  - **gRPC –** En opensource RPC variant från Google som använder HTTP/2.
  - **Message Queues –** En meddelandebuffer som temporärt håller meddelanden. Services kan kontakta kön för att hämta/skicka meddelande(tex requests och response) Meddelande lagras I kön tills dess att någon accessar det.
    I en message que modell publiseras ett meddelande en gång och konsumeras en gång,(till skillnad från pub/sub modellen, consumed X times), tänk unicast vs broadcast.
    En kö är ytterligare ett medel för att göra services självständiga pga att kön har hand om många funktioner som annars måste implemeteras I själva servicen.
  - **AMQP –** Advanced Message Queuing Protocol – Ett protokoll som användas vid meddelande hantering.
  - **RabbitMQ & Kafka :**
    RabbitMQ - Är en message broker mjukvara som använder sig av AMQP men har via moduler möjlighet att använda flera protokoll. Rabbit använder arkitekturen smart broker-dumb client dvs Rabbit gör det "tungrodda" I meddelande hanteringen, klienten behöver mer eller mindre bara skicka meddelandet(den biten behöver dock implementeras I koden). Rabbit kan använda sig av flera modeller, exempelvis: message que eller pub/sub
    Apache Kafka – Är en Distributed Stream Processing System som kan användas för messaging I stor skala, med låg latens och hög throughput. Använder dumb broker-smart client arkitektur(tvärtom från RabbitMQ).
    Snabb rundown: Meddelande(tex) I key-value format sparas I partationer som återfinns inom olika topics, consumers kan sedan läsa dessa meddelande.
    Kafka är en klustrad lösning där brokers återfinns på olika servrar, partitioner distrubieras ut över klustret, Kafka har inbyggd replikering då samma partition sprids till mer än 1 broker.

- **Storage:**
  - **OpenStack Cinder vs GlusterFS:**
    Cinder – Openstacks block storage tjänst. Cinder virtualiserar hanteringen av lagring genom att exponera ett API för att använda lagringen utan att veta något om vad för typ av block storage device eller vart den faktiskt finns dvs vet inget om den underliggande implementationen.
    GlusterFS – Distrubierat filsystem. Kan användas som lagringsbackend till Cinder. Mycket av skillnaden här är att Cinder är en tjänst för hantering och Gluster är ett distrubierat filsystem(som nämnts är kompatibelt med Cinder).
  - **Openstack Switft vs S3 Storage:**
    Openstack Switt - Openstacks object storage tjänst designat för stor skala. Swift likt Cinder exponerar ett API för att lagra och komma åt data.
    S3 Storage – AWS object storage tjänst.
    Dessa 2 skiljer sig lite I
    1: Hur dom lagrar (bucket för S3 och annan metod för Swift).
    2: Swift är opensource och native till Openstack, S3 är inte opensource.
    3: S3 verkar ha flera features än Swift.
    Om man kör Openstack lär man ju dock överväga om man vill binda in en tredjeparts lagringstjänst till sin övriga infrastruktur som man har full kontroll över, men där man då inte har särskilt mycket kontroll över S3.
  - **Block storage vs File Storage vs Object Storage:**
    Block storage – Lagrar data I "råformat", dvs bryter ner filer till bits/bytes och lagrar dessa bitar separat där varje bit har en adress för att hitta/urskilja den. Lagrar likt object storage I en icke hierarkisk struktur.
    File storage – Lagring I ett hierarkiskt filsystem, path är avgörande för att hitta filer.
    Object storage – Data lagras I så kallade objekt I en platt dvs icke hierarkisk pool. Relevant metadata som används för att hitta/urskilja datat läggs till (metadata och data lagras tillsammans) Används främst för lagring av ostrukturerad data. Key point: Går att skala mycket bra tack vare att det bryter från behovet att ha ett filsystem som klarar av data av den magnituden.
- **Kubernetes**
  - **RBAC –** Ett API för auktorisering som används för att begränsa användares åtkomst till resurser. Bygger på roller där permissions(tex get), vilken resurs samt vem(vad) det gäller för defineras. Tex ge <användare> access till det här namespacet eller den här specifika podden. Värt att påpeka att permissions är additiva, dvs att neka är inte en möjlighet.
  - **PV vs PVC:**
    PV(Persistent Volumes) – Resurs för persistent lagring I klustret som provitioneras av admin(statisk) eller av en storage class(dynamisk). Till skillnad från "vanliga" volymer är PVs livscykel självstående från den pod som använder den vilket betyder att om podden försvinner finns PVn kvar.
    PVC(Persistent Volume Claim) – En claim är ett anrop som görs av en service för att få en PV. Förutsatt att PVn inte skapades statiskt kan volymer skapas dynamiskt när en PVC görs, detta med hjälp av storage classes.
  - **Storage Classes –** En storage class definerar I princip vilken typ av lagring som används för att lagra data persistent, tex: Använder vi Ceph, AWS eller har vi kanske en class för SSD och en för HDD osv.
    En storage class behöver inte vara "förstådd" av en service, dvs man väljer storage class utefter behov utan att behöva veta hur lagringen är implementerad, med storage classes behöver man alltså inte deklarera vartifrån eller hur man får volymer, tänk istället för:
    "ge mig en XGB volym från NFS servern 192.168….." är det snarare "Ge mig en

volym från storage class X storlek YGB”, all information relaterad till vart lagring kommer ifrån finns I storage classen.
Jag vill dra en pararell till assignment 1 och säger därmed att Storage classes hjälper pods volymhantering att bli/vara deklarativ.

- **Ingess Controller –** Ingress är en resurs som möjliggör access utifrån klustret samt har hand om lastbalansering och TLS terminering. En ingress controller möjliggör Ingress funktion, utan en ingress controller gör inte en ingress resurs någon nytta. Kubernetes har support för GCE och Nginx men det finns en uppsjö av andra tex Traefik och HAProxy.
  I korta ord vill jag förklara det: Ingress Controller är den mjukvara som sköter åtkomst, lastbalansering och TLS hantering.

Task 5 essay:

Av: Henrik Ählström
CMT YHL-18

Message Qs

En message que är en av flera metoder för inter-process-communication och inter-threade-communication som ofta används för att lösa eventuella problem I distrubierade system(bland annat skalning). Message ques är vanligt I större Enterprise miljöer, men jag kan endå se fördelarna med att ha det I även I mindre miljöer, systemet blir organiserat och man får ett tydligt flöde. Allt detta bygger på att (minst)en kö skapas där publishers(sändare) och consumers(mottagare) skickar och hämtar meddelanden med hjälp av en message que som agerar mellanhand. En message que är inte enkelriktad, en publisher kan också vara en consumer och vice versa. Ett meddelande kan exempelvis vara en SQL query eller en HTTP request.

Om vi börjar med att reda ut message quens placering I det distrubierade systemet: message quen placeras vanligtvis som en mellanhand mellan frontend applikationer och backend applikationer I syfte att få dessa mindre sammankopplade och öka förmågan till pararell körning.
De olika tjänsterna kommunicerar istället genom en eller flera message ques som tillhandahålls av en message broker istället för att kommunicera direkt med varandra.

Vad är då en message broker?

En message broker är själva mjukvaran som dels tillhandahåller message ques men också ansvarar över att översätta så att meddelandet som kommer in från publishern kan tolkas av consumern samt att meddelanden routas till rätt consumer. Exempel på mjukvara som kan agera message broker är bland annat RabbitMQ och Apache Kafka.

Kan Redis användas för message ques?

Ja och nej, inte Redis per se, det finns dock minst en variant baserad på Redis kallad RestMQ, Riktiga Redis kan vara message broker, men använder sig av pub/sub modellen vilket är ett annat mönster för att hantera messages, message que är alltså inte synonymt med pub/sub.

Vilka problem löser en message que? Varför ska vi ha message ques?

Med en message que slipper vi en del statisk “binding” som ofta uppkommer I tex request/response modellen, detta betyder att istället för att varje nod/service/applikation I ett distrubierat system måste ha full vetskap om vart andra services befinner sig, behöver den endast veta vart message quen befinner sig eftersom det är med hjälp av message quen som services kommuncierar med varandra.
Med en message que finns det heller inget krav på att pulisher och consumer ska interagera med message quen samtidigt, meddelanden lagras I kön tills consumern är redo att ta emot dom.

En message que har även möjlighet att göra kommunkationen asynkron vilket möjliggör paralellitet, detta till skillnad från request/response modellen som är synkron. En message que är dock inte uteslutande asynkron, möjligheter att köra synkront finns, I synkront läge skickar producern och får ett ack tämligen snabbt(kommer från message brokern), jag tänker att detta lär användas om man dels vill säkra leverans hos bägge parter och vid eventdrivna system, exempelvis: när du har fått ack, fortsätt med det här…...
Skillanden är att I en synkron modell låses en applikations process(er) som har gjort en request och fortsätter först när denne har fått tillbaka någon typ av resultat/acknowledgment,
I en asynkron message que finns det lite olika approach: Antingen används fire-and-forget principen eller så har man flera bollar I luften samtidigt, dvs publisern skickar ett meddelande och färväntar sig ett ack, men under väntetiden skickas flera meddelanden.

AMQP(Advanced Message Queing Protocol) är ett av flera protokoll inom message handling som stödjer många olika patterns. AMQP möjliggör kryptering och har ett urval av olika metoder som dikterar hur meddelanden ska levereras, exempelvis:

- at-most-once(ett meddelande levereras en gång eller aldrig)
- at-least-once(meddelande levereras minst en gång)
- exactly once(garanteras levereras exakt en gång).

Inte alla brokers har stöd för AMQP, RabbitMQ är ett exempel på en broker som har det.

Säg att vi vill ha en message que, vad finns det för val?

Antingen likt tidigare nämnt med en message broker(också kallat MoM – Message Oriented Middlerware), till denna kategori hör bland annat RabbitMQ, där Rabbit då implementeras som en konfigurerbar mjukvarulösning antingen på en VM eller I en container.
Ett alternativ till detta är att använda ZeroMQ som är ett bibliotek för att skapa message ques utan behov av en message broker, detta faktum gör att ZeroMQ tenderar till att luta mer åt ligthweight hållet än fullblodad message handling. ZeroMQ använder sig av sockets för att utföra inter-process-communication. Några viktiga punkter att belysa som RabbitMQ har till fördel över ZeroMQ är:

- Persistence – Meddelanden kan sparas på disk I key-value format, fördelen här är att om Rabbit skulle gå ner tex till följd av en krasch, kan det startas upp igen I samma tillstånd som innan och således undvika att meddelande går förlorade.
- Mirroring – En message que speglas över flera noder, för high availability, används I kluster.
- Monitoring – Med hjälp av plugins kan RabbitMQ monitoreras enkelt genom att använda tex Prometheus+Grafana eller ELK stack.

En punkt till ZeroMQs fördel verkar dock vara lägre latens än RabbitMQ, detta I och med att Rabbit agerar som en egen full instans(tänk container eller VM) och på det tillkommer en del overhead kopplat till att driva denna instans, till skillnad från ZeroMQ som kan vara helt broker-less(i och med det även instans-less) och då undvika samma overhead.

I fallet Rabbit vs Zero behöver dock den ena inte utesluta den andra, det finns möjlighet att använda ZeroMQ som en klient till RabbitMQ.

Vid implementation av message ques I sin miljö bör man vara beredd på att anpassa koden till att kunna skicka meddelande via exempelvis ett messaging protokoll som AMQP. Vilken mjukvara man sedan använder är en annan fråga, Rabbit med dess persistens och monitorering eller lättviktaren ZeroMQ med dess något enklare implementation och lägre latens men samtidigt mindre features.