

F20 I7-BAC-01 Bachelorprojekt, 2020

Gruppe 2020F49, Aarhus Universitet

Andreas Schjødt Nielsen, studentID. 201602421
Anton Sakarias Rørbaek Sihm, studentID. 201504954
Oliver Monberg-Jensen, studentID. 201606654
Peter Marcus Hoveling, studentID 201508876

Dokumentation for Cluster Supported Parallel Libraries (CSPL)

Afleveringsdato: 27-05-2020
Vejleder: Jesper Rosholm Tørresø

Indholdsfortegnelse

1 Forord	6
1.1 Ordliste	6
2 Kravspecifikation	7
2.1 Aktør kontekst diagram	7
2.2 FURPS	7
2.3 Funktionelle krav - MoSCoW analyse	8
2.4 Ikke-funktionelle krav	9
2.5 User stories	10
2.6 Forretningsmodel	11
2.6.1 Hvad er produktet egentligt?	11
2.6.2 Hvem er den primære kunde?	11
2.6.3 Hvilke problem løser produktet for kunden?	11
2.6.4 Hvad er værdien for kunden?	12
2.6.5 Hvem er konkurrenterne?	12
2.6.6 Hvad gør vi for kunden, som konkurrenterne ikke gør?	12
2.6.7 Hvilke indtægter har vi?	12
2.6.8 Hvordan fastsættes priserne?	12
3 Metode og Process	12
3.1 Indledning	12
3.2 Gruppedannelse	13
3.3 Samarbejdsaftale	13
3.4 Projektledelse	14
3.5 Scrum	14
3.6 Udviklingsforløb	16
3.7 Risikovurdering	16
3.8 Møder	18
3.9 Planlægning	18
3.10 Arbejdsfordeling	19
3.11 Continuous Integration/DevOps	19
3.11.1 Opsætning af Byggeserver	19
3.11.2 Opsætning af pipelines	20
3.12 Udviklingsværktøjer og programmeringssprog	22
3.13 Coronavirus tilpasning	23
3.14 Mødeindkaldelser og mødereferater	23
4 Analyse	35
4.1 Cluster hardware	35
4.2 Jenkins vs. TeamCity	36
4.3 Master OS	37
4.3.1 Internetdeling og pålidelighed	37
4.3.2 Kubernetes og Docker på Linux vs. Windows	37
4.4 Raspberry Pi OS	38
4.5 Orkestrering	39
4.5.1 Kubernetes som orkestreringsværktøj	40
4.6 Docker	41
4.6.1 Docker Engine	41
4.6.2 Docker Swarm (vs Kubernetes)	41
4.6.3 Dockerfiler og images	41
4.7 Valg af programmeringssprog & frameworks	41
4.8 .NET Core i clusteret	42
4.9 Én vs fire pods pr. Raspberry Pi	42
4.10 Fjernsekvering af kode	43

4.10.1	Reflection	45
4.11	Distribuering af arbejde til workers	45
4.11.1	Undersøgelse af kubernetes loadbalancing	45
4.12	Firebase	47
5	Arkitektur	48
5.1	Introduktion	48
5.2	Systemsekvensdiagram	48
5.3	Kontekstdiagram	49
5.4	Containernediagram	50
5.5	Komponent- og kodediagrammer	51
5.5.1	CSPL NuGet-pakken	51
5.5.2	Master API	53
5.5.3	Worker API	57
5.5.4	CSPL.Common	59
5.5.5	Hjemmesiden	61
5.6	Hosting af applikationer	65
5.6.1	Hjemmesiden	65
5.6.2	CSPL NuGet-pakken	66
5.6.3	Master API	66
5.6.4	Worker API	66
5.7	Kommunikation mellem frontend og Master	66
5.7.1	Hardware oversigt	66
6	Design	67
6.1	Load balancing	67
6.1.1	Løsning 1 - Label switching	67
6.1.2	Løsning 2 - Custom load balancer	69
6.2	Forurening af pods i clusteret	71
6.3	Køstruktur	71
6.3.1	Hvornår får klienter tildelt workers?	71
6.3.2	Thread safety	72
6.3.3	Køstatus	72
6.3.4	Hvornår er det ens tur?	73
6.3.5	QueueUpdater	74
6.4	NuGet-pakke	75
6.5	Compile time properties problem	76
7	Implementering	77
7.1	Sekvensdiagrammer	77
7.1.1	NuGet-pakken	77
7.1.2	Master API	78
7.1.3	Worker API	81
7.2	Delt NuGet-pakke	81
7.3	DLL dependencies og filter	82
7.4	DLL-pakke	82
7.4.1	Sending af DLL-pakke	82
7.4.2	Eksekvering af DLL-pakke	83
7.5	Opsætning og problemer med Kubernetes cluster	84
7.5.1	Kubeadm	84
7.5.2	Pod network	84
7.5.3	Dynamiske IP-adresser	85
7.6	Deployment og konfigurering	85
7.7	Forurening af pods i clusteret	89
7.7.1	CustomAssemblyLoadContext	89
7.7.2	Oprydning af worker-pods	90
7.8	SSL	91

7.8.1	TLS tradeoff	92
7.8.2	TLS hjemmeside, master og workers	92
7.8.3	Opsætning	92
7.9	API-Key history view	96
7.10	Firebase abstraktion	99
7.11	Autorisation	100
7.12	Portforwarding	101
7.13	Fejlhåndtering	102
7.13.1	Generel fejlhåndtering	102
7.13.2	Klient-exceptions	103
8	Kubernetes installation and opsætningsguide	105
8.1	Prerequisite	105
8.2	Install OS	105
8.3	Enable SSH	106
8.4	Disable firewall	106
8.5	Edit host name	107
8.6	Configure cgroup boot options	107
8.7	Install updates	107
8.8	Permanently disable swap	108
8.9	Install Docker	108
8.10	Configure Docker daemon options	108
8.11	Setup Kubernetes repository	109
8.12	Install Kubernetes	109
8.13	Setup Kubernetes	109
8.14	Setup pod network	109
8.15	Join worker nodes to the Kubernetes cluster	110
8.16	Cluster SSH	110
9	Test	110
9.1	Introduktion	110
9.2	ManuelTest	111
9.2.1	Manuel test - Adminstrative produkt	111
9.2.2	Monkey tests	111
9.3	Feedback	111
9.4	Modultest	111
9.4.1	Master	112
9.4.2	ClusterSupportedParallelLibrary (NuGet-pakke)	113
9.4.3	CSPL.Common (NuGet-pakke)	114
9.5	Integrationtest	114
9.5.1	MonteCarlo	115
9.5.2	Brute force hash	115
9.5.3	Fejlhåndtering af klient-exceptions	116
9.6	Performancetest	116
9.6.1	Raspberry Pi 3B vs. Raspberry Pi 3B+	117
9.6.2	Docker overhead vs. normal proces	118
9.6.3	CSPL with Kubernetes vs. Docker container overhead	120
9.6.4	Opsummering - hosting overhead i CSPL systemtet	120
9.6.5	Desktop vs. Laptop vs. CSPL	121
9.6.6	Data sending overhead	124
9.6.7	HTTP vs. HTTPS	125
9.6.8	Billedekomprimering	125
9.6.9	MapReduce	126
9.6.10	Performance diskussion	126
9.7	Accepttestspezifikation	126
9.7.1	Download NuGet-pakke	126
9.7.2	Implementere NuGet-pakkens interface	127

9.7.3	Start et job fra Nuget-pakket	127
9.7.4	Cluster status	127
9.7.5	Opret en bruger på hjemmesiden	127
9.7.6	Login på hjemmesiden	127
9.7.7	Log ud af hjemmesiden	128
9.7.8	Ændrer password	128
9.7.9	Tilsending af nyt password	128
9.7.10	Slet account på hjemmesiden	128
9.7.11	Se dokumentationen	128
9.7.12	Se krav for brug af NuGet-pakken	129
9.7.13	Se eksempler for brug af NuGet-pakken	129
9.7.14	Se proof of work	129
9.7.15	Få genereret en API-nøgle	129
9.7.16	Se hvor mange kald der er tilbage på API-nøglen	129
9.7.17	Se tabel over jobs som er associeret med specifik API-nøgle	129
9.8	Accepttest	130
9.8.1	Funktionelle krav	130
9.8.2	Ikke-funktionelle krav	131
10	Resultater	131
10.1	Viden om cloud computing	131
10.2	Raspberry pi Cluster	131
10.3	Scrum	132
10.4	NuGet-pakke	132
10.5	Kommercialisering	132
10.6	Performance	132
10.7	Accepttest	132
10.8	DockerHub	132
11	Diskussion	133
11.1	Fejl og fremtidige forbedringer	133
11.1.1	CPU overlap	133
11.1.2	Manuel crashing af master	133
11.1.3	Manuel slowdown af master	133
11.1.4	Ulovigheder/sikkerhed	133
11.1.5	Datasæt sendes som JSON	134
11.1.6	Streaming af klient-data	134
11.1.7	DLLPackage på compiletime	134
11.1.8	DLL Cache	134
11.1.9	Stateful master	134
12	Logbog	135
13	Litteraturliste	141
	Hjemmesider	141

1 Forord

Denne rapport dækker over bachelorprojektet Cluster Supported Parallel Libraries som er udarbejdet af diplomingeniørstuderende ved uddannelsesinstitutionen AU Engineering, Aarhus Universitet. Projektgruppen består af fire studerende fra Software Teknologi-uddannelsen. Projektgruppens vejleder er lektor ved AU Engineering, Aarhus Universitet, Jesper Rosholt Tørresø.

Ud fra gruppens vurdering anses produktet som værende unikt, i og med at det ikke har været muligt at finde eksempler på lignende software, som tilbyder parallelisering i cloud via en NuGet-pakke.

Specielt tak til **kamstrup** for at have sponsoreret 15 Raspberry Pi 3B til projektet.

1.1 Ordliste

Følgende underafsnit er til læserens gavn for kendskab til forkortelser og andre betydninger af ord, der anvendes igennem rapporten.

- **Job(s)**: bruges kode som bliver eksekveret på clusteret
- **SPA**: Single page application
- **HW**: Hardware
- **PSU**: Power Supply Unit
- **SSD**: Solid State Drive
- **LAN**: Local Area Network
- **Node**: Én PC i et cluster, f.eks. en Raspberry Pi
- **Pod**: En pod hoster én eller flere container som yderligere hoster én eller flere processer/programmer
- **Master API**: Det program som håndtere klient-kald mm.
- **Worker API**: Det program som processere en klients kode
- **Worker-pod**: En pod i Kubernetes-clusteret som hoster et Worker API
- **SW**: Software
- **DB**: Database
- **SSL**: Secure Socket Layer
- **CSPL**: Cluster Supported Parallel Libraries
- **POC**: Proof of concept
- **X509**: Certificate standard
- **CI**: Continuous Integration
- **OS**: Operativ system
- **CLI**: Command line interface
- **SSL**: Secure Socket Layer
- **TLS**: Transport Layer Security

2 Kravspecifikation

2.1 Aktør kontekst diagram

Aktør kontekst diagrammet, som set på figur 1, viser de primære og sekundære aktører. Den primære aktør er brugeren, altså udvikleren, som anvender clusterets biblioteker eller bruger hjemmesiden til monitorering, mens de sekundære aktører er databasen, NuGet-pakke repository og hosting.

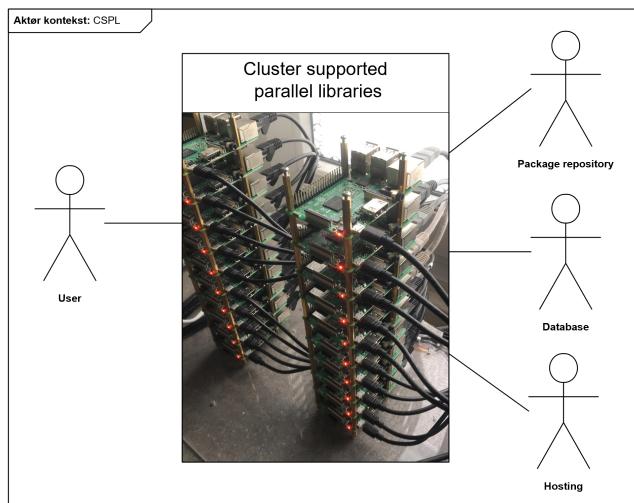


Figure 1: Aktør kontekst diagram

2.2 FURPS

De funktionelle og ikke-funktionelle krav er udarbejdet ud fra FURPS som klassificerer de krav der er opstillet til systemet. De funktionelle krav er alle de funktionaliteter som systemet kan eller skal bestå af. Hvorimod de ikke-funktionelle krav er de aspekter af systemet som kan måles og testes. F'et i FURPS står for functional og de sidste fire bogstaver er henholdsvis usability, reliability, performance og supportability.

Nedenstående tabel viser gruppens prioritering blandt de forskellige FURPS attributter på en skala fra 1-5.

Table 1: Prioritering af FURPS

Prioritering af FURPS	
Functional	5
Usability	3
Reliability	4
Performance	5
Supportability	2

I tabel 1 ses de forskellige vægtnings prioriteter som gruppen har til systemet.

Usability er vægtet 3 da systemet nødvendigvis ikke skal være nemt at bruge, da det kræver en logisk og programmatisk viden inden for software for at kunne interagere med systemet. Programører vil kunne interagere med systemet via et 'fluent interface' som vil have en tilhørende dokumentation.

Reliability er vægtet 4 da systemet skal have høj stabilitet samt en høj uptime. Grundet denne høje prioritet, hostes de forskellige software komponenter i et cluster, og igennem allerede etableret cluster software såsom Kubernetes og Service Fabrik, vil systemet kunne sikres en lav downtime, failureovers, mulighed for opgraderings senearier og scalability.

Performance er vægtet 5 og er gruppens højeste prioritet, da dette bygger på hele ideen omkring projektet.

Når brugeren interagere med systemet, skal systemet kunne distribuere arbejde til forskellige Raspberry Pies og gøre brug er alle cpu-kerner, for at processere arbejdet med højeste performance.

Supportability er vægtet 4, da systemet skal være muligt at administrere hvis man har visse rettigheder. Det skal være muligt at monitorere systemet, her i blandt se load på forskellige Raspberry Pies, konfigurere hvilke services der kører i clusteret samt hvor mange klonede instanser skal kører. Derudover skal det være muligt at teste systemet for forskellige processerings-arbejde samt se tids optimerings forskellen fra en normal 'user' pc i forhold til en cluster processering.

2.3 Funktionelle krav - MoSCoW analyse

Følgende er udkast til funktionelle og ikke-funktionelle krav til systemet.

Must have:

- **Cluster**

- Clusteret skal parallelisere arbejde ved brug af flere nodes.
- Clusteret skal kunne benyttes vha. et bibliotek i en NuGet-pakke.
- NuGet-pakken skal indeholde funktionalitet der gør det muligt at parallelisere arbejde på clusteret.
- NuGet-pakken skal hostes på en NuGet-server der er tilgængelig for brugeren.
- Det skal kun være muligt at benytte clusteret med en gyldig API-nøgle.
- Systemet skal hostes med et container-orkestrerings-system som Service Fabric eller Kubernetes.

- **Hjemmeside**

- Hjemmesiden skal bestå af henholdsvis en dokumentationsside, API-nøgle data side, login og create account samt proof-of-work side.
- Det skal være muligt for en bruger at oprette en account på hjemmesiden.
- Det skal være muligt at logge ind på hjemmesiden.
- Det skal være muligt at kunne få tilsendt glemt password via email.
- Det skal være muligt for brugere at læse dokumentation af bibliotekerne på hjemmesiden.
- Det skal være muligt at kunne få genereret en API nøgle på hjemmesiden.
- Hjemmesiden skal være online selvom clusteret ikke er.
- Brugerinformation skal gemmes i en database.
- Det skal være muligt at kunne ændre sit password.
- Sensitivt brugerdata skal hashes/krypteres.

Should have:

- **Cluster**

- Kommunikation mellem master og worker Pi's bør være krypteres.
- Al kommunikation fra client til clusteret bør krypteres.
- Systemet bør kunne auto deployeres vha. scripts.
- Det bør være muligt at kunne se status på clusteret selv under strøm svigt og downtime.
- Det bør være muligt at kunne eksekvere selv definerende parallelle jobs via et generisk fluent API.

- **Hjemmeside**

- En bruger bør kunne se brugerspecifik analyse/statistik på hjemmesiden.
- Systemet bør have en fælles log, hvor en alarm monitor service monitorere alarmer.
- Det bør være muligt at kunne få tilsendt et nyt password.

Could have:

- **Cluster**

- Det kunne være muligt for brugeren at dele predefinerede jobs med hinanden.
- Det kunne være muligt at deployere systemet ud i clouden.
- Det kunne være muligt at downscale systemet.
- Det kunne være muligt at upscale systemet.
- Det kunne være muligt at opgradere en microservice.

- **Hjemmeside**

- Det kunne være muligt, at logge ind med ens egen Facebook eller Google account.
- Det kunne være muligt for bruger at se et overview over antal API-kald.
- Det kunne være muligt at se proof of work på hjemmesiden.
- Det kunne være muligt for brugeren at se information om downtimes/uptimes på hjemmesiden.
- Det kunne være muligt at dele forskellige generiske algoritmer på dokumentations siden.

Won't have:

- **Cluster**

- Clusteret vil ikke have bedre HW end Raspberry Pi nodes.
- Biblioteket vil ikke understøtte flere end ét programmerings sprog.

2.4 Ikke-funktionelle krav

Usability:

- Hver Raspberry Pi skal indikere dens arbejde via en LED.
- Hjemmesiden skal være intuitiv at bruge.
- Biblioteket (NuGet-pakken) skal være intuitiv at bruge ved hjælp fra dokumentationen på hjemmesiden.

Reliability:

- Hvis en pod crasher (unhandled exception) skal den være tilgængelig i clusteret igen inden for 30 sekunder.
- Hvis en node dør midt i at processere noget arbejde, skal de andre noder kunne kompensere.
- Clusteret skal minimum have en uptime på 95% over en uge.
- Under opdatering af pods skal der være 100% uptime.

Performance:

- Ved brug af biblioteket skal brugere kunne se et performance gain ved parallelisering.
- Clusteret skal bestå af minimum 5 noder.
- Det skal være muligt at skalere systemet med flere nodes

Supportability:

- Hvis en service på en node crasher allokeres fejlen i en ekstern fil.
- Det bør være muligt at kontakte administratorer igennem hjemmesiden.

2.5 User stories

På user story diagrammet, som ses på figur 2, ses de 6 overordnede user stories kategorier som er opstillet. Det ses her hvilke user stories den primære aktører interagere med, samt hvilke user stories de sekundære aktører interagere med.

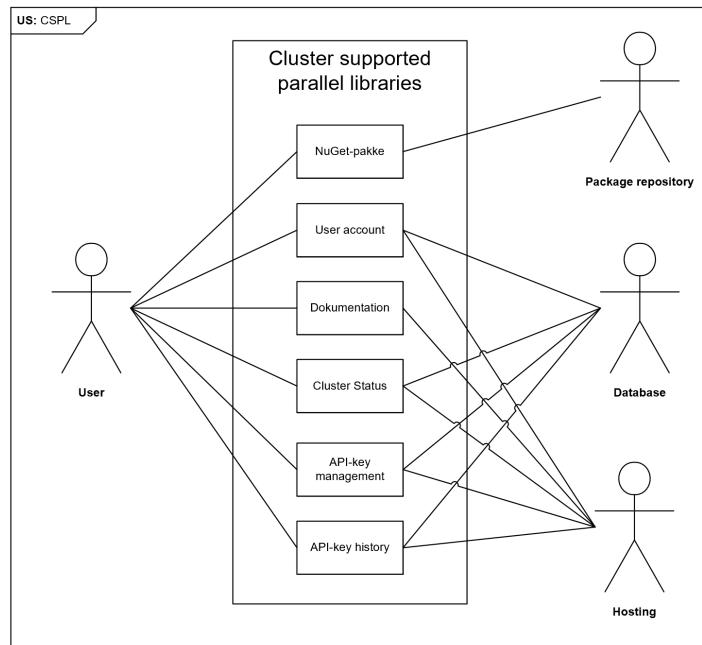


Figure 2: User stories diagram

En user story er et alternativ til en use case. Det er en kort og præcis beskrivelse af en hændelse hvor historien fortælles fra perspektivet af en person som ønsker en funktion i et system. Typisk følger user stories følgende skabelon: "Som <en type af bruger> ønsker jeg <et ønske i systemet> så jeg <et bestemt formål med ønsket>."

Nedenstående er en liste af de user stories, som er udarbejdet for Cluster Supported Parallel Libraries.

NuGet-pakke

- Som bruger kan jeg downloade NuGet-pakken så jeg kan benytte den i mit C# .NET Core projekt.
- Som bruger kan jeg implementere et generisk interface så min kode kan eksekveres parallelt på clusteret via NuGet-pakken.
- Som bruger kan jeg starte et job med NuGet-pakken.

Cluster status

- Som bruger kan jeg se status for clusteret, om det er online eller offline.
- Som bruger kan jeg se hvor mange pods der er i clusteret.

User account

- Som bruger kan jeg oprette en account med e-mail og password, så jeg kan logge ind på hjemmesiden.
- Som bruger kan jeg slette min account, så den er ude af systemet.
- Som bruger kan jeg logge ind, så jeg kan bruge hjemmesidens funktionaliteter.
- Som bruger kan jeg logge ud af hjemmesiden.
- Som bruger kan jeg kunne ændre mit password.
- Som bruger skal jeg kunne få tilsendt et nyt password, hvis jeg har glemt det.

Dokumentation

- Som bruger kan jeg se dokumentation for, hvordan NuGet-pakken skal bruges.
- Som bruger kan jeg se de krav, der er for brug af NuGet-pakken.
- Som bruger kan jeg se et eksempel på, at parallelisering via clusteret giver ekstra performance.
- Som bruger kan jeg se "proof of work", som viser grafer for performance-gain med forskellige jobs.

API key management

- Som bruger kan jeg se hvor mange kald, der er tilbage på min API-nøgle.
- Som bruger kan jeg få genereret en API-nøgle på hjemmesiden, så jeg kan bruge NuGet-pakken.

API-key history

- Som bruger kan jeg se en tabel med de jobs, som er eksekveret på clusteret for min nuværende API-nøgle.
- Som bruger kan jeg se hvor hurtigt mit arbejde bliver eksekveret på clusteret, så jeg kan sammenligne eksekveringstider.
- Som bruger kan jeg se hvor mange worker-pods, der er blevet brugt til at eksekvere et stykke arbejde.
- Som bruger kan jeg se status (kø, fejl og succes mm.) på de jobs, som jeg har anmodet om at få eksekveret på clusteret.
- Som bruger kan jeg se et selvdefineret label på de jobs, som jeg har kørende på clusteret.
- Som bruger kan jeg se start- og slutdato på de jobs, som jeg har kørende på clusteret.
- Data omkring jobs skal være i realtid og dermed visuelt for brugeren i mens jobbet kører.

2.6 Forretningsmodel

Følgende afsnit omhandler produktets forretningsmodel, som indebærer spørgsmål som, hvem den primære kunde er, hvad produktet egentlig er, hvem der er konkurrenter mm. Forretningsmodellen er med til at give projektet et mål og bruges til at skabe fokus og enighed i gruppen. Derudover skaber den et billede af, hvad produktet løser for en eventuel kunde og eller investor. Følgende er en gennemgang af projektets forretningsmodel.

2.6.1 Hvad er produktet egentligt?

Produktet består af en NuGet pakke hvortil kunden skal købe en Api-nøgle, for at få lov til at lave Http kald på clusterets service. NuGet pakken tilbyder parallelisering på multiple distribuerede maskiner, hvor kunden ikke behøver at hoste et cluster af computere, men istedet blot bruge NuGet pakken.

I en fremtidig løsning kan kunden selv købe hardware og hoste clusteret, hvor det software der er skrevet i forbindelse med dette projekt, kan være det som kunden køber.

2.6.2 Hvem er den primære kunde?

Den primære kunde er virksomheder, organisationer eller videnskabelige fakulteter som ikke har det hardware de skal bruge til eksempelvis stor dataprocessering. Til forskning i ny medicin, matematiske beregning eller lignende, vil målgrupper som ønsker hurtige beregninger, her have mulighed for at gøre brug af clusterets hurtige parallelle egenskaber.

2.6.3 Hvilke problem løser produktet for kunden?

Produktet løser problemet med ikke at have hardwaren og softwaren til at parallelisere en algoritme på multiple maskiner. Problemets løsning for kunden er problemet med eksempelvis ikke at kunne processere data hurtigt nok. Kunden behøver ikke at investere penge på hardwaren, udviklingen af softwaren og vedligeholdelse af systemet.

2.6.4 Hvad er værdien for kunden?

Kunden kan både købe produktet som en enkelt gangs ydelse eller flere gangs ydelse. Kunden binder sig altså ikke ved at investere i sit eget system. Kunden betaler kun for det arbejde der bliver eksekveret i clusteret - det er ikke et abonnement.

2.6.5 Hvem er konkurrenterne?

Cloud services hvor kunder selv kan købe virtuelle maskiner/hardware og deploy deres eget system til dataprocessering. Kunden kan selv implementere et system der kan udnytte parallelisme til dataprocessering. Vores løsning skal være brugervenligt og effektivt nok, til at det kan betale sig for kunden.

2.6.6 Hvad gør vi for kunden, som konkurrenterne ikke gør?

Vi tilbyder parallelisering på multiple distribuerede maskiner gennem en NuGet-pakke. Altså softwaren bag paralleliseringen.

2.6.7 Hvilke indtægter har vi?

Salg af API nøgle/kald til clusteret.

2.6.8 Hvordan fastsættes priserne?

En mulighed for fastsættelse af pris er en kombination af antal kald på clusteret og eksekveringstid.

3 Metode og Process

I følgende afsnit gives en beskrivelse af gruppens metode og arbejdsproces. Afsnittet kommer omkring emner som samarbejdskontrakt, agil projektstyring (scrum), udviklingsforløb, gruppeledelse, proof of work arbejde, planlægning, risiko vurderinger og tidsplan.

3.1 Indledning

En proces betyder et forløb sammensat af handlinger som munder ud i forandring eller udvikling [46]. Handlinger i dette bachelor projekt kommer i form af aftaler, udvikling, initiativtag og beslutninger. Processerne og handlingerne ender med at blive til et produkt og det er rejsen imod det produkt som bliver beskrevet i de kommende afsnit.

Der er i løbet af dette projekt lagt stor vægt på, at processen forløber hensigtsmæssigt i forhold til gruppens måde at arbejde på. Dette betyder, at der bliver lavet revurderinger af måden gruppen gør tingene på og derefter fokuseret på tilpasning af nye processer eller tiltag.

Opgaver	Anton	Peter	Oliver	Andreas
Kravsspecifikation	x	x	x	x
Metode og proces	x	x	x	X
Analyse	x	x	x	x
Arkitektur	x	x	x	x
Interface i NuGet-pakke	X		x	X
Load balancing	X	x	x	
Kø-struktur	X		x	x
DLL-dependencies	x		X	x
Fjerneksekvering af kode	X		X	X
Forurening af pods	x		X	
Fejlhåndtering		x	X	
Kubernetes	x	X	X	x
Hjemmeside		X		X
TLS/SSL	x	X	x	
Opsætning af build server	x	X	x	x
Integrationstests	x	x	X	x
Unittests	X	x	X	
Performancetests	x	x	x	x
Hardware opsætning		X	x	
Port forwarding		X	X	
API-Key validering		X		
Firebase		X		X
Scrum master				X
Kontakt til vejleder			X	
Sekretær (referent)				x

Table 2: Arbejdsfordeling for projektet.

3.2 Gruppeddannelsel

Gruppen på fire personer er sammensat på baggrund, af et ønske for at arbejde sammen. Gruppen har ofte arbejdet sammen i udformning af opgaver og vidste derfor gruppen kunne nå et godt samarbejde. Gruppen er sammensat af Software Teknologi studerende fra ingeniøruddannelsen på Katrinebjerg.

3.3 Samarbejdsaftale

I starten af projektet bliver der udarbejdet en samarbejdskontrakt fælles i teamet som har til ansvar at opstille nogle normer og regler for hvordan der eksempelvis bliver truffet beslutninger. Punkterne forventes overholdt, men i særlig tilfælde er kontrakten der. Kontrakten er godkendt af alle gruppens medlemmer.

Hvor ofte holdes gruppemøder?

- Daily scrum møder holdes tirsdag-torsdag om morgen klokken 8.30. Møder med vejleder holdes efter behov men ca. en gang hver eller hver anden uge.

Hvordan indkaldes til møderne og af hvem?

- Mødeleder(Oliver)-Indkaldelse sker via mail med en vedlagt dagsorden.

Hvem udformer dagsorden?

-Møde leder har ansvar men snakker med gruppen.

Med hvor kort varsel kan et medlem melde afbud til et møde?

- Der kan være situationer hvor man må melde afbud med meget kort varsel, det kan vi ikke kontrollere. Men det forventes at man melder afbud, hurtigst muligt når man ved, at man ikke kan deltage.

Hvad er konsekvensen, hvis et medlem udebliver fra et møde?

- Deltager man ikke i et møde, kan man heller ikke være med til at træffe beslutninger. Gruppens evne til at træffe en beslutning skal ikke hindres af en enkelt persons manglende tilstede værelse. - Man er forpligtet til at holde sig selv opdateret. Læs mødeindkalder og referater. Spørgruppens andre medlemmer hvis der er spørgsmål eller tvivl. Manglende handling eller utilfredshed over beslutninger mens man ikke var der er ikke acceptable. -Ved gentagne/konsekvente udeblivelser vil Gruppelederen og/eller gruppen sammen med personen

tage et møde for at klarlægge årsagen, og afhjælpe det om muligt. I tilfælde hvor det ikke er muligt for grupplederen/gruppen må vejlederen inddrages.

Hvem tager referat af møderne?

- Referenten Peter.

Hvem udsender referatet?

- Referenten fra det pågældende møde. Lægges i bachelor dokumentationen i Latex dokumentet.

Hvordan ledes gruppen?

- Gruppen træffer beslutninger på demokratisk vis. - I tilfælde hvor argumenterne på begge sider er lige gode og det er umuligt at komme til enighed, har Andreas det sidste ord.

Hvem er scrum master?

- Andreas er scrum master hvilket betyder at han faciliteter scrum processen.

Hvordan afgøres, om et problem har en karakter, så vejlederen bør informeres?

- Ved problemer som gruppen har med et individ eller anden gruppe: Ved konsensus-Ved problemer hvor individet har et problem med gruppen: man bør tage det op med gruppen, men hvis man mener der er nødvendigt må det være ens egen beslutning at gå til en vejleder.

Hvad er gruppens ambitionsniveau, vil vi blot netop bestå, eller går vi efter 12 tallet?

- Gruppen som helhed har et højt ambitionsniveau, vi vil gerne have et gennemarbejdet produkt og rapport og levere vores bedste. Hele gruppen ønsker at yde deres bedste for at opnå et godt resultat.

Hvad er konsekvensen for et medlem, der ikke overholder samarbejdskontrakten?

- Det formodes at alle har til hensigt at overholde aftalen idet at den i sin endelige form er blevet godkendt af hvert medlem. - Eventuelle mislighold forventes ikke at være intentionelle. Skulle det imod forventning ske, taler hele gruppen sammen om tilfældet. Er det ikke muligt at tale med medlemmet som har brudt aftalen, taler gruppen sammen uden ham og beslutter samtidigt konsekvensen, evt. i samråd med vejleder.

I hvilke situationer vil vi inddrage vejlederen i for eksempel konfliktløsning?

- Det forventes ikke at der opstår en situation som gruppen ikke selv kan håndtere. Det vil dermed være i situationer hvor vi, på trods af kommunikation ikke kan komme overens. Men det forventes ikke idet at gruppens medlemmer generelt er på samme bølgelængde og har de samme generelle ambitioner og ønsker for projektet. Samarbejds kontrakten kan reevalueres til møderne.

3.4 Projektledelse

Hvad angår projektledelse blev det bestemt fra starten i forbindelse med samarbejdsaftalen at gruppens beslutningstagen styres kollektivt og demokratisk. I forbindelse med uoverensstemmelser i beslutningsprocessen vinde der det bedste argument og i tilfælde af fuldstændig uenighed har gruppens scrum master det sidste ord.

Der er valgt en kommunikationsleder som bindeled imellem gruppen og vejleder. Denne person har også til ansvar, at udarbejde dagsordenen samt at sende den til gruppens øvrige medlemmer og vejleder inden vejledermøder.

3.5 Scrum

Gennem hele projektforløbet har gruppen brugt udviklingsmetoden scrum som processstyringsværktøj. Scrum er en agil arbejds metode som er velegnet til den iterative processtyrings-form, hvor man i stedet for at have en bestemt sekvens af arbejdsopgaver, kører iterationer over en sekvens af arbejdsopgaver. Dette muliggør, at man nemmere kan justere og tilpasse sig ændringer i projektet og dets krav.

Gruppen har tilegnet sig de metoder og processer fra scrum, som giver mest mening for styringen af lige netop dette projekt. Heriblandt er begreberne sprints, sprint planning, backlog, daily scrum, retrospektiv og scrum master. En scrum master er en person, som styrer scrum-aktiviteterne og forsøger at holde styr på at gruppens medlemmer overholder tidsplanen i forhold til at udarbejde sprints. Et sprint er en udviklingsfase på et antal uger, helst ikke for mange. I dette projekt består et sprint som udgangspunkt af to uger men ændres efter behov. Daily scrum afholdes hver morgen, hvor gruppens medlemmer fortæller hvad de lavede i går og hvad de skal lave i dag. Her fortælles også om eventuelle forhindringer eller andre foretagende som skal diskuteres på gruppen. Retrospektiv holdes ved afslutningen af hver sprint, og er et tilbagekig på det afsluttede sprint. Her diskuteres og snakkes der om hvad der kunne være gjort bedre. Giver de valgte processer mening for projektet, om der skal medtages nye eller om der skal laves noget om.

I starten af hver sprint afholdes sprint-planning, hvor gruppen samles og bliver enige om hvilke prioriteter eller "sprint goals" det kommende sprint skal have. Når gruppen er enige, designes sprintet og der bliver trukket opgaver fra backloggen ind i det aktive sprint på udviklingsværktøjet Jira. Backloggen er en samling af opgaver som skal laves til projektet. I dette forløb er aftalen at gruppemedlemmer smider opgaver ind i backloggen hver gang man støder på noget som skal laves. Opgaver vælges eller fravælges så til sprint planning.

Til at styre scrum, har gruppen benyttet sig af udviklingsværktøjet Jira. Jira understøtter udarbejdelse af backlog og har et dashboard til at holde styr på det aktive sprint. Sprintboardet eller dashboardet ses på figur 3. Opgaver bliver tidsestimeret og prioriteret inden de bliver trukket ind i sprintet.

Grunden til at gruppen har valgt at tidsestimere opgaver, er ikke nødvendigvis for at vide, lige præcis hvor lang tid en opgave tager. De fleste opgaver er nogle som bare skulle laves, uanset hvor lang tid det måtte tage. Den primære grund er for at være sikker på, at der er enighed om hvad en opgave indebærer. Hvis nogle fra gruppen mener, at opgaven tager 2 timer, imens andre synes den skal tage 20, så er der nok en misforståelse i opgavens størrelse eller indhold. Til tidsestimering er der benyttet story points from for timer. Det startede med at være timer, men efter et par sprints prøvede gruppen at bruge story points og har benyttet story points siden.

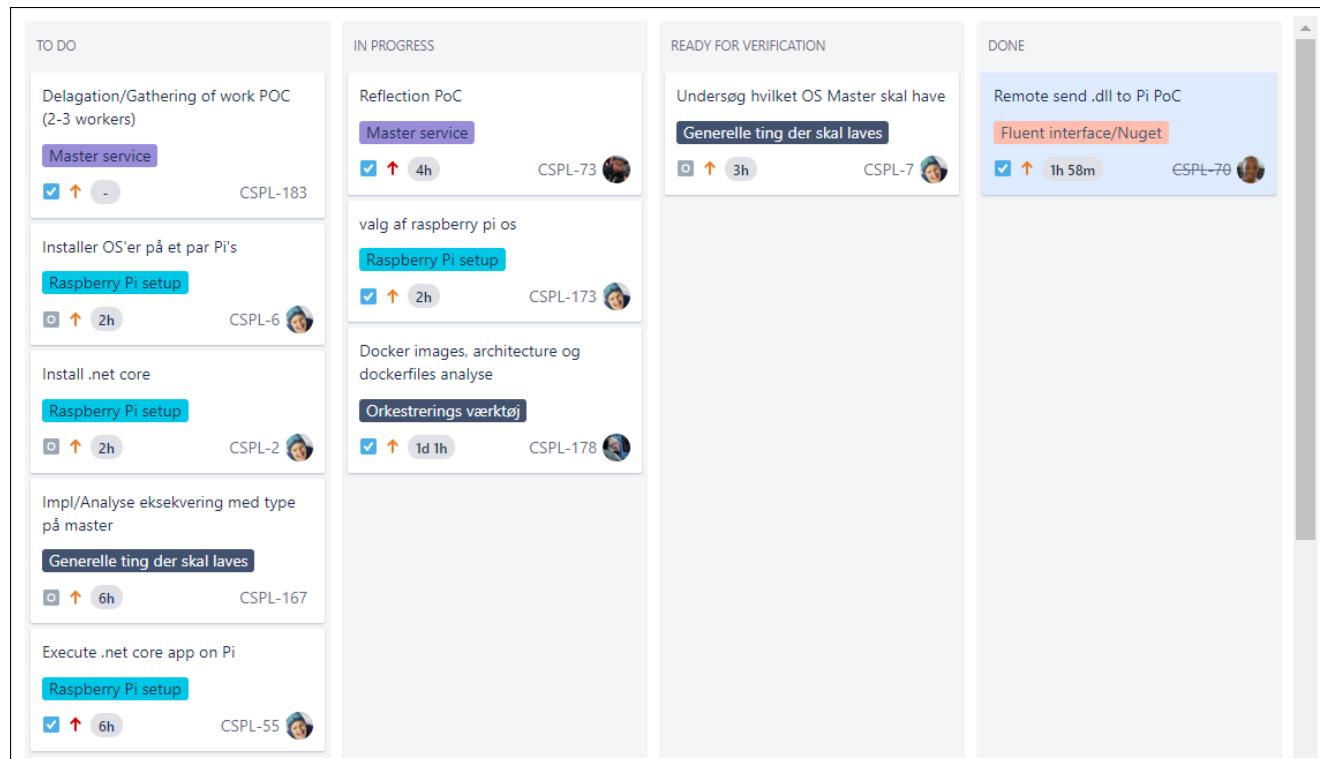


Figure 3: Jira scrumboard, aktivt sprint.

Gruppen har defineret 4 "swim lanes" hvor "TO DO" er opgaver i sprintet som skal laves. "IN PROGRESS" er opgaver som er ved at blive lavet. "READY FOR VERIFICATION" er opgaver som skal testes, inspiceres eller verificeres af andre gruppemedlemmer. "DONE" er når opgaver er helt færdige.

Hver opgave som laves i et sprint, skal verificeres af et andet gruppemedlem. Hvis der er kode indblandet skal koden inspiceres og hvis det handler om rapport skrivning, skal det skrevne materiale gennemgås. Dette er ikke en proces som udtrykker at gruppemedlemmerne ikke stoler på hinandens kompetencer, men snarere et kvalitetstjek og "knowledge sharings" element. Det er en god måde at blive inkluderet i hvad de andre har lavet, samtidig med at inspektøren kan komme med konstruktiv kritik eller forbedringer.

3.6 Udviklingsforløb

Udviklingsforløbet har været styret af Scrum og har derfor været en iterativ proces. Det betyder at krav, arkitektur, design og implementering af forskellige komponenter har været igennem flere iterationer, og dermed er blevet revurderet og genovervejet over flere gange. Udviklingsforløbet kan på trods af den iterative tilgang inddeltes i nogle faser.

Den første fase af projektforløbet bestod af, at udforme et udkast til en problemformulering samt at specificere systemets funktionelle og ikke-funktionelle krav. Der opstod et spørgsmål, om hvorvidt det ønskede produkt kunne blive en realitet og projektet blev derfor midlertidigt forvandlet til et undersøgelseprojekt. Det var med andre ord nødvendigt, at finde ud af hvorvidt teknologien i kombination med gruppens viden, kunne fungere før udarbejdelsen af et reelt produkt gik i gang.

Den efterfølgende tid gik med at undersøge hvilke teknologier, algoritmer og koncepter skulle bruges for at få projektet på benene. Der blev derefter lavet en risikovurdering af koncepterne som kan ses i afsnit 3.7, for at lave en prioritering samt vurdering af hvilke features skulle prioriteres inden for den skemalagte tid. Efter risikovurdering blev der udarbejdet POCs (proof of concept) på de forskellige koncepter for at kunne bevise at de kunne lade sig gøre. Eksempler på disse POCs er "remote eksekvering af Dll'er samt reflektion" og "opsætning af Kubernetes for et Raspberry Pi cluster" som nogle af de mest kritiske.

Da alle POCs var på plads, betød det, at det rent faktisk var muligt, at lave det system, som var på dagsordenen. Derfor kunne gruppen igen forvandle projektet til et produkt frem for en undersøgelse. Det betød, at der skulle udarbejdes en forretningsmodel, som kan ses i afsnit 2.6, for at præcisere emner som hvem kunden skulle være, hvad produktet egentligt bestod af mm. Derefter blev der lavet et udkast til arkitekturen for systemet. Gruppen valgte i den forbindelse, at benytte sig af C4-modellen[4] til udarbejdelsen af arkitekturen. Her blev dannet et overblik over, hvordan systemet skulle se ud i store træk, eksempelvis at systemet kunne bære fordel af at benytte Kubernetes, samt hvilke former for applikationer der skulle implementeres og hvor de skulle hostes.

Da arkitekturens første udkast var færdigt, kunne implementerings-fasen starte. I og med at projektet kører iterativt bliver der lavet om i både arkitektur og design under implementerings-fasen, i forbindelse med at gruppen finder bedre løsninger. To af gruppens medlemmer gik i gang med implementeringen af master og worker applikationerne som skulle hostes i clusteret. De to andre gik i gang med opsætningen af clusteret i Kubernetes. Nu var alle de "farligste" opgaver i gang. Opsætning af clusteret voldte mange problemer og tog lang tid. Det passede med gruppens forudsigelse i risikovurdering. Da gruppen fik nogenlunde styr på de førnævnte opgaver, begyndte arbejdet på hjemmesiden.

Efter implementeringen havde gruppen planlagt at bruge noget tid på at lave performance test af systemet, for at se om det udarbejdede produkten kunne det, som var meningen. Der blev også kørt en accepttest som kan ses i afsnit 9.8

3.7 Risikovurdering

For at danne et overblik over de features, koncepter og teknologier som potentielt kan risikere projektets helhed, har gruppen udarbejdet en risikomatrix. Matrixen er lavet med et tredje parts værktøj, som benyttes via Excel[18]. Matrixens kolonner består af henholdsvis; hændelse, sandsynlighed, konsekvens og risiko. Risikoen er beregnet ud fra graden af sandsynlighed og konsekvens. Risiko skalaen går fra 1-20 og den røde farve betragtes som meget høj risiko, gul som moderat risiko og grøn som acceptabel eller ingen risiko. Sandsynligheden og konsekvensen er bestemt ud fra en kvalificeret vurdering, hvor der er inkluderet information fra internettet samt gruppens personlige erfaringer. Første udkast til risikomatrix ses på figur 4.

Risikoanalyse: Cluster supported parallel libraries		Risknon fra Risikoanalyser.dk			Risiko
Nr	Hændelse	Sandsynlighed	Konsekvens		
slet 20	Implementering af job queue på master fejler	Høj	Medium	12	
slet 19	Implementering af NuGet-pakke og NuGet server fejler	Lille	Meget alvorlig	10	
slet 18	Sending af .dll over netværket fejler	Meget lille	Meget alvorlig	5	
slet 17	Opsætning af Raspberry Pi fejler	Lille	Meget alvorlig	10	
slet 16	Opsætning af build server fejler	Medium	Medium	9	
slet 15	Eksekvering af specifik kode fra .dll fejler	Medium	Meget alvorlig	15	
slet 14	Sikker kommunikation og beskyttelse af data fejler	Medium	Medium	9	
slet 13	Auto deployment af cluster fejler	Høj	Lille	8	
slet 12	Eksekvering af .net core applikationer på Pi's fejler	Lille	Meget alvorlig	10	
slet 11	Kommunikation mellem master og hjemmeside fejler	Medium	Medium	9	
slet 10	Opsætning af hjemmeside fejler	Meget lille	Stor	4	
slet 9	Visning af brugerrelateret statistik/analyse på hjemmeside fejler	Medium	Lille	6	
slet 8	Visning af dokumentation/brugervejledning på hjemmesiden fejler	Lille	Medium	6	
slet 7	Implementering af master service fejler	Medium	Stor	12	
slet 6	Implementering af worker service fejler	Medium	Stor	12	
slet 5	Kommunikation mellem master og bruger fejler	Lille	Meget alvorlig	10	
slet 4	Kommunikation mellem worker Pi's og master fejler	Lille	Meget alvorlig	10	
slet 3	Opsætning af Kubernetes fejler	Medium	Stor	12	
slet 2	Implementering af generisk library til parallelisering af brugerdefineret arbejde fejler	Høj	Lille	8	
slet 1	Implementering af et hardcoded eksempel på et paralleliserbart arbejde fejler	Lille	Meget alvorlig	10	

Figure 4: Risiko matrix

Efter ca. halvanden måneds arbejde, blev risikomatrixen revurderet af gruppen. Denne revurdering ses på figur 5. Her ses det at flere af de opgaver med høj risiko er forsvundet, da de blev udryddet under implementeringen af POCs. Denne risikomatrix var den sidste revurdering, men man kunne med fordel revurdere igen, for både at tilfredsstille udviklingsteamet og kunde.

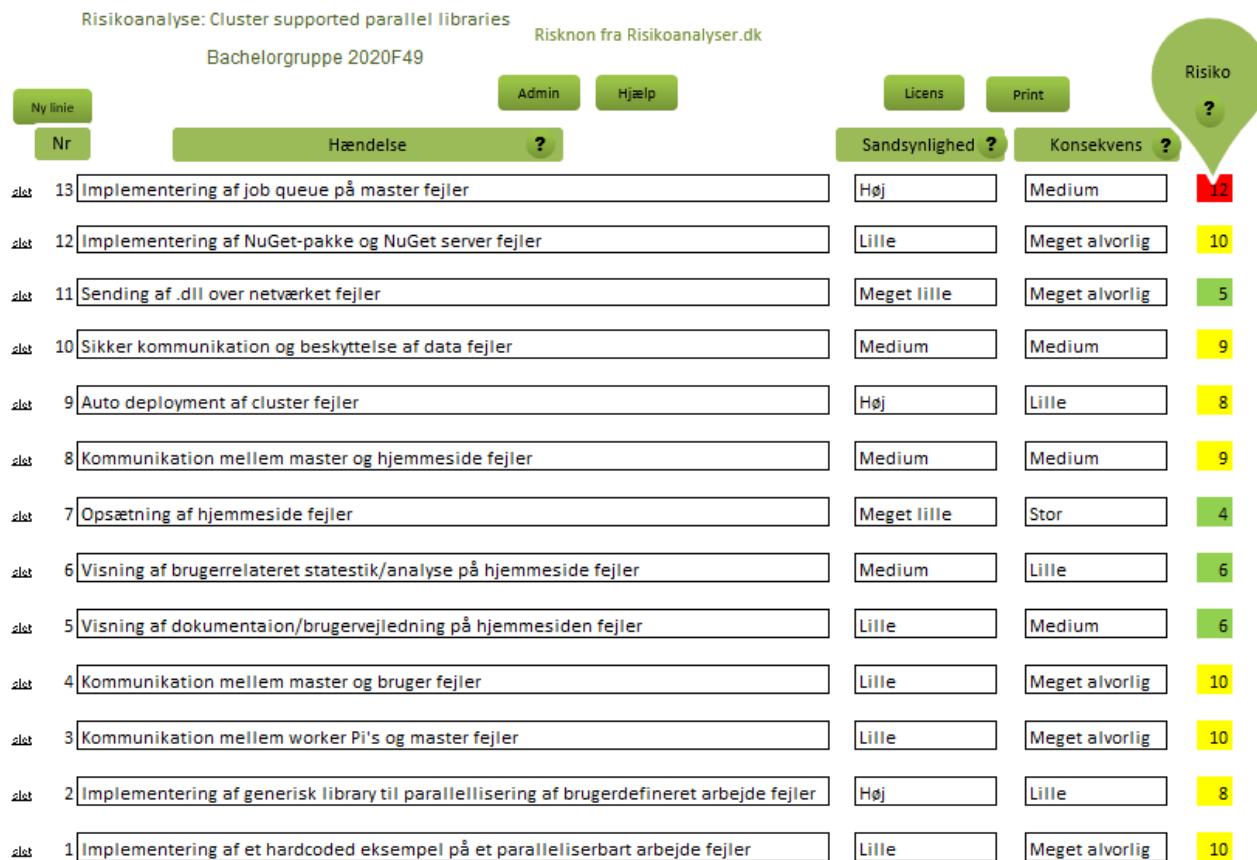


Figure 5: Risiko matrix, revurdering.

3.8 Møder

Gruppen har prioriteret at holde daily scrum møder hver dag for at fortælle hvad hver i sær har af planer for den kommende dag og hvad de lavede dagen før. Derudover er det aftalt med vejleder at mødes en gang om ugen, medmindre der er behov for mere eller mindre. Vejledermødernes indhold går primært ud på, at give et resume af hvordan det går og få svar på de spørgsmål gruppen måtte have i forbindelse med udarbejdelsen af produktet eller håndtering af processen.

Der bliver udsendt dagsorden af gruppens mødeleeder, til vejleder og de øvrige gruppemedlemmer inden hver møde, så alle har en chance for at forberede sig til mødet, så udbyttet kan blive størst muligt. Til hver møde bliver der taget referat som bliver lagt op på et fælles Google drive.

3.9 Planlægning

Selvom gruppen har gjort brug af den agile tilgang til arbejdsprocessen, er der alligevel udarbejdet en tidsplan. Tidsplanen er brugt som et estimat på hvornår det første udkast til de forskellige emner skulle være klar eller udarbejdet. Det betyder med andre ord, at der ikke kun er arbejdet med arkitektur i uge 8 og 9, men snarere at gruppens egne retningslinjer siger, at arkitekturen burde være påbegyndt omkring det tidspunkt.

Uge	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Problemformulering																		
Kravspecifikation																		
Analyse																		
Arkitektur																		
Acepttestspecifikation																		
Design og implementering																		
Procesbeskrivelse																		
Acepttest																		
Færdiggørelse af projektrapport																		
Aflevering af projekt																		

Figure 6: Tidsplan for projektet, gruppe 2020f49.

3.10 Arbejdsfordeling

Arbejdsopgaverne er blevet inddelt primært efter gruppemedlemmernes ønsker, men også kompetence niveauet i de forskellige teknikker som er brugt. Der har af helt naturlige årsager været opgaver som har krævet hele gruppens opmærksomhed. Dette er opgaver som systemets overordnede arkitektur samt krav og problemformulering. Men i de mere tekniske opgaver som indebærer programmering af eksempelvis hjemmeside eller .NET API'er, har opdelingen været mere bestemt af kompetence niveau og lyst. En stor del af projektets opgaver er dog løst med peer-programmering og specielt i de mere kritiske og teknisk avancerede opgaver.

Gruppen har generelt været enige om at de bedste resultater skabes når gruppens individer finder opgaverne sjove og spændende, og der har derfor altid været mulighed for allegruppens medlemmer at ønske opgaver som også var udenfor personens spidskompetence.

Et skema over arbejdsfordeling kan ses i tabel 2 i afsnit 3.1

3.11 Continuous Integration/DevOps

Det blev besluttet tidligt at continuous integration var ønsket. Continuous integration, som også forkortes CI, er en udviklingspraksis hvor udviklere i projektet integrerer kode kontinuert i en fælles repository. Dette repository vil have en webhook, hvor automatiseret tests vil køre asynkront med udviklernes arbejdssproces [6]. Byggeserveren, som er den maskine som har til formål at køre de diverse tests, skal dertil opsættes med det ønsket CI software.

3.11.1 Opsætning af Byggeserver

Byggeserveren er en Windows virtuel maskine, hosted på en proxmox server. Proxmox er et open-source-server virtualiseringssmiljø som tillader hosting af containers og virutelle maskiner. Set på figur 7, er vist den virtuelle maskine, som internt kører TeamCitys continuous integration software. Denne virtuelle maskine, bliver hver søndag kl 05:00 slukket og gemt som et snap shot af Proxmox til data recovery.

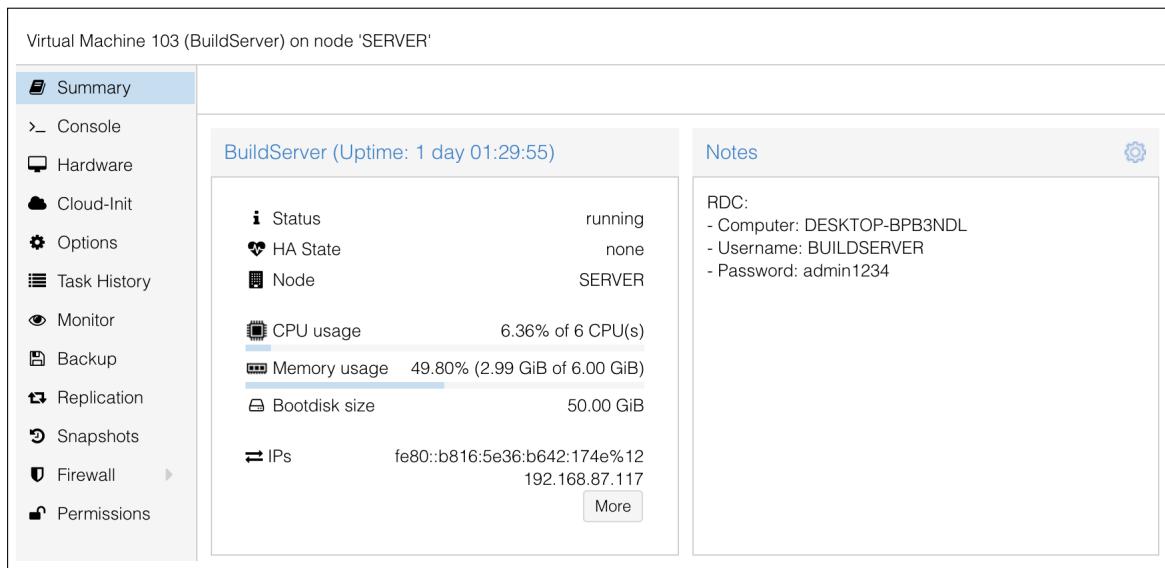


Figure 7: Proxmox vm. byggeserver

Set på figur 8, er vist at byggeserverens TeamCity-side er tilgængelig fra en remote connection. Porten 85 er blevet åbnet på hostens netværk og tillader derfor forbindelse fra klienter som ikke er på samme netværk. Til byggeserverens TeamCity hjemmeside, er der blevet lavet 4 admin kontier, én til henholdsvis hvert gruppe medlem, med email notifikationer på master branch bygge fejl eller den enkeltes gruppemedlem branch commit fejl.

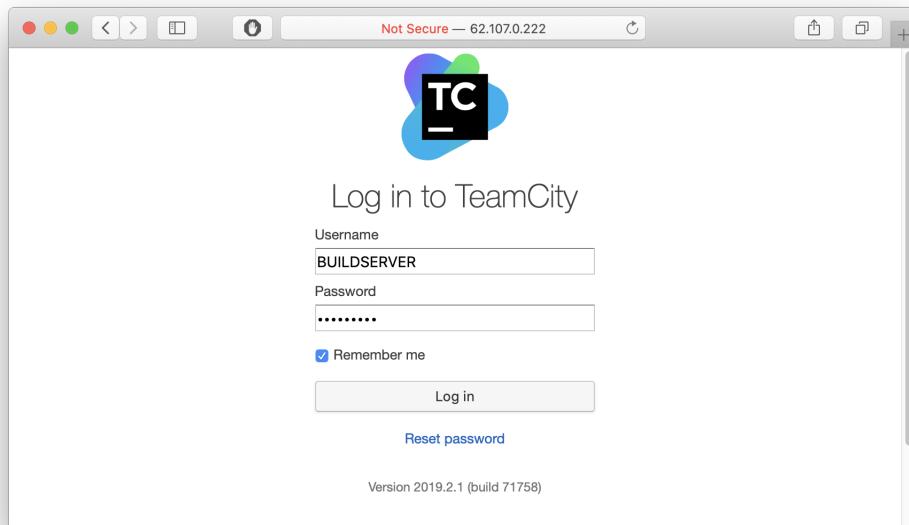


Figure 8: TeamCity login page hosted på vm.

3.11.2 Opsætning af pipelines

Den kørende TeamCity build server poller github for ændringer hvert 10'ende sekund, indtil der er sket en ændring. Når en ændring er sket, vil TeamCity hente den nuværende branch fra github og klargøre en bygge kontekst den kan bygge i, i form af en mappe. Når TeamCity har hentet filerne kan de forskellige definerede bygge steps startes. Da nogle af gruppens komponenter skal hostes i kubernetes, er specielle steps nødvendige, da byggede docker images skal ende på en registry hvor kubernetes kan hente dem. Hvert step er afhængig af hi-

nanden som en pipeline, hvilket betyder at hvis det første build step fejler, vil test-steppet ikke blive kørt. På figur 9 ses de nødvendige steps for at verificere et valid push til en branch:

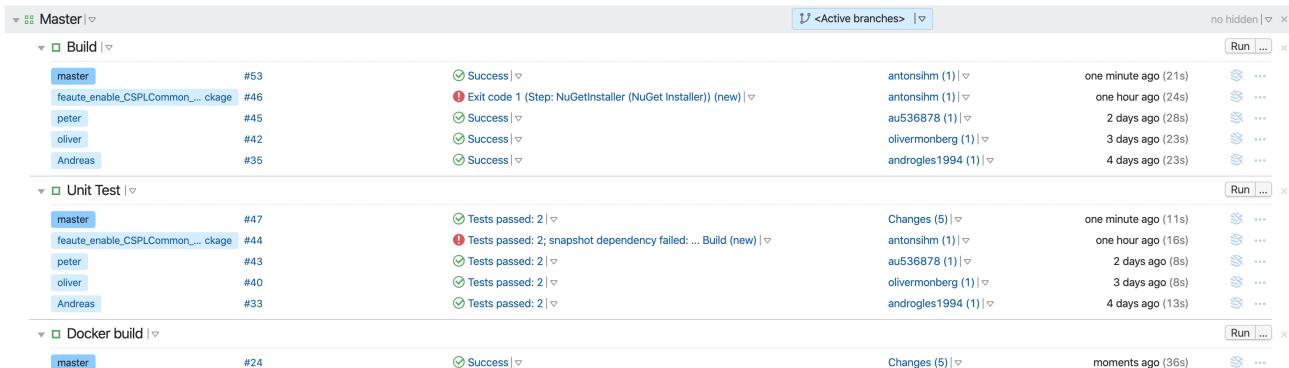


Figure 9: Pipeline for kubernetes komponenter

- **Build(Alle branches)** Kalder dotnet restore som restore missing dependencies som fx. nuget pakker samt kalder dotnet publish som bygger projektet via MSBuild.
- **Unit Test(Alle branches)** Kører alle test i projektet.
- **Docker Build(Kun Master)** Bygger et docker image som bliver tagget via semantisk versionering, som efter publisher det byggede image til et public registry på docker hub. Dette step kører ikke på andre branches da det ikke giver mening at skubbe det byggede docker image til docker hub.

Som det ses på figur 10, vil der efter et push til master, blive lagt et færdigt bygget docker image op i skyen på et public image registry på docker hub : <https://hub.docker.com/u/cspl1>



Figure 10: Master image på Docker Hub.

En anden pipeline der også er nødvendig for projektet er en pipeline der kan bygge en NuGet-pakke. Nuget pakken bliver versioneret, samt skubbet til en NuGet-server som udviklere kan gøre brug af, fx fra Visual Studio som det ses på figur 11



Figure 11: NuGet package overview Visual Studio.

Pipelinen for at bygge en NuGet-pakke er lidt anderledes for bygget en NuGet-pakke:

- **Build(Alle branches)** Kalder dotnet restore som restore missing dependencies som fx. nuget pakker samt kalder dotnet publish som bygger projektet via MSBuild.
- **Unit Test(Alle branches)** Kører alle test i projektet.

- **Package NuGet** Pakker de byggede filer til en nuget pakke, samt versionere pakken via semantisk versionering.
- **Publish** Publisher den færdige pakke, til en public NuGet-server som udviklere kan hente pakken fra

Det vigtige ved at have en automatiseret pipeline, på udvikler branches, er at udviklere hele tiden kan få verificeret at ældre test stadig kører, samt nye features også kører som de skal. Derved reduceres risikoen for at forskellige komponenters features ødelægges, når ny funktionalitet tilføjes løbende.

3.11.2.1 Versionering af komponenter

Til versionering af komponenter er det valgt at bruge semantisk versionering. Ideen med semantisk versionering, er at holde styr på hvornår der er breaking changes, tilføjet features eller hotfix/bugfix. Som det ses på figur 12 består versionering af tre tal, med hver deres vigtige betydning. Hvis der er sket en breaking change, som gør at de klienter der gør brug er komponenten også breaker, vil major nummeret blive bumpet. Hvis en feature er blevet implementeret i komponenten uden at breake noget, vil tallet i midten blive bumpet. Hvis en bug er blevet fixet for en feature, vil det sidste tal også blive bumpet. Det ses på forrige figur 10 at der også er et nummer mere efter bugfix. Dette nummer er et autogeneret bygge-nummer fra TeamCity bygge serveren, som sikre at samme komponent-version aldrig vil blive publiceret mere end en gang. Yderligere sikre semantisk versionering også, at klienter har mulighed for at tage stilling til om det vil opdatere deres pakke.

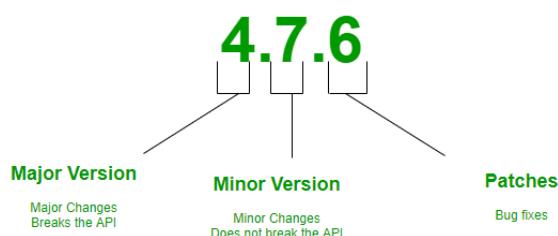


Figure 12: Semantisk versionering

3.12 Udviklingsværktøjer og programmeringssprog

Til at supportere udvikling af produktet er der i processen gjort brug af en række forskellige remedier. Dette er blandt andet værktøjer til kodning, design, arkitektur, orkestrering og processtyring.

- **Git** er brugt til versionsstyring og backup.
- **Jira** er et værktøj til styring af processer i form af scrum, sprints og backlogs. Det har integrationer med værktøjer som Git og Teamcity.
- **TeamCity** er brugt til continuous integration i form af unit-testing, integration-testing, deployment af NuGet-pakke versioner og byg af images til docker-hub.
- **app.diagrams.net** er brugt til udarbejdelse af arkitekturens diagrammer. Det er et gratis værktøj som har masser af muligheder for at lave diagrammer som forklarer arkitektur og design.
- **Microsoft Visual Studio 2019** er et stort IDE til udvikling af diverse applikationer. Alle projektets ASP .NET Core applikationer er udviklet her.
- **Microsoft Visual Studio Code** er et lightweight IDE med muligheder for extension-pakker. Her er projektets hjemmeside programmeret.
- **Docker** som værktøj til at lave, deployere og køre applikationer ved at bruge containere.
- **Kubernetes** er et orkestrerings værktøj som har et hav af muligheder i forhold til automatisk deployment, skalering og styring. I dette projekt er Raspberry Pi clusteret orkestreret vha. Kubernetes.

- **Powershell** er benyttet til pipelines på TeamCity. Her er den brugt til at bygge dockerfiler og automatisk versionering af NuGet-pakken.
- **Unix shell** er benyttet til at styre al hardware i clusteret, bla. til opsætning af Kubernetes.
- **Teamspeak, Discord og Zoom** er blevet benyttet til dagligt at kommunikere internt i gruppen i form af samtale og skærmdeling.
- **Excel** er benyttet til at samle data og opstille grafer for diverse performance tests.
- **SequenceDiagram.org** er benyttet til at udarbejde sekvensdiagrammer.

Følgende liste er en beskrivelse af de forskellige programmeringssprog, frameworks og større biblioteker som er benyttet i udfærdigelsen af projektet.

- **C#** er programmerings sproget som er brugt til at lave alt det som ikke er hjemmesiden. Dvs. API'erne master og worker samt NuGet-pakken CSPL og client-test programmer.
- **.NET Core** er det framework som er brugt til C# udviklingen af nogle af systemets applikationer. Webapplikationerne er specifikt udviklet i ASP.NET Core. .NET Core er cross-platform og er derfor brugbart til udvikling på Windows samt Raspberry Pi's og Docker containere med Linux.
- **HTML, css, javascript** er henholdsvis markup-language, styling og logik til opbygning af hjemmesider.
- **React** er et UI-library eller light-weight framework til javascript.
- **React material UI** er et bibliotek med react komponenter til afbenyttelse.

3.13 Coronavirus tilpasning

I den første del af projektet havde gruppen fået tildelt et lokale til alle fire mand oppe på skolen ved Katrinebjerg. Her kunne vi arbejde sammen og have alt vores hardware til clusteret stående. Efter en måneds tid kom udmeldingen om at alle institutioner og dermed også Aarhus universitetet, skulle lukke for alle studerende. Dette betød at gruppen ikke kunne mødes i lokalerne på skole hvor gruppens base lå.

Selvom omstændighederne var strenge havde gruppen stadig en bachelor opgave som skulle afleveres, og derfor blev der fundet måder at tilpasse sig den mærkværdige situation. Alt hardware blev sat op hjemme hos et af gruppemedlemmerne, dette gruppemedlem har statisk-IP og der blev åbnet en port på hans modem, så de øvrige gruppemedlemmer kunne arbejde på clusteret hjemmefra. En forklaring om opsætning af portforwarding kan læses i afsnit 7.12.

For at kommunikere sammen har gruppen benyttet sig af Teamspeak. En af gruppemedlemmerne havde en Teamspeak server som gruppen har brugt under hele forløbet. Derudover har gruppen benyttet sig af programmet Discord til at dele skærm, når det har været nødvendigt. Det har eksempelvis været nødvendigt til peer-programmering, som gruppen har benyttet sig meget af. Derudover er programmet Zoom blevet brugt til at holde møde med vejleder.

Alle gruppens medlemmer har vist stor fleksibilitet omkring den mærkværdige situation som Corona har sat hele Danmark i. Selvom alle er enige i at det er nemmere at kommunikere med hinanden, når man er i samme rum, så har de forskellige værktøjer som er taget i brug, gjort det muligt for gruppen fortsat at samarbejde.

3.14 Mødeindkalderer og mødereförater

Følgende afsnit indeholder mødedagsordener og mødereförater for projektet.

Referat - møde #01		<input type="checkbox"/> Uden Vejleder <input checked="" type="checkbox"/> Med Vejleder
Dato: 04-02-2020	Tid: 8.15	Lokation: Kahn 245
Fremmødte: <input checked="" type="checkbox"/> Andreas Schjødt Nielsen <input type="checkbox"/> Anton Sakarias Rørbaek Sihm <input checked="" type="checkbox"/> Jesper Rosholm Tørresø <input checked="" type="checkbox"/> Oliver Monberg-Jensen <input checked="" type="checkbox"/> Peter Marcus Hoveling	Udeblevet:	
	Uden afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	Med afbud: <input type="checkbox"/> Andreas <input checked="" type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter
Dagsorden: <ol style="list-style-type: none"> 1. Kravspecifikation - hvad skal der til for at vi specificere noget som et krav - hvor specifikt skal det være? 2. Funktionelle krav - skal de kunne måles/bevises? 3. Tidsplan 4. Risikoanalyse 5. Formalia - sprog i rapporten, skabelon? 6. Budget 7. Cluster kabinet - LaserLab 8. Kubernetes i et Raspberry Pi cluster? 9. Fremtidige vejledermøder 10. Evt. 		
Referat: Ad: <ol style="list-style-type: none"> 1. Brug kubeadm 2. Det der et svært er at sætte netværket op. et Controller- og pop-netværk. Her bruges KubeAdm 3. Én master er ikke nok, én master til Kubernetes som IKKE er en del af applikationene og en anden til at være den HTTP kald'ene dirigeres til 4. Kabinet vil Jesper først give grønt lys for, når vi har en POC med måske 5-10 Pi's 5. Tidsplan, Risikoanalyse og krav fik Jesper ikke tid til 6. Prøv at eksekver DLL-fil lokalt i en docker container. 7. Jesper foreslår, at vi begynder med nogle tidlige prototyper 8. Find ud af om det overhovedet kan lade sig gøre. Prøv at parallelisere et arbejde på 2 lokale docker containere og saml resultatet. se om det kan lade sig gøre. 		

Referat - møde #02		<input type="checkbox"/> Uden Vejleder <input checked="" type="checkbox"/> Med Vejleder	
Dato: 11-02-2020 Tid: 9.45	Lokation: Kahn 245		
Fremmødte:		Udeblevet:	
<input checked="" type="checkbox"/> Andreas Schjødt Nielsen <input checked="" type="checkbox"/> Anton Sakarias Rørbæk Sihm <input checked="" type="checkbox"/> Jesper Rosholm Tørresø <input checked="" type="checkbox"/> Oliver Monberg-Jensen <input type="checkbox"/> Peter Marcus Hoveling	Uden afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	Med afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input checked="" type="checkbox"/> Peter	
Dagsorden:			
<ol style="list-style-type: none"> 1. PoC 2. Næste skridt - Cluster PoC 3. Budget 4. Næste vejledermøde 5. Evt. 			
Referat:			
Ad:			
<ol style="list-style-type: none"> 1. Vurdering af vores cluster, er der alternativer som er bedre? "Ryzen et eller andet". <ol style="list-style-type: none"> a. Noget med capping af hardware når vi bruger kubernetes. man kan konfigurere hvor meget cpu kraft der skal bruge på pi's. b. kubernetes kan ikke altid garantere en node med cpu kræft nok til at en pod kan køre. c. Hvad betyder det for os at vi putter kubernetes ovenpå vores cluster og gemmer hardwaren? Er det overhovedet muligt at dele opgaverne op i 100% parallel ? 2. Få for guds skyld skrevet ned at der ikke er nogen forretnings model eller kundevalg. VI ved ikke endnu hvordan det evt skulle sælges. Om vi hoster eller om brugeren selv kan lave cluster og hoste. 3. Evt følg en rigtig case og vis at clusteret virker. Frem for at have et generelt problem. eksempelvis problemer med data indsamling og procesering. Hurtig sortering eller transformering af data inden for en tidsbegrensning. 4. Forretningsmodel der kræver at vores teknik (clusteret) findes. find konkret eksempel. 5. Igen, konkretiser og vær fokuseret på hvad det er for et problem vi gerne vil løse. Hvis fokus bliver for genereisk er det svært at have en konkret case, og svært at finde ud af hvornår man er færdig. 6. Hold fokus på hver sin ting for sig. Lav prioritering af hvad der er vigtigst (parallelisering, kubernetes cluster opsætning) 			

<h2>Referat - møde #03</h2>		<input type="checkbox"/> Uden Vejleder <input checked="" type="checkbox"/> Med Vejleder
Dato: 25-02-2020	Tid: 9.45	Lokation: Kahn 245
Fremmødte:	Udeblevet: Uden afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	
	Med afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	
Dagsorden:	<ol style="list-style-type: none"> 1. Sprint resume 2. Process beskrivelse - hvor vigtig er den? Skal den medtages i rapport? 3. Aktørkontekstdiagram 4. Forretningsmodel 5. Evt. 	
Referat:	<p>Ad:</p> <ol style="list-style-type: none"> 1. Tencer CPU'er Google.(konkurrenter), hvad kan de tilbyde, som vi måske ikke kan? 2. Kubernetes cluster vs én stærk CPU fx. Ryzen AMD Threadripper 64 kerner? 3. Eventuelt teori omkring parallelisering. 4. Ha' en konkret case. Fortæl censor hvorfor det vi har lavet er smart 5. (Husk lige at sende Forretningsmodel/Dokumentations rapporten til Jesper) 6. Arkitektur i forhold til Flere towers (clusteret). Fx. hvad ville gavne mest, hjemmeside hosting på cluster vs cloud. Eller Db på master vs cloud? 7. Vertical vs. horizontal scallering 	

Referat - møde #04		<input type="checkbox"/> Uden Vejleder <input checked="" type="checkbox"/> Med Vejleder	
Dato: 10-03-2020 Tid: 9.45	Lokation: Kahn 245		
Fremmødte: <input checked="" type="checkbox"/> Andreas Schjødt Nielsen <input checked="" type="checkbox"/> Anton Sakarias Rørbaek Sihm <input checked="" type="checkbox"/> Jesper Rosholm Tørresø <input checked="" type="checkbox"/> Oliver Monberg-Jensen <input checked="" type="checkbox"/> Peter Marcus Hoveling	Udeblevet: Uden afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	Med afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	
Dagsorden: 1. Sprint resume 2. Skal vi skrive dokumentation om setup og installering af Kubernetes cluster? 3. Hosting af DB og Web på Firebase eller i cluster? 4. Kubernetes metrics? CPU usage for PI's på Kubernetes dashboard. 5. LAN-kabler. 6. Evt.			
Referat: Ad: 1. Skriv omkring problemer som går ud over det som står på nettet. Hvilke erfaringer har vi gjort os? Eksempelvis i opsætningen af cluster, skriv hvilke problemer vi har mødt og hvordan de er løst. Nævn eventuelt nogle enkelte problemer og referer til resten af problemerne i dokumentationen. 2. Ingeniører der siger hvis jeg bruger teknikken på den her måde, så kan jeg gøre et eller andet. Det giver bedre end bare at bruge teknikken. 3. Rapport skrivning:: Pointer først - argumentation bagefter fisken på disken då.			

Referat - møde #05		<input type="checkbox"/> Uden Vejleder <input checked="" type="checkbox"/> Med Vejleder
Dato: 17-03-2020 Tid: 9.45	Lokation: Zoom	
Fremmødte: <input checked="" type="checkbox"/> Andreas Schjødt Nielsen <input checked="" type="checkbox"/> Anton Sakarias Rørbaek Sihm <input checked="" type="checkbox"/> Jesper Rosholm Tørresø <input checked="" type="checkbox"/> Oliver Monberg-Jensen <input checked="" type="checkbox"/> Peter Marcus Hoveling	Udeblevet: Uden afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	Med afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter
Dagsorden: <ol style="list-style-type: none">1. Corona-situation2. Evt.		
Referat: Ad: <ol style="list-style-type: none">1. https://stackoverflow.com/questions/750574/how-to-get-memory-available-or-used-in-c-sharp2. deamon set		

Referat - møde #06		<input type="checkbox"/> Uden Vejleder <input checked="" type="checkbox"/> Med Vejleder	
Dato: 31-03-2020 Tid: 9.45	Lokation: Zoom		
Fremmødte:		Udeblevet:	
<input checked="" type="checkbox"/> Andreas Schjødt Nielsen <input checked="" type="checkbox"/> Anton Sakarias Rørbaek Sihm <input checked="" type="checkbox"/> Jesper Rosholm Tørresø <input checked="" type="checkbox"/> Oliver Monberg-Jensen <input checked="" type="checkbox"/> Peter Marcus Hoveling	Uden afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	Med afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	
Dagsorden:			
<ol style="list-style-type: none"> 1. Eksamensrapport 2. Hvor er vi? Kan vi nå det hele? Hvornår er produktet færdigt? 3. Sprint resume (LoadBalancer, Hjemmesiden, Error handling, refactorering og arkitektur, master på cluster) 4. Certificate trust i kubernetes 5. Rapport skrivning om 2 sprints 6. Næste step? 7. Evt. 			
Referat:			
<p>Ad:</p> <ol style="list-style-type: none"> 1. Forretningsmodel. Gør nogle overvejelser. 2. Krav i forhold til kø i clusteret. 3. Rapporten skal guide vejleder og censor igennem projektet. 			

Referat - møde #07		<input type="checkbox"/> Uden Vejleder <input checked="" type="checkbox"/> Med Vejleder
Dato: 14-04-2020 Tid: 9.45	Lokation: Zoom	
Fremmødte: <input checked="" type="checkbox"/> Andreas Schjødt Nielsen <input checked="" type="checkbox"/> Anton Sakarias Rørbaek Sihm <input checked="" type="checkbox"/> Jesper Rosholm Tørresø <input checked="" type="checkbox"/> Oliver Monberg-Jensen <input checked="" type="checkbox"/> Peter Marcus Hoveling	Udeblevet: Uden afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	Med afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter
Dagsorden: 1. Eksamens 2. Sprint resume 3. Implementeringsfase snart slut 4. Evt.		
Referat: Ad: 1. Skriv om hvordan man kan skrabe både pods, kubernetes og pi's til minimum (hvordan har vi gjort og hvad kunne man mere have gjort?). Pille ting ud af frameworkt. 2. Status til Kamstrup, business case.		

Referat - møde #08		<input type="checkbox"/> Uden Vejleder <input checked="" type="checkbox"/> Med Vejleder
Dato: 28-04-2020 Tid: 9.45	Lokation: Zoom	
Fremmødte: <input checked="" type="checkbox"/> Andreas Schjødt Nielsen <input checked="" type="checkbox"/> Anton Sakarias Rørbæk Sihm <input checked="" type="checkbox"/> Jesper Rosholm Tørresø <input checked="" type="checkbox"/> Oliver Monberg-Jensen <input checked="" type="checkbox"/> Peter Marcus Hoveling	Udeblevet: Uden afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	Med afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter
Dagsorden: <ol style="list-style-type: none">1. Implementeringsfase er slut2. Demo af produkt3. Præsentation af test cases/performance test4. Rapportskrivning (Formalia: https://studerende.au.dk/fileadmin/user_upload/Vejledning_til_udfaerdigelse_af_projektrapporter_v1.5.pdf)5. Evt.		
Referat: Ad: 1.		

<h2>Referat - møde #09</h2>		<input type="checkbox"/> Uden Vejleder <input checked="" type="checkbox"/> Med Vejleder
Dato: 13-05-2020	Tid: 8.30	Lokation: Zoom
Fremmødte:	Udeblevet:	
<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Andreas Schjødt Nielsen <input checked="" type="checkbox"/> Anton Sakarias Rørbaek Sihm <input checked="" type="checkbox"/> Jesper Rosholm Tørresø <input checked="" type="checkbox"/> Oliver Monberg-Jensen <input checked="" type="checkbox"/> Peter Marcus Hoveling 	Uden afbud: <ul style="list-style-type: none"> <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter 	Med afbud: <ul style="list-style-type: none"> <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter
Dagsorden:		
1. Status 2. Udkast til rapport 3. Evt.		
Referat:		
Ad:		
1. Forside 2. - Giv titel på forside 3. Generelt: <ul style="list-style-type: none"> - Skriv udvikling er datid, ting vi har gjort - Skriv nutid omkring det nuværende produkt. Det vi har i hænderne nu - Refleter mere, hvorfor er tingene som det er. - Iterationer omkring hvad der ikke virkede, om hvad vi gjorde for at få det til at virke(LoadBalancing) 		
4. Forord:		
<ul style="list-style-type: none"> - Forklar hvad det betyder "Cluster Supported Parallel librarries" cluster parallel processering 		
5. Kort beskrivelse af projektet :		
<ul style="list-style-type: none"> - Vores projekt er egentlig cloud computing - brug IKKE fremtid -- NUTID i stedet - Tænk som læser - Hvilke 2 applicationer? 		

- Hvorfor bruge Rasberry Pi cluster?? frem for google cloud eller amazon services

- Hvorfor bruge c# fremfor andet ? generelt hvorfor ?

Aktør kontext diagram:

- Nuget.org ---> code repository -- burde være mere generelt

- Firebase ---> database -- burde være mere generelt

- Cluster Supported palle libries -- Hvad er der i midten ? evt hvis billed er cluster hw

6. Process :

- Lav gerne subsections så det bliver mere læsbart

- peer-programming --> hvad har det givet os ?

- Scrum --> hvad har det givet os ?

7. Krav -- Skal ligges i billag

-- Furps -- Moscow -- userstories -- skal billag

Referat - møde #10		<input type="checkbox"/> Uden Vejleder <input checked="" type="checkbox"/> Med Vejleder
Dato: 20-05-2020 Tid: 14.30	Lokation: Zoom	
Fremmødte: <input checked="" type="checkbox"/> Andreas Schjødt Nielsen <input checked="" type="checkbox"/> Anton Sakarias Rørbaek Sihm <input checked="" type="checkbox"/> Jesper Rosholm Tørresø <input checked="" type="checkbox"/> Oliver Monberg-Jensen <input checked="" type="checkbox"/> Peter Marcus Hoveling	Udeblevet: Uden afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter	Med afbud: <input type="checkbox"/> Andreas <input type="checkbox"/> Anton <input type="checkbox"/> Jesper <input type="checkbox"/> Oliver <input type="checkbox"/> Peter
Dagsorden: <ol style="list-style-type: none">1. Feedback på udkast til rapport2. Evt.		
Referat: Ad: <ol style="list-style-type: none">1. Indledning et godt udgangspunkt, følg denne struktur igennem rapporten		

4 Analyse

I følgende afsnit beskrives nogle af de overvejelser og mulige løsninger til forskellige koncepter i projektet. Afsnittet giver beskrivelser af forskellige undersøgelser samt nogle "proof of concepts" som er blevet lavet for at demonstrere og verificere videnskabelig viden. Derudover skaber "proof of concepts" et overblik over kritiske vurderede aspekter i bachelor projektet kan læses i afsnit 3.7. Gennem analysen vil forskellige valg blive taget, ud fra hvilke fordele og ulemper der er for de forskellige løsninger.

4.1 Cluster hardware

Til projektets prioriteringen af FURPS, se afsnit 2.2, er funktionalitet og performance vægtet højst. Dette resulterer i, at valg af hardware skal tages med omhu og de endelige valg skal kunne argumenteres for.

For det samlede system, opdeles hardware i primær og sekundær hardware. Det primære hardware er Raspberry Pi's og masteren(J5005). Det er her at udstyret har den største indflydelse for systemets samlet performance. Det primære hardware danner således fundamentet for hele clusteret ydeevne og fremtidige forbedringer på denne front vil naturligvis gøre systemet dyrere.

Det sekundær hardware er komponenter som LAN kabler, strømforsyning, SSD, switch og router. Alle disse komponenter har en indflydelse systemet, men kun på det overordnede plan. Hvis LAN kablerne skulle være af type CAT 5 og ikke CAT 5/e, ville der være en naturlig flaskehals på 150 MBs. Det sekundære hardware skal altså ikke kunne skabe en flaskehals for det primære hardware.

En tabel for primær og sekundær HW.

Primær HW.						
Platform (enhed)	Role	Stk.	OS	CPU	Cores	Mhz
Raspberry Pi 3B	worker	15	Raspbian Buster Lite	Broadcom BCM2837	4	1.200 Mhz
Raspberry Pi 3B+	worker	5	Raspbian Buster Lite	Boradcom BCM2837B0	4	1.400 Mhz
J5005 mini-ITX	master	1	Ubuntu Desktop	Intel Quad Core Pentium silver J5005	4	1.500 Mhz

Sekundær HW.				
Platform (enhed)	Role	Stk.	Hastighed	
Cisco SG112-24	switch	1	10/100/1000 MBps	
Cisco Linksys WRT320N	router	1	10/100/1000 MBps	
Kingston SSDNow A400 SSD	master SSD	1	500 MBps(read) / 320 MBps (write)	
CAT.5e	LAN kabler	24	1000 MBps	

Master (J5005 mini-ITX) er af bundkort størrelsen mini-ITX, hvilket er en kompakt størrelse. Dette ITX-board, model (J5005-ITX), er udstyret med en Quad Core CPU og mulighed for op til 8Gb Dual DDR4-2400 ram. Bundkortet er udstyret med integreret GPU og passiv køling på CPU, hvilket resulterer i et lydløst kort. Valget om ikke at have en Raspberry Pi som master ligger i, at masteren kommer til at stå for en masse delegering af arbejde. Der er kun en replica af masteren og den har flere funktioner. Den står for Kubernetes Adm og har alene kontakten til firebase og eksponere det eneste endpoint klienter har mulighed for at ramme. Dette kræver naturligvis mere CPU og med de totale 8G ram, er der også mulighed for at cache flere programmer og evt. data, i memory. Masterens LAN port er også af gigabyte hastighed, hvilket tillader hurtigere trafik til og fra.

Worker (Raspberry Pi 3B3/3B+) er af serien Raspberry Pi Model 3B og 3B+. Denne model er valgt grundet dens LAN-port og quad core CPU. Raspberry Pi's ældre end model 3B er ca. i samme prisleje som 3B, har færre CPU kerner og ingen LAN-port [55]. Raspberry Pi'en har også dens eget Unix OS, Raspian, som er designet til at kunne udnytte de tilgængelige ressourcer bedst muligt. Dette i forlængelse med, at Kampstrup valgte at sponsorer CSP projektet, resulterede i Raspberry Pi's som worker enhed.

Switch (Cisco SG112-24) Som skal være bindeleddet mellem master og workers, skal have nok porte, men

samtidig være hurtig nok til ikke at skabe en flaskehals. Switchen har ingen indstillings muligheder (den er headless) og understøtter LAN hastigheder op til 1 Gbit.

Selvom Raspberry Pi's kun understøtter hastigheder op til 150 MBit, skal de mange requests fra master til workers og fra Kubernetes til workers, ikke begrænses til 150 Mbit.

Router (Cisco Linksys WRT320N) står for konfigureringen af alle statiske Ip'er i clusteret. Uden statiske Ip'er vil Kubernetes Adm have problemer med at kommunikere med nodes. Ved en nodes anmodning om at tilkoble clusteret, bliver nodens Ip logget. Ændres denne nodes Ip, har den ikke længere adgang til clusteret og afvises. Hertil har Linksys mulighed for at konfigurerer statiske Ip'er via den indbygget admin panel.

Portforwarding er også opsat via routeren. Denne funktionalitet er kommet i brug i forbindelse med corona krisen, se afsnit 3.13, hvor trafik fra klienter som ikke har adgang til det lokale netværk, således kan forbinde. Ønsker administratoren at monitorer netværks trafikken er det også mulighed for dette via dens indbygget wifi-hotspot.

Master SSD (Kingston SSDNow A400 SSD) er master nodens primære lagerings komponent. En hurtig SSD betyder hurtigere loading af filer og programmer ind i ram, og samtidig tilbyder hurtigere genstart og installering af programmer.

LAN kabler (CAT .5e) Hvert LAN-kabel er af type Cat5e. Disse kabler har en maksimal dataoverførsels hastighed på 1Gbit/s, hvilket stemmer overens med hastighederne for master og switch .

Yderligere er en USB hub købt til at levere strøm til alle workers. Hver port i USB hubben skal kunne levere 2.5A:5V til hver Raspberry Pi. Strømkabler til Raspberry Pi's er også købt med omtanke for at levere som minimum 5A.

4.2 Jenkins vs. TeamCity

Valgtet til hvilket CI software byggeserveren skulle køres med, stod mellem Jenkins og TeamCity. Jenkins, som er open source værktøj, er væsentlig ældre end TeamCity og tilbyder dertil et større community og generelt set brugt mere. Jenkins er også værtsat for dens mange forskellige plugins og integrationer med andre platforme.

TeamCity, som også er CI værktøj, er oprettholdt af JetBrains og er forholdsvis nyt til sammenligning med Jenkins. TeamCity har ikke et lige så stort community som Jenkins, men er betragtet af dens brugere, til at være lettere at bruge og mere direkte i opsætningen af pipelines [57].

Opsummering af hvad Jenkins og TeamCity tilbyder:



Jenkins:

- Gratis open source
- Rig på plugins med god dokumentation
- Mange integrations muligheder
- ...



TeamCity:

- Let at konfigurere
- Github integration
- Brugervenligt
- ...

Det er besluttet at benytte TeamCity, da den lettere konfigurerering og brugervenlighed vægtes højt.

4.3 Master OS

Valget om hvilket operativ system, Masteren skulle have, overvejes flere ting end bare kendskab til operativ systemet. Eksempelvis hvilket OS Docker og Kubernetes kører bedst, hvilke muligheder der var til internet deling, performance og stabilitet.

4.3.1 Internetdeling og pålidelighed

En nødvendighed for master operativ systemet, er muligheden for at dele netværk. Da alle Raspberry Pi nodes er forbundet til en headless switch, som ikke selv er forbundet til en router, er det nødvendigt at master OS'et kan dele dens wifi forbindelse via en LAN port til switchen, *se figur 13*.

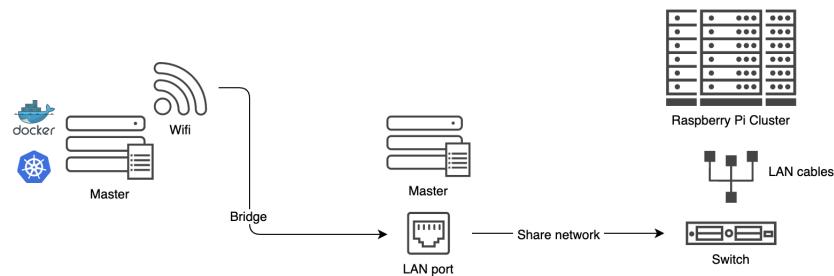


Figure 13: Master Wifi leverer internet til switch igennem LAN port

Prekonditionen for overhovedet at kunne dele internet forbindelse er at have et netværkskort som understøtter dette. Da hardware specifikationerne er opfyldt på masteren, mangler der et OS der kan tilbyde internetdeling. Både Windows og Linux tilbyder internetdeling. Windows har et userinterface til at opnå dette, og Linux en CLI.

Et punkt hvor Windows og Linux dog differerer fra hindanden er deres pålidelighed. Pålidelighed som skal forstås i hvor stabilt systemet er, er Linux specielt kendt for at være særligt godt. Med ingen automatiske opdateringer og dertilhørende påkrævet genstart af systemet, er Linux på det punkt specielt hvertsat.

4.3.2 Kubernetes og Docker på Linux vs. Windows

Kubernetes som container-orkestrationssystem er ligeglæd med om den køres inde i en virtuel maskine, på 'bare-bone' hw eller på en host pc. Hvis der vil køres linux containere fra en windows maskine, skal en en virtuel linux kerne imellem således linux container kan bruge den virtuelle linux kerne som host. Hvis der vælges linux som OS kan containerne direkte bruge host OS'et i stedet for et virtuelt et som på windows.

Kubernetes er generelt ligeglæd med om nodes er faktiske bare-bone metal maskiner eller virtuelle maskiner, dette ses på figur 14.

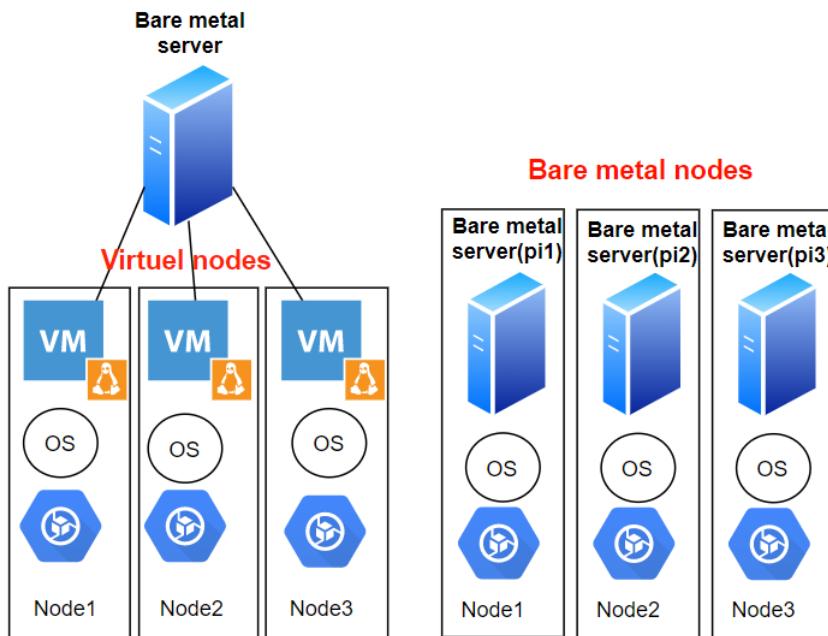


Figure 14: Virtuelle nodes vs. rigtige nodes.

Nogle ligheder og uligheder stillet op på punkt form:

Ligheder [59]

Docker-containere på Linux og Windows ligner hindanden på følgende måder:

- De er begrænset til at indeholde applikationer, der naturligt understøttes af host OS'et. Med andre ord kan Docker til Windows kun være host for Windows-applikationer inde i Docker-containere, og Docker på Linux understøtter kun Linux-apps.
- De kører natively, hvilket betyder, at de ikke er afhængige af hypervisorer eller virtuelle maskiner.

Forskelle [59].

Hvad der gør Docker på Windows forskelligt:

- Nogle Docker-netværksfunktioner til containere understøttes endnu ikke på Windows
- Docker understøtter kun visse versioner af Windows

Der er ikke et klart valg til valg af master OS. Begge operativ systemer fungere magen til. Kubernetes er kun bekymret med at have et Linux image som host OS, og dette krav tilbyder Docker med et ekstra abstraktionsslag.

Linux kan argumenteres for at være mere stabilt end Windows, men kræver erfaring med dens CLI og filsystem, frem for den lettere Windows GUI. Hertil vælges Linux, alene på det grundlag, at det er mere stabilt.

4.4 Raspberry Pi OS

De officielle operativ systemer for Raspberry Pi's er Raspbian og NOOBS. Raspbian kommer med pre-installeret software til programmering og generel brug. Såsom Python, Scratch, Sonic Pi, Java og mere [15]. Raspbian fås med og uden desktop, hvor Raspbian med desktop fylder 1123MB og Raspbian uden 435MB. Til de officielle operativ systemer, findes der mange andre 'uofficielle' operativ systemer. Klassiske linux operativ systemer, såsom Ubuntu, Debian og Mint er blandt mange som også er mulige at installere på en raspberry pi.

Da der findes mange forskellige operativ systemer skal vælges med omtanke for bla. arkitektur (32- eller 64-bit), størrelse, GUI og support.

Arkitekturen for et operativ system diktterer hvilke typer programmer der kan køres og hvor meget hukommelse (ram) kan tilgås. Selvom Raspberry Pi serien 3B og 3B+, som clusteret har, har en 64-bit processor, kan et 32-bit operativ system flaskehælse systemet, således at kun 2^{32} -bit kan tilgås $\sim 4GB$ ram. Hvor et 64-bit operativ system kan tilgå 2^{64} -bit hukommelse.

Programmer kompileret til enten 32- eller 64-bit vil kun kunne afvikles på et operativ system med 64-bit, dog kan 32-bit programmer kun afvikles på 32-bit. Til .NET er der dog en undtagelse. Når der kompileres .net bliver det ikke assembled. IDE'et kompilere koden til et intermediate object (kendt som en .net assembly) som ikke er assembled. Typisk når kode kompileres, fx. i sprog som C/C++, kompileres det til assembly instruktioner, som deliges videre til en assembler som så genererer maskine kode, hvilket er specifik for CPU'ens arkitektur.

Humlen ved .NET er, at når en bruger åbner et program, startes der en instance af .NET runtume, hvor programmet køres i. Programmet køres aldrig direkte på maskinen selv. Når det kørende program skal kalde en metode, bruges en speciel komponent i CLR(.NET virtuel machine) kaldet JIT(just in time compiler), som har til opgave at tage ensemble MSIL (microsoft intermediste language) til de specifikke maskine kode instruktioner. Da clusteres worker nodes, afvikler deres arbejde med .NET DLL'er, betyder det at arkitekturen på det valgte operativ system ikke har den store betydning når der arbejdes med .NET [47], derud bygges kodes typisk til any-CPU således både kan afvikles på 32-bit og 64-bit.

Selvom et operativ system såsom Ubuntu 64-bit tillader udnyttelsen af mere ram, har clusterets Raspberry Pi's kun 4Gb ram.

64-bit operativ system har til tider at have en bedre performance på afvikling af programmer, men kan samtidig være et overhead som ikke gavnner.

Raspbian som er det officielle operativ system til Raspberry Pi's, lover hw-acceleration, adgang til GPIO pins og optimeringer rettet mod det specifikke HW. Raspian er kun lavet i 32-bit, men i forlængelse af 64-bit .NET DLL'er afvinkling ikke er et problem, gør at **Raspian Buster Lite** vælges.

4.5 Orkestrering

Hvad er softwareorkestrering og hvad kan det tilbyde et projekt som CSPL? Når man snakker om softwareorkestrering er det ofte i sammenhæng med service-orienteret arkitektur, virtualisering og konvergeret infrastruktur [63]. Disse går hånd i hånd med softwareorkestrering, idet softwareorkestrering er den automatiserede konfiguration, styring og koordinering af computersystemer, applikationer og services. Softwareorkestrering kan på denne måde hjælpe til lettere at styre komplekse opgaver og arbejdsprocedurer [54].

Der findes mange forskellige former for orkestreringsværktøjer til softwareorkestrering som f.eks. Microsoft Service Fabric og Kubernetes. Værktøjer som disse, gør det nemt at pakke, implementere og administrere skalbare og pålidelige services. Brugen af et orkestreringsværktøj adresserer således de væsentlige udfordringer ifm. udvikling og styring af cloud-applikationer såsom CSPL systemet. Udviklere kan undgå komplekse problemer med infrastruktur og fokusere på at implementere arbejdsopgaver, der er skalbare, pålidelige og nemt håndterbare. [39]. Orkestreringsværktøjer som Microsoft Service Fabric og Kubernetes gør dette muligt igennem deres container orkestration.

Set på figur 15, er der vist hvordan og hvorfor containerorkestrering er smart. Traditionelt set var applikationer hosted direkte på et host operativsystem og det underliggende hardware, det man kalder 'bare-metal'. Senere kom der virtualiseret deployments, som stadig bruges i dag og har sine tradeoffs i kontrast med container deployments. Virtualiseret deployments er med hjælp fra en hypervisor [62], som har det formål at oprette og køre virtuelle maskiner, en måde at opdele host-computerens ressourcer på, således at applikationer kan køre i deres eget miljø. En computer med en hypervisor, som kører en eller flere virtuelle maskiner, kaldes en 'host machine', hvor hver virtuelle maskine kaldes en 'guest machine'.

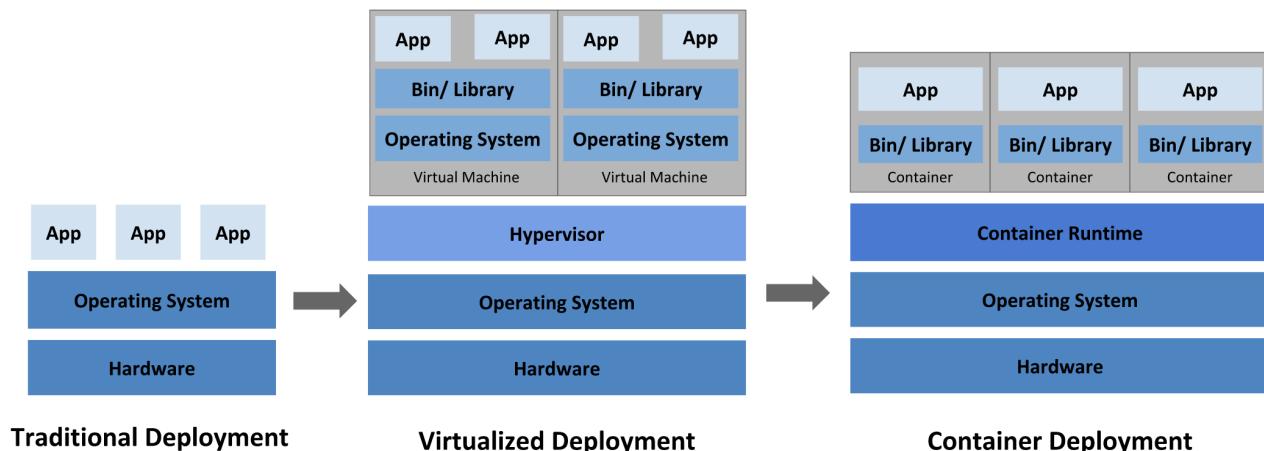


Figure 15: Containerorkestrering, billedet stammer fra [68]

Selvom virtualiseret deployment stadig har sin plads i orkestreringsmiljøet, er orkestrering i form af container deployment, den metode orkestreringsværktøjer som Kubernetes og Microsoft Service Fabric benytter. Forskellen mellem virtualiseret deployment og container deployment er hvordan ressource opdeles. I forhold til en virtuel maskine skal en container ikke have dens eget operativsystemet og kræver dertil heller ikke en hypervisor. Med en container deployment, bliver host-OS'et opdelt i mindre bidder, som hver container får en del af. En container kører som en process på host-OS'et og kan af denne grund starte hurtig op og lukke hurtigt ned. Applikationen som køres inde i en container, har også et mindre abstraktionslag imellem den fysiske hardware. Container deployment er også smart grundet evnen til at have mange containere. Ift. en virtuel maskine, skal en container ikke have afsat en fast mængde RAM, CPU kerner og diskplads, der kan i stedet sættes en maks begrænsning for dette og containeren kan dynamisk kræve de resourcer fra host'en som den behøver.

4.5.1 Kubernetes som orkestreringsværktøj

CSPL systemet bygger på et cluster som er sammensat af 20 Raspberry Pi's og et J5005 mini-ITX motherboard. Til orkestrering af dette, er de tre deployment-typer sammenlignet, med henblik på at få et overblik over hvilken metode der passer bedst ift. de hvilke udfordringer der er til CSPL systemet.

- **Traditional**

- + Pros: Mindre abstraktion fra hardware.
- + Pros: Til tider, et minimalt overhead.
- Cons: Monolit orkestrering.
- Cons: Ingen features fra orkestreringsværktøjer som Microsoft Service Fabric og Kubernetes.

- **Virtualized**

- + Pros: Mulighed for forskellige operativsystemer på samme maskine.
- + Pros: Isolering af programmer og logik.
- + Pros: 'Self-healing' i form af restat af virtuel maskine.
- Cons: Langsom restat af virtuel maskine.
- Cons: Et ekstra abstraktionslag.
- Cons: Dyrt i ressourcer at lave virtuelle maskiner.
- Cons: Ingen features fra orkestreringsværktøjer som Service Fabric og Kubernetes.

- **Container**

- + Pros: Hurtig 'Self-healing' i form af container restart.
- + Pros: Load balancing.
- + Pros: Ingen downtime under opdateringer (rolling updates).

- + Pros: Mulighed for horizontal-skalering.
- + Pros: Isolering af programmer og logik.
- Cons: Kun mulighed for én type operativsystem.
- Cons: Højere læringskurve.

Af de tre deployment-typer, er valget endt med container deployment. Ved at vælge en container deployment, kan orkestreringsværktøjer som Kubernetes og Microsoft Service Fabric benyttes. Gældende for dem begge er, at de kan orkestre containere igennem fil-konfigurationer. Microsoft Service Fabric konfigureres via en pakke med konfigurationsfiler og Kubernetes igennem .yaml-filer. Begge værktøjer har hver deres tradeoffs, men minder generelt om hinanden, som automatisk container restarts, horizontal-skalering og 100% uptime under rolling opdateringer [22].

I og med at hvert gruppemedlem modtager undervisning i faget ITONK Objektorienteret netværkskommunikation [60], samt at projektgruppens vejlederen Jesper Rosholm Tørresø underviser i dette fag, gør at Kubernetes vælges som orkestreringsværktøj.

4.6 Docker

4.6.1 Docker Engine

Docker Engine giver mulighed for at køre applikationer i containers. En container er et virtuelt OS, som er isoleret fra det faktiske OS som containeren kører på [10]. Forstået sådels, at en container får allokeret en del af host OS'et, så den underliggende kernel kan bruges. Docker Engine kan hoste disse containere og er oplagt at benytte i forbindelse med Kubernetes, men der kan også benyttes andet container hosting software som f.eks. rkt (Rocket) [56]. Valget er faldet på Docker Engine, da det er mere udbredt og der derfor findes mere beskrivelse af det online.

4.6.2 Docker Swarm (vs Kubernetes)

Det er muligt at benytte Docker som container orkestreringsværktøj vha. swarm mode. Docker Swarm er forholdsvis lightweight, samt nemt at opsætte og benytte sig af, hvis man allerede har erfaring med Docker CLI'en. Sammenlignet med Kubernetes som container orkestreringsværktøj, tilbyder Docker Swarm mindre kompleksitet, men også mindre funktionalitet og har bla. derfor ikke en så høj en læringskurve som Kubernetes. Kubernetes tilbyder bedre værktøjer for intern kommunikation og organisering imellem pods (containers), som kan være en fordel i et produktionsmiljø [28]. Docker Swarm er som udgangspunkt mere skalbart og er lettere at skalere hurtigt. Kubernetes er bedre når det kommer til at opretholde clusteret styrke vha. automatisk skalering, da det kan analysere belastningen på clusteret og skalere derefter. Dette gøres eksempelvis ved at tilføje flere nodes [17]. Til sidst er der et mindre community omkring Docker Swarm end Kubernetes, der er derfor meget information om Kubernetes online. For at konkludere er valget imellem Docker Swarm og Kubernetes landet på Kubernetes.

4.6.3 Dockerfiler og images

Docker kan benyttes til at bygge brugerdefinerede images, som indeholder runtime dependencies, for den applikation der skal køres på det specifikke image, f.eks. .NET Core runtime. Dockerfiler kan hjælpe med at specificere, hvordan disse images skal bygges. Derudover kan der f.eks. også specificeres i en dockerfil, hvordan en .net core applikation skal bygges/publishes. Det er bla. muligt at starte et image op under byggeprocessen, bygge applikationen på det specifikke image og derefter kopiere programmet (samtidig med dependencies der måtte være) over på det image, der skal køres i en container. Disse images skal bygges til de korrekter processorarkitekture (f.eks. arm eller amd64) alt efter hvilket hardware de skal køres på.

4.7 Valg af programmeringssprog & frameworks

Backend:

Der er valgt C#, ASP.NET Core og .NET Core til udviklingen af backenden grundet følgende:

- Softwareudviklers primær programmering sprog er C# på .NET Core platformen.

- Valgt for at få den højeste produkt kvalitet.
- Cross-platform med .NET Core.
- Comptible med linux containerne i kubernetes.
- Nem tilgængelig dokumentation da det kommer fra Microsoft
- ASP.Net Core er cross platform
- Hosting framework fra Microsoft
- Udviklerenes primær service hosting platform

Frontend

Der er valgt React til udviklingen er Frontend grundet følgende:

- Brugen af den virtuelle DOM
- Light weight i forhold til større frameworks som Angular
- CSS bibliotek som React material
- Client side rendering
- Masse dokumentation
- Mulighed for importering af biblioteker, såsom routing

4.8 .NET Core i clusteret

Under analyse af Dockerfiler og images i afsnit 4.6.3, viste det sig at det var oplagt at benytte officielle images fra Microsoft som kommer bundlet med .NET Core runtime. Der er images baseret på forskellige OS'er samt processorarkitekturen som hver især understøtter forskellige versioner af .Net Core [11]. De .NET Core applikationer som skal køre som services i Kubernetes clusteret kan altså eksekvere på disse images og derfor vil de blive benyttet som containere. Et alternativ ville være at bygge brugerdefinerede images med udgangspunkt i f.eks. Ubuntu-images til de forskellige processorarkitekturen der vil blive benyttet i clusteret og vha. dockerfiler bundle .NET Core runtime i de images. Et andet alternativ ville være at køre images uden .NET Core runtime bundle og sørge for at applikationerne som skal køre i clusteret er published som self-contained [38].

4.9 Én vs fire pods pr. Raspberry Pi

Ved hjælp af det valgte orkestreringsværktøj Kubernetes se afsnit 4.5.1, er der forskellige måder der kan orkestre workers/nodes i CSPL systemet. Da systemets hardware består af 20 Raspberry Pi's som helholdsvis har fire cores hver, er der to forskellige løsninger hvorpå CSPL systemet effektivt kan gøre brug alle cores. Som det ses på figur 16 er den første mulighed er oprette fire workers/pods pr. Raspberry Pi/node, således kan hver worker/pod få tildelt én core. Den anden mulighed er at sætte en worker/pod pr. Rasberry Pi/node således den givne worker/pod har fire cores. Se illustration af begge løsninger på figur 16.

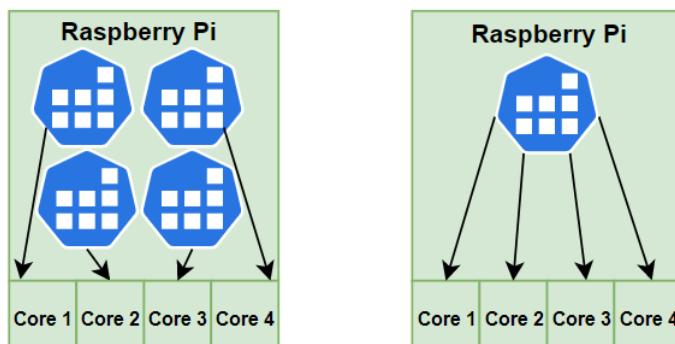


Figure 16: Forskellige orkesteringsmuligheder.

Én pod pr. Raspberry Pi

Med denne løsning vil clusteret bestå af 20 pods/workers, da der er 20 Raspberry Pi's. Klientens DLL'er er bliver sendt til en worker, skal være implementeret således at koden kan udnytte de x-antal cores der er på en worker-node. Klienten har her brug for at viden hvilken hardware han programmere op imod, da han vil have brug for at starte x-antal tråde, i dette tilfælde fire, for at udnytte alt CPU-kraft på Raspberry Pi'en. Derudover kan der kun requestes cores i clusteret i pakker af fire. En klient vil f.eks. ikke kunne requeste et skævt antal cores som 6 eller 14. Klienten vil derfor altid skulle dele sit arbejde op som vist i modulo forskriften 1:

$$NumberOfWorkers \mod 4 = 0 \quad (1)$$

Fordelen ved at lade en pod gøre brug af fire cores er, at en klient kan bruge flere tråde i hans program, dette gør det lidt sværere for klienten. Dog har det også visse fordele, da de forskellige tråde nemt kan kommunikere med hinanden, samt dele data.

Fire pods pr. Rasberry Pi

Den anden løsning vil være at lade clusteret bestå af 80 workers/pods. Således vil hver Raspberry Pi bestå af fire workers/pods, hvor hver især har én core. Ved denne løsningen har klienten ikke brug for at vide hvor mange cores der på worker-node hardwaren. Derudover skal klienten ikke selv oprette tråde for at udnytte Raspberry Pi's bedst muligt. Med denne løsning vil klienten kunne reueste alle former for antal workers mellem 1-80. Klienten vil ikke være fast låst til altid at skulle vælge pakker er fire workers. Klientimplementation af IClusterParallelizer vil også blive simplere, da klineten bare skal dele hans arbejde op i x-antal stykker, samt implementere en parallel-metode, som efterfølgende vil blive paralleliseret med "true parallelisme" via clusteret.

Endelig valg

Løsningen hvor hver Raspberry Pi har fire workers/pods er blevet valgt pga. følgende:

- Pros:

- Klientens implementering af det IClusterParalellizer, bliver simplificeret da klienten ikke selv skal oprette tråde.
- Klienten skal ikke kende hardwaren i clusteret for at kunne parallelisere yderligere på nodes i clusteret.
- Klienten vil kunne reueste alle former for antal workers mellem 1-80, og der vil dermed ikke være en restriktion for at antallet af workers modulus 4 altid skal være 0.

- Cons:

- Klienten vil med denne løsning ikke have mulighed for at dele data mellem tråde i hans parallelle algoritme.

4.10 Fjerneksekvering af kode

Når en klient gør brug af den tilgængelige NuGet pakke, for at implementere kode kompatibel med clusteret, er der forskellige løsninger for at få eksekvere koden på en ekstern maskine. Under runtime af brugerens program, skal det cluster relateret kode sendes til en ekstern maskine. Som det fremgår af figur 17 findes der tre forskellige metoder hvorpå klienten kan sende sit kode.

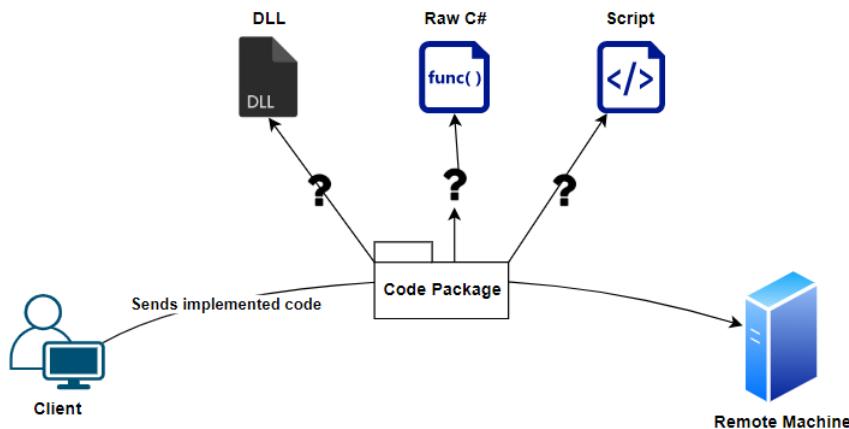


Figure 17: Fjern eksekverings muligheder

- **Script language:**

- **Pros:** Script language kan sendes "rå" fra klient til en ekstern maskine og eksekveres med det samme. Script languages er specielle languages da de består af commands der kalder ned i allerede compilere DLL'er. Scripts skal derfor ikke kompileres. De kan derfor med fordel blive kørt direkte på en ekstern maskine, uden at den eksterne maskine skal kompilere det først.
- **Cons:** Da slutbrugeren vil være en .NET-udvikler, giver det ikke mening at sende scripts. Den tilgængelige NuGet pakke er også en C#-pakke, hvilket gør at slutbrugeren også skal bruge C#.

- **Rå C#**

- **Pros:** Det er nemt blot at sende klientens ikke kompileret C# kode direkte til den eksterne maskine.
- **Cons:** Når den eksterne maskine modtager klientens kode, skal koden først runtime kompileres før det kan blive eksekveret, hvilket kræver tid. For at være yderst effektiv under runtime af det parallelle arbejde, nyttet det her ikke at der først bliver kompileret på runtime.
- **Cons:** Når ikke-kompileret C#-kode skal sendes til en ekstern maskine, vil det være en meget stor udfordring at finde frem til de dependencies der måtte være i koden.

- **DLL:**

- **Pros:** Når den eksterne maskine modtager en DLL kan den direkte via reflection kalde klientens metoder.
- **Pros:** Der skal ikke bruges tid på runtime-kompilering. Der kan på compile time klargøres DLL'er til fjerneksekvering.
- **Pros:** Når den eksterne maskine modtager en DLL, har den brug for at vide, hvad den skal lede efter i DLL'en. Der bliver derfor nødt til at være et striks interface mellem klienten og den eksterne maskine. Derved kan klienten love at implementere et interface IClusterParallelizer som har specifikke metoder som den eksterne maskine kan leder efter. Således vil den eksterne maskine vide hvad den skal lede efter i DLL'en og dermed hvad den skal eksekvere.
- **Cons:** Hvis den sendte DLL fra klienten kalder andre DLL'er, skal disse også sendes med.

Valg af metode til fjerneksekvering

Da fjerneksekvering af klientens kode skal være yderst effektiv, er det oplagt at klargøre den kode der skal sendes på compile time. Således kan klientens kode blive eksekveret direkte på den eksterne maskine. På figur 18 ses forløbet fra klienten til den eksterne maskine. Det oplagte valg for fjerneksekvering er at sende DLL'er, da klient-koden allerede er klar til at blive kørt på runtime. Ved at definere et brugervenligt interface, kan klienten simpelt implementere specifikke metoder, således at den eksterne maskine ved hvad den skal lede efter i DLL'en.

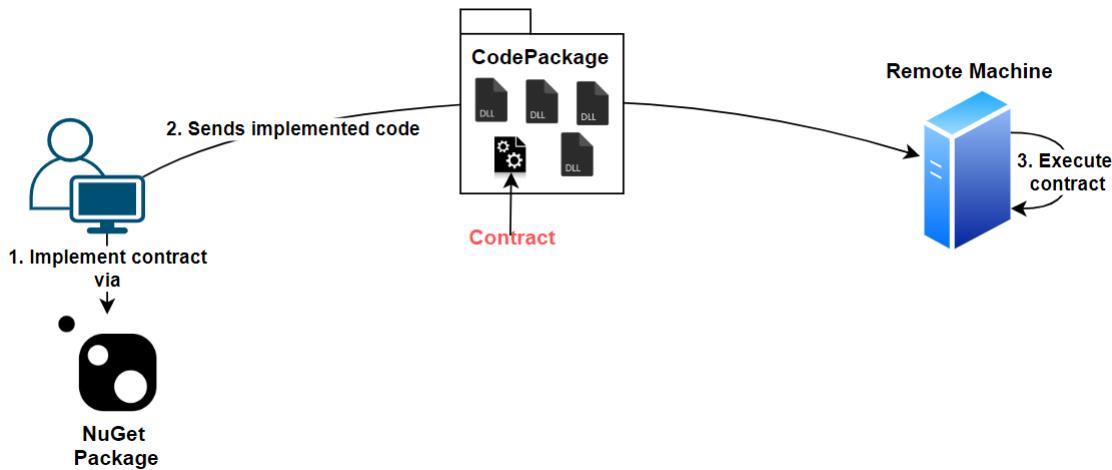


Figure 18: Fjerneksekvering af DLL'er.

4.10.1 Reflection

Reflection er en måde at dynamisk load typer og klasse-properties fra DLL'er på runtime. Reflection kan load typer samt lave instanser af typer. Når en instans af en type er lavet, kan der via reflection hentes attributter samt kaldes metoder på instansen [40].

4.11 Distribuering af arbejde til workers

Følgende afsnit giver en beskrivelse af, hvorledes arbejde kan distribueres fra master API'et til workers i clusteret, samt hvilke kritiske problemstillinger der er involveret i at distribuere arbejde på clusteret.

4.11.1 Undersøgelse af kubernetes loadbalancing

For at få et indblik i hvordan Kubernetes distribuer arbejde til workers er der blevet udarbejdet nogle tests. De problemstillinger der potentielt kan ligge i at distribuere arbejde til workers skal undersøges. Undersøgelsen går ud på at finde ud af, hvordan en Kubernetes service opfører sig, når den skal videregive arbejde til et rep-ketasæt af pods på clusteret.

Et andet problem er følgende scenarie; en klient vælger at splitte sit arbejde op i 10 pakker, for at få 10 pods at køre på. Hvis Kubernetes så vælger at uddeleger 2 pakker til den samme pod, kan jobbet risikere at tage længere tid. Grunden til dette er, at de første 9 pods kører på samme tid (parallel) og derfor bliver færdige på samme tid (først). Til sidst skal de 9 pods stå i idle og vente på at den sidste pakke uddelegeres til pod'en og at den bliver færdigeksekvet. Derfor er det vigtigt at have kontrollen over hvordan pakker bliver uddelegeret og hvilke pods de bliver fordelt på.

Problemstilling af distribuering via worker-servicen, ses på figur 19. Figuren viser at 2/10 pods ikke har fået noget arbejde, imens har 2/8 pods har fået mere arbejde end de andre, hvilket medfører længere ventetid på de to sidste pods. Det vil derfor være ønsket 'idle-states' på de forskellige worker-pods.

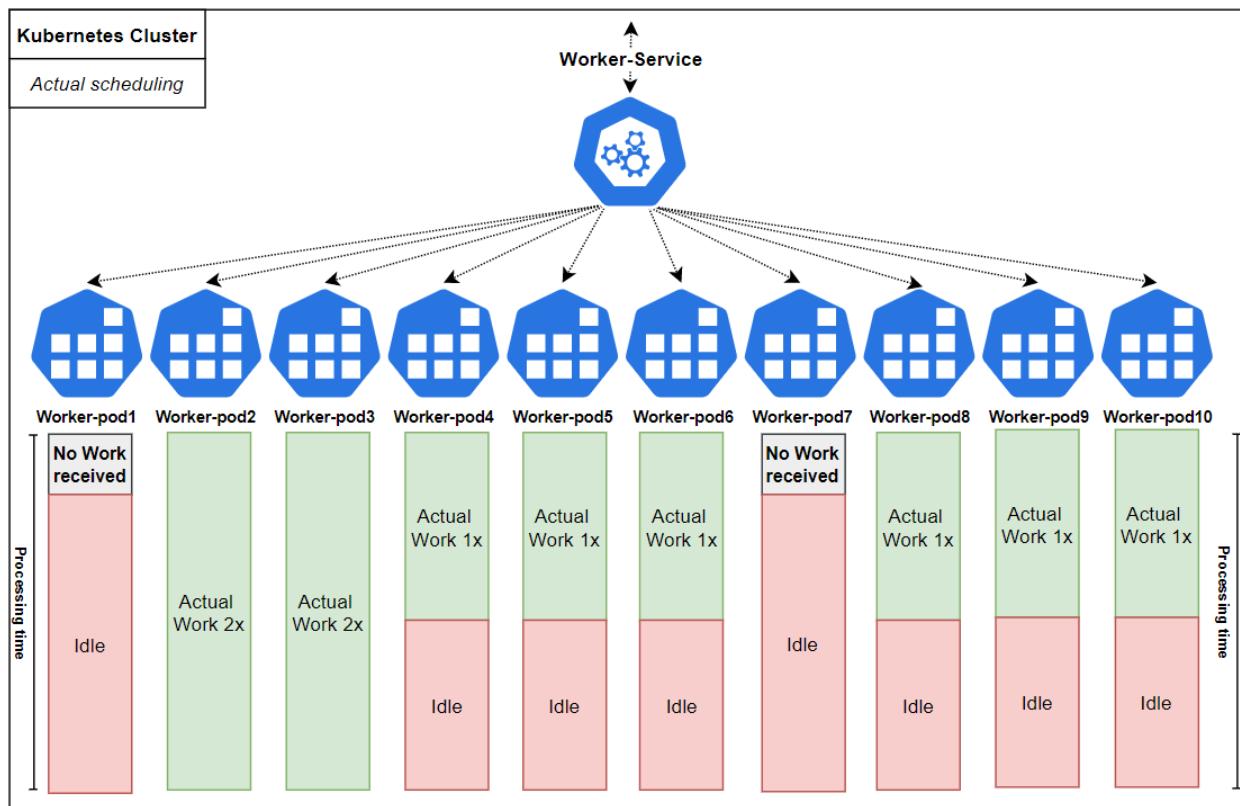


Figure 19: Kubernetes default skedulering.

Testen udføres på et lille cluster med otte pods fordelt på to Raspberry Pi's, for at afkraeft/bekräfte overstående hypotese. Klientprogrammet implementerer interfacet således at arbejdet opdeles i otte pakker. Jobbet sendes til masteren som opdeler pakkerne og sender otte http-requests til worker-servicen. Worker-servicen, som har til ansvar at fungere som en reverse proxy, giver en worker-pod-IP tilbage som master så kan sende arbejde til. Efter arbejdet tjekkes alle logs på de forskellige pods og det ses at nogle pods har fået to styks arbejde og nogle har intet fået.

Ud fra undersøgelsen kan det konkluderes at Kubernetes worker-sevicens vælger tilfældige pods og ikke får fordelt de forskellige arbejdsopgaver ud, således at der kommer ligevægt i clusteret. Dette er et stort problem da en Raspberry Pi B/B+ kun har fire kerner, og dermed kun kan køre "true parallelism", hvis hver kerne kun får et stykke arbejde ad gangen. Hvis der bliver tildelt flere opgaver til en worker-node/Raspberry Pi, end den har antal af kerner, vil nogle af kernerne blive delt og operativsystemet vil hertil lave context switching på eksekveringen af hvert stykke arbejde samtidigt, også kaldt concurrency.

På figur 20 ses den ønskede fordeling af arbejde markeret med grønt og den uønskede markeret med rød. På den øverste fordeling markeret med rød, ses det at én ud af de fire kerner eksekvere forskellige arbejdsopgaver på samme kerne. Raspberry Pi'en har fået for mange arbejdsopgaver til at køre "true parallel" og skedulerer derfor nogle af opgaverne ved siden af hinanden på en delt kerne. Den ønskede fordeling markeret med grøn, nederest på figur 20, viser at hver af de fire kerner, har fået ét stykke arbejde hver og kan derfor køre "true parallel". Dette er den ønskede fordeling for at en klient får den hurtigste eksekveringstid.

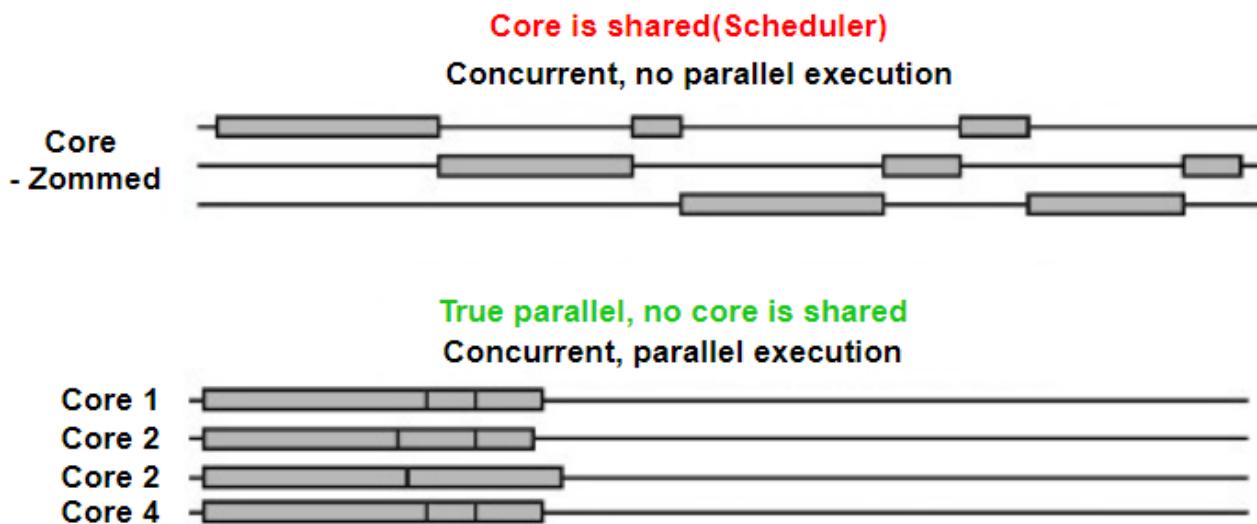


Figure 20: Concurrent vs. true parallel [58].

Med udgangspunkt i dette eksempel med fordeling af arbejde på 10 pods, kan det altså sikres at ingen pods går i idle, hvis hver Raspberry Pi maksimalt får tildelt fire arbejdsopgaver hver. Således kan hver Raspberry Pi skedulere arbejdsopgaverne parallelt og dertil blive færdig på samme tid. Den ønskede fordeling er vist på figur 21.

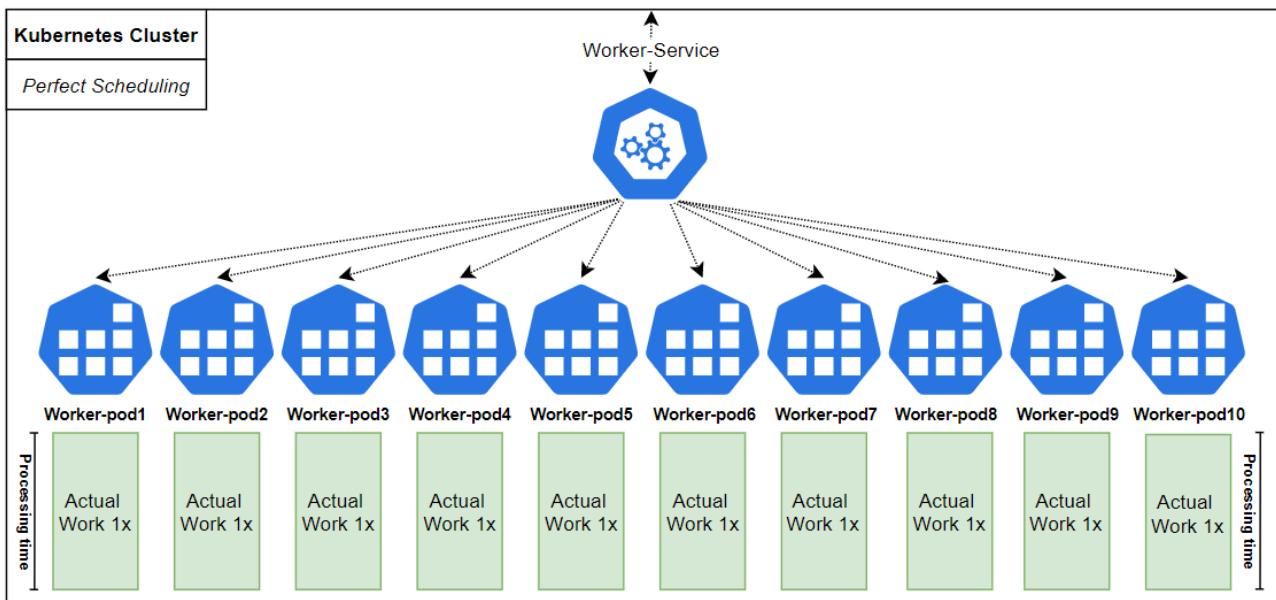


Figure 21: Optimal skedulering.

Der henvises til Load balancing designafsnit 6.1 for yderligere dokumentation/beskrivelse af hvordan disse problemstillinger er blevet håndteret/implementeret i systemet.

4.12 Firebase

Produktet har brug for persistering af data til henholdsvis brugere, data omkring jobs og API-Nøgler til brugerne af systemet. Derudover skal hjemmesiden kunne vise information omkring jobs i realtid, således at en bruger af systemet kan se med det samme at hans job er i kø, sat i gang eller færdigt. Data i realtid kan opnås på flere forskellige måder. Eksempelvis konstant polling af data, long-polling eller websockets. [29]

- **Konstant polling** er hårdt for både server og klient, da klienten laver mange requests til serveren. Det er spild af resourcer når der findes andre måder, som er mere effektive at gøre det på.
- Ved **Long polling** spørger klienten serveren om nyt data. Når der er nyt data sender serveren data tilbage, og klienten spørger med det samme igen.
- **Websockets** er mere lightweight i det klienten og serveren åbner en forbindelse som holdes åben, indtil en af parterne lukker den igen. Imens kan der sendes request begge veje hvilket er smart til reeltidsdata.

Valget for reeltidsdata ligger på Websockets. Men i stedet for at implementere det selv, så har firebase en reeltids database som gør brug af den samme teknik. Grunden til at valget ikke ligger på, selv at lave et API med dertilhørende websockets, men i stedet gøre brug af firebase er der flere grunde til.

fordeler ved firebase:

- Firebase kommer med funktionalitet til at lave accounts og logge ind, via et simpelt API.
- Der er SSL på alt kommunikation med firebase.
- Udviklingstiden er kortere.
- Firebase funktionaliteterne er gennemtestet af Google, hvilket betyder at man kan stole på det, og være mere sikker på ikke selv at lave fejl.

fordeler ved selv at lave api:

- Man har hundrede procent styr på hvad der sker.
- Det er gratis.

En stor pointe ved selv at lave et API, er at det er gratis. Dette betyder meget hvis man har mange brugere idet firebase kan blive dyrt i større systemer [9]. En vigtig pointe her, er at dette projekt blot er en prototype og derfor ikke vil lide af store brugermængder og dermed mange kald til databasen. Derudover er udviklingstiden kortere, hvilket betyder at fokus kan ligge på det primære produkt, som er NuGet-pakken og clusteret. Derfor er valget faldet på Firebase API'et som leverandør til database, reeltids data og autentifikation.

5 Arkitektur

5.1 Introduktion

Følgende afsnit er en beskrivelse af systemets softwarearkitektur. Arkitekturen er udformet ved hjælp af C4 modellen som er opfundet af Simon Brown [4]. Modellen bygger på at man zoomer ind på softwarekomponenterne, ligesom man zoomer ind på et kort. Hvert "C" står for en arkitektonisk model, hvor man i det første "C" kigger på systemet som helhed. Derefter zoomer man ind på systemets komponenter og beskriver softwaren med modeller og pile samt korte og præcise tekster. Modellen består af fire niveauer hvor hvert niveau repræsenterer forskellige zoom-levels eller kontekster. Afsnittet er derfor delt op i fire dele, et for hvert "C", samt et afsnit med supplerende diagrammer/modeller til nærmere at beskrive enkelte dele af systemet.

5.2 Systemsekvensdiagram

Figur 22 viser et overordnet overblik over systemets sekvensdiagram. Systemsekvensdiagrammet viser de interaktioner systemets aktører har med selve systemet. Det er ikke alle udfald som er medtaget, da diagrammet blot har til opgave at vise en simplificeret form. Diagrammet viser fra det tidspunkt hvor en bruger downloader CSPL NuGet-pakken, til det tidspunkt hvor han får svar/resultat fra clusteret og ser hans status på systemets hjemmeside.

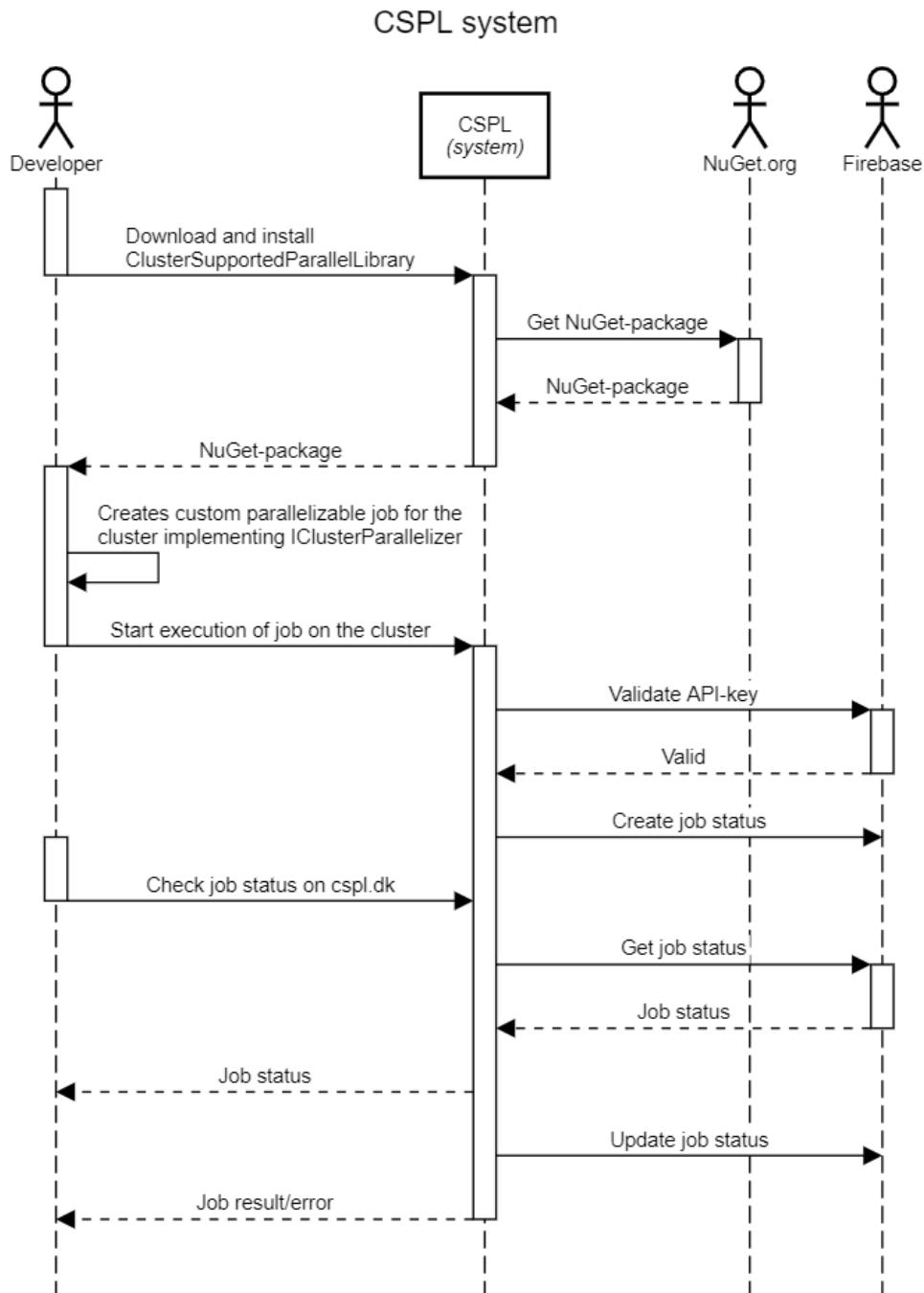


Figure 22: Overordnet systemsekvensdiagram.

5.3 Kontekstdiagram

På det første niveau af C4-modellen vises systemet oppe fra. Her er systemets afhængigheder til eksterne systemer, samt de aktører som påvirker systemet, i fokus. På figur 23 ses en kasse med navnet "Cluster system" som repræsenterer det system, som opbygges i dette projekt. Det er den kasse, som vil være i fokus i de kommende afsnit. Det ses også, at systemet "Cluster system" har to eksterne systemer som afhængigheder og en aktør i form af en udvikler, som bruger systemet. De to eksterne systemer er henholdsvis en NuGet-server, som hoster en NuGet-pakke, som er en del af det interne system. Den anden er Cloud-hosting til det interne systems hjemmeside.

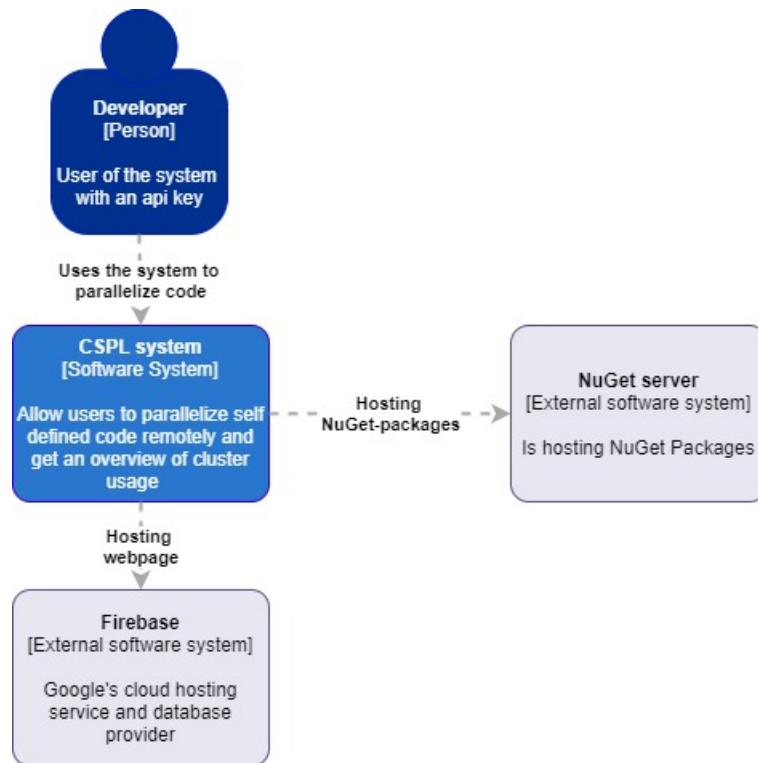


Figure 23: Context diagram, niveau ét.

5.4 Containerdiagram

Niveau to af C4 er et containerdiagram. En container kan være en database, SPA, API'er, apps, Windowsprogrammer mm. Diagrammet på figur 24 viser softwarearkitekturens high-level-komponenter, hvordan de snakker sammen og hvordan de hostes. Derudover vises det også hvordan komponenterne i det interne system, snakker med de eksterne systemer. På figur 24 ses også hvilken teknologi der bruges til at implementere og kommunikere imellem komponenter. Det ses også, at der er en lille kort beskrivelse af, hvilket ansvar hver komponent har.

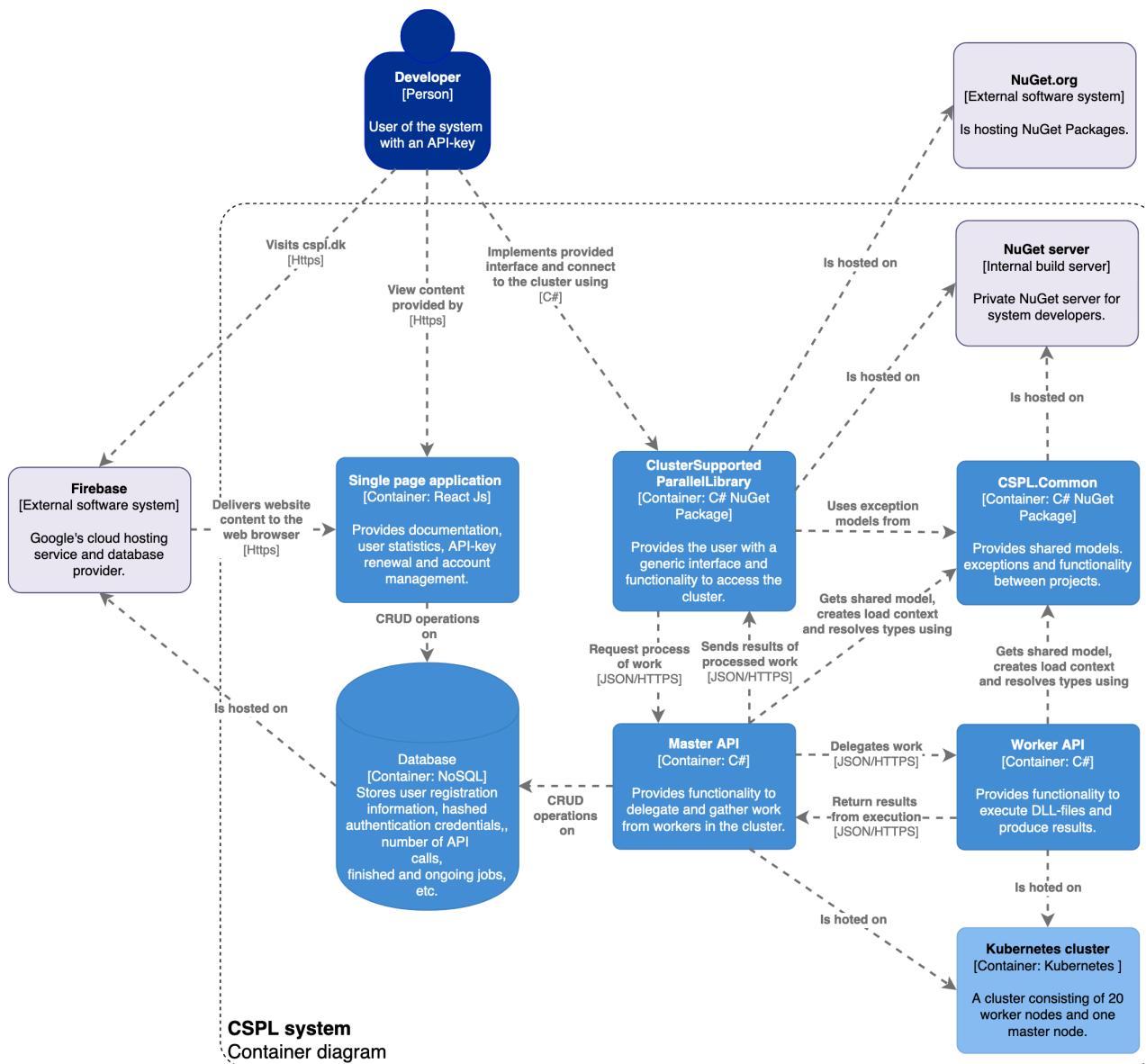


Figure 24: Containerdiagram, niveau to.

5.5 Komponent- og kodediagrammer

Det tredje niveau i C4 modellen finder man byggestenenene for hver container i systemet. Ingen er de containere som komponenterne er afhængig af taget med. På disse diagrammer finder man de klasser som skal bruges samt deres interaktioner og forhold.

På niveau fire i C4 modellen zoomes der ind på komponenterne. Her bliver komponentmodellerne mere kodenaere og der ses bla. de dertilhørende metoder (klassediagram) og yderligere afhængigheder (Code Map diagram). Disse diagrammer er autogenereret vha. redskaberne Class Diagram [35] og Code Map [37] i Microsoft Visual Studio 2019.

5.5.1 CSPL NuGet-pakken

5.5.1.1 Komponentdiagram

Komponentdiagrammet for ClusterSupportedParallelLibrary på figur 25 viser de komponenter, som ClusterSupportedParallelLibrary-containeren fra niveau to består af. ClusterSupportedParallelLibrary er den NuGet-pakke som udvikleren benytter sig af for at tilgå clusteret samt til at implementere den kode clusteret

skal eksekvere. Komponenten kommunikere til master API'et ved brug af DTO'er og derudover benyttes der delt kode fra CSPL.Common til bla. håndtering af assemblies og fejl.

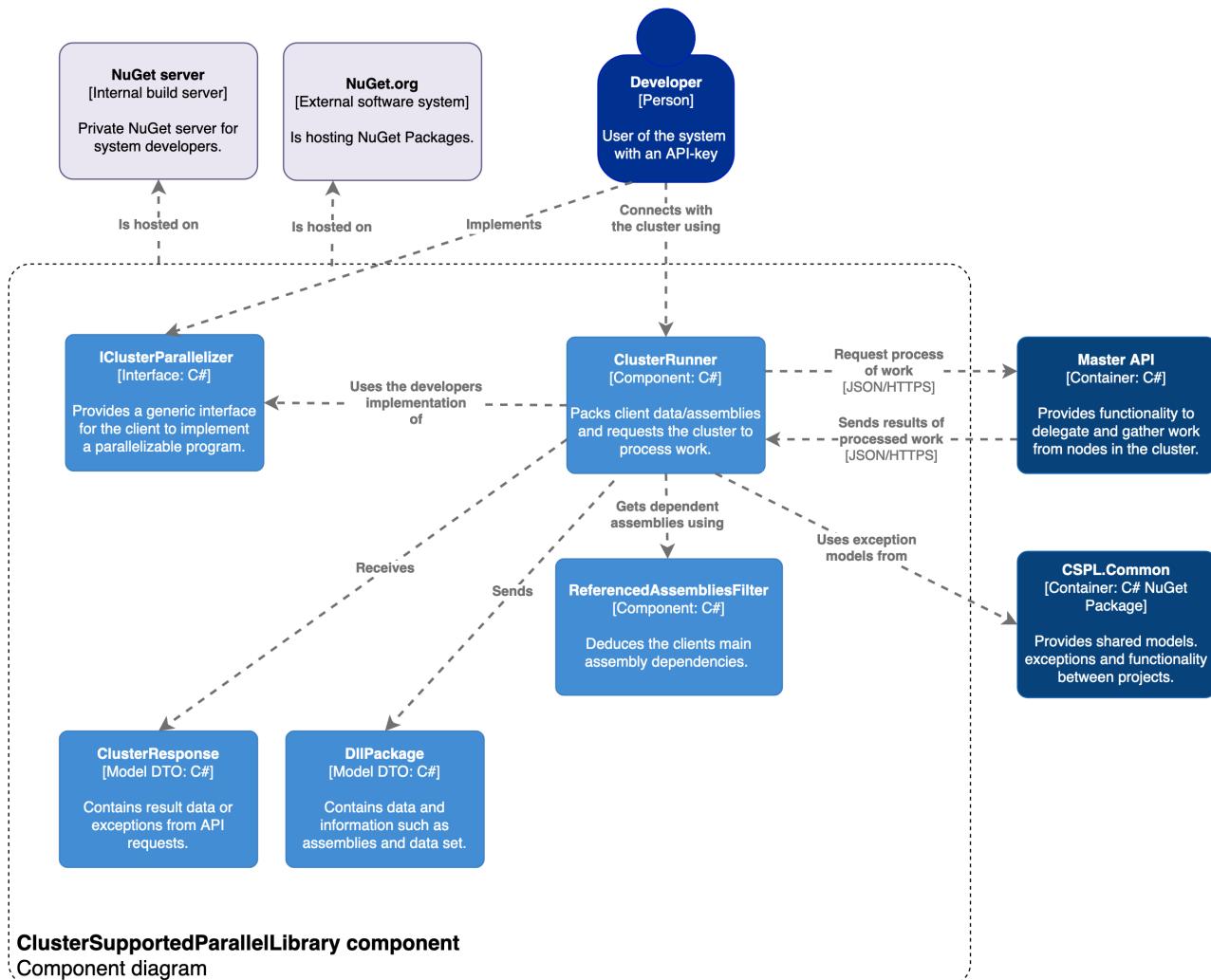


Figure 25: ClusterSupportedParallelLibrary komponentdiagram, niveau tre.

5.5.1.2 Kodediagrammer

Klassediagram

Figur 26 viser niveau fire og dermed flere kodenære detaljer om de komponenter ClusterSupportedParallelLibrary-containeren består af.

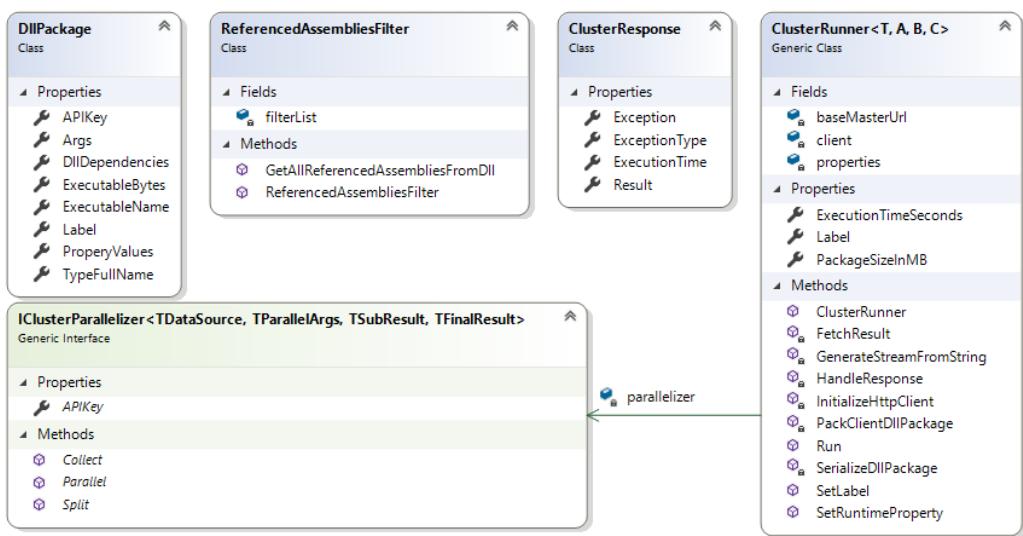


Figure 26: Klassediagram for ClusterSupportedParallelLibrary, niveau fire.

Code Map

Code Map-diagrammet på figur 27 viser de interne afhændigheder for de komponenter der udgør ClusterSupportedParallelLibrary-containeren fra niveau to.

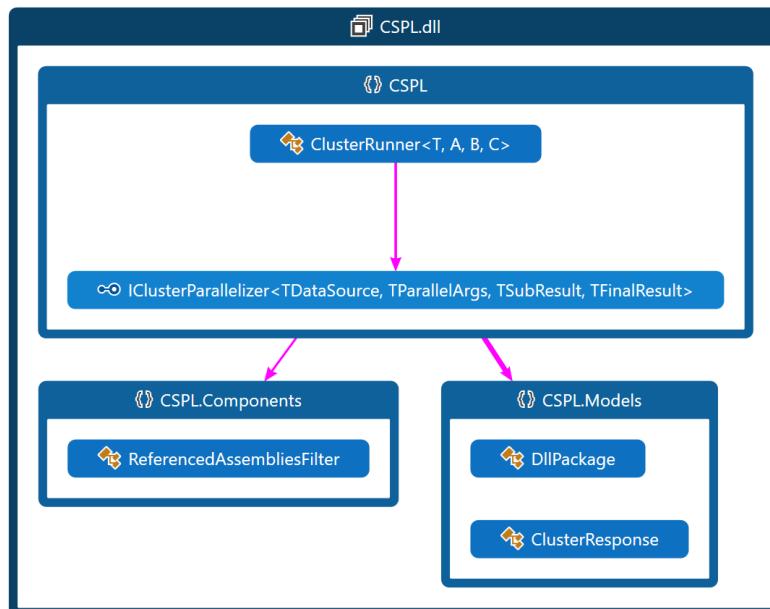


Figure 27: Code Map diagram for ClusterSupportedParallelLibrary, niveau fire.

5.5.2 Master API

5.5.2.1 Komponentdiagram

På figur 28 er der zoomet ind på master API-containeren fra containerdiagrammet fra niveau to. Mater API er det program om hostes på master-noden i clusteret. Diagrammet består af en "MasterController", som er klassen hvor API-kald fra NuGet-pakken rammer. MasterController gør brug af nogle forskellige komponenter til at udføre dens opgave.

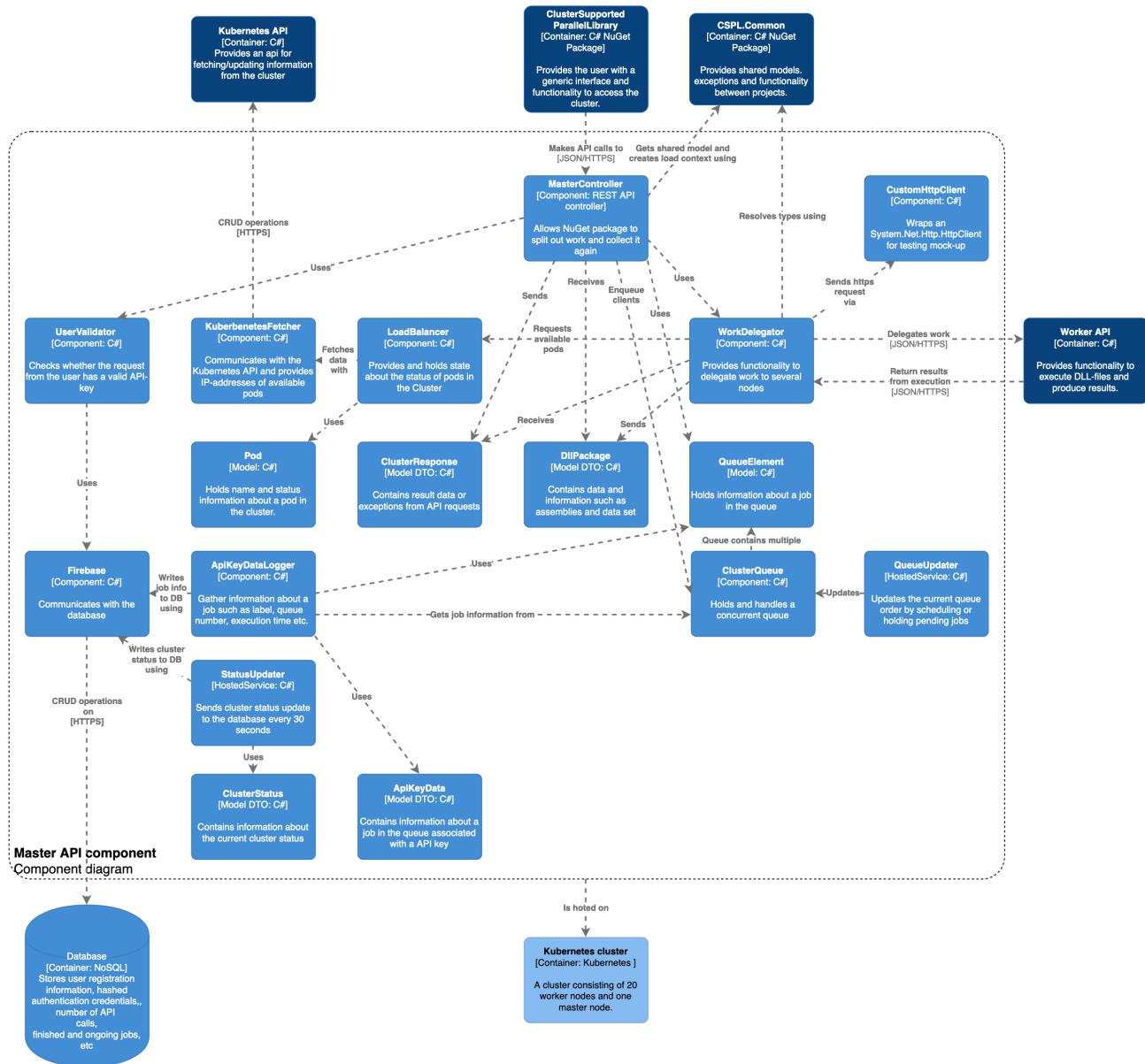


Figure 28: Master API komponentdiagram, niveau tre.

5.5.2.2 Kodediagrammer

Klassediagram

På figur 29 og 30 er der zoomet yderligere ind på master API-komponenterne fra niveau 3, de udgør et samlet klassediagram og viser yderligere detaljer for komponenterne bla. de dertilhørende metoder.

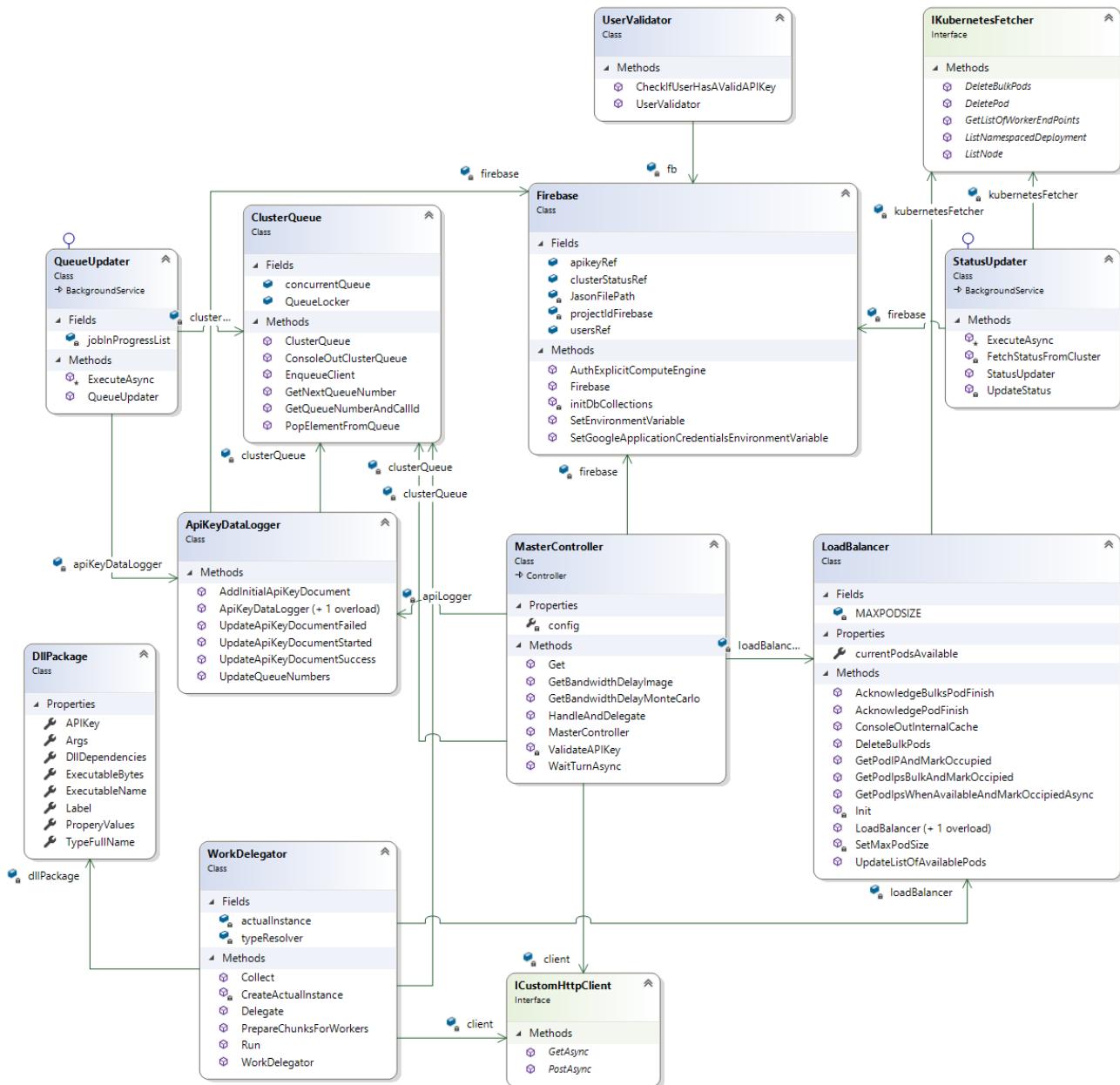


Figure 29: Klassediagram 1 af 2 for master API, niveau fire.

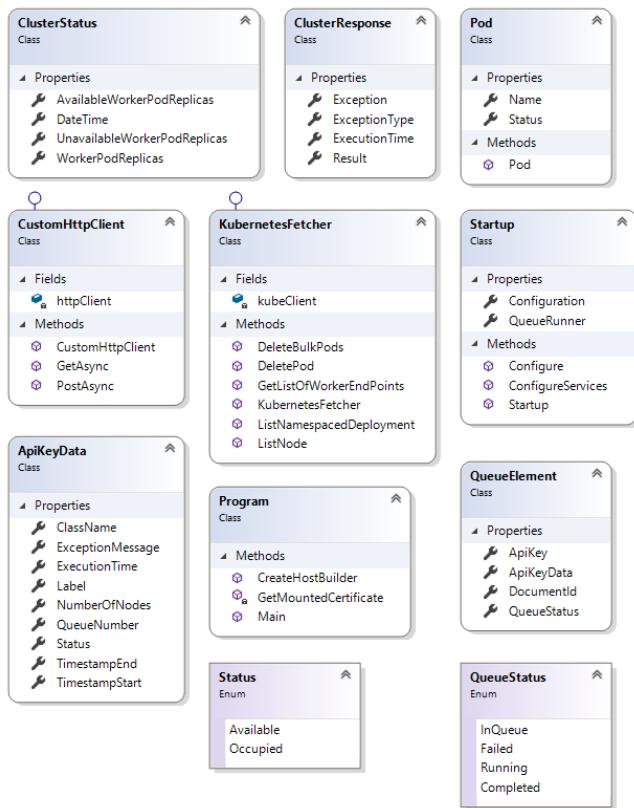


Figure 30: Klassediagram 2 af 2 for master API, niveau fire.

Code Map

Figur 31 er et Code Map, som viser de interne afhængigheder mellem komponenterne og modellerne i master API-containeren.

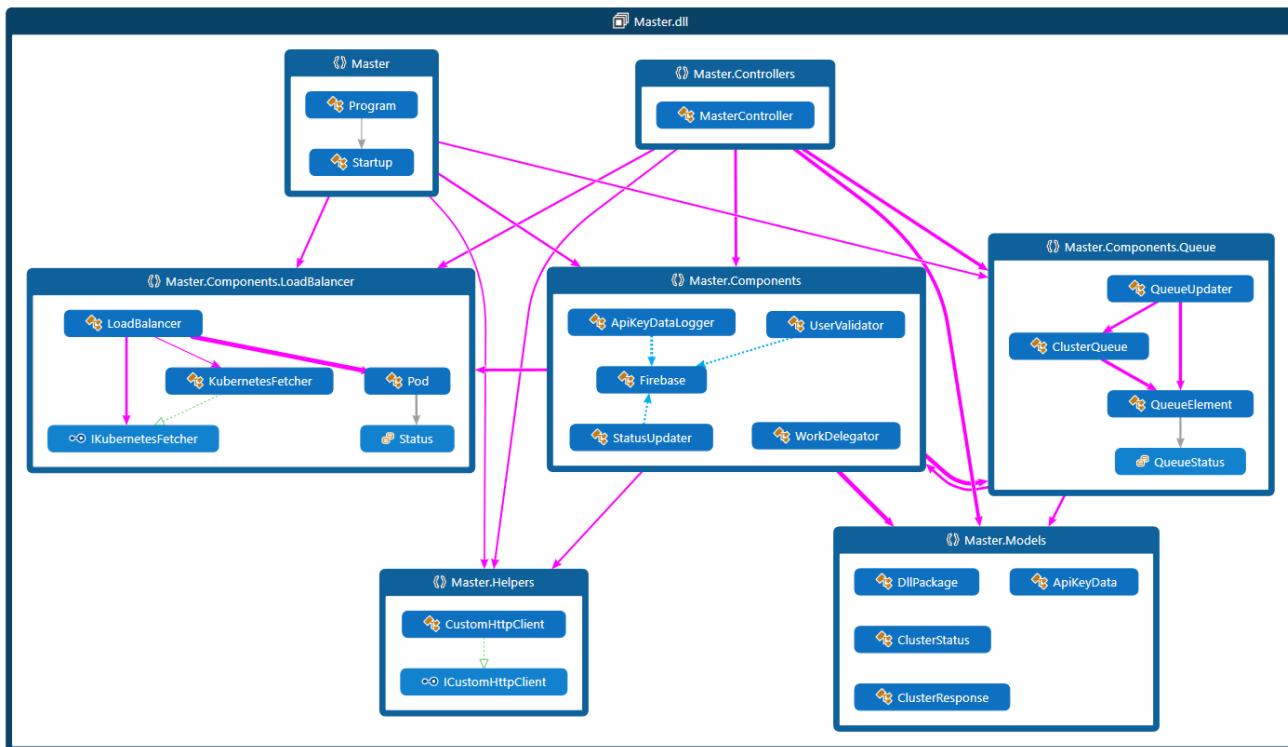


Figure 31: Code Map diagram for master API, niveau fire.

5.5.3 Worker API

5.5.3.1 Komponentdiagram

Zoomer man ind på worker API-containeren fra niveau to, ses komponentdiagrammet som vist på figur 32. Worker API'et er det program som hostes ude på worker-nodes i clusteret. Det indeholder en WorkerController som er den klasse der modtaget API-kald fra Master API'et. WorkerController'en benytter nogle DTO'er til kommunikation til Master API'et, derudover benyttes funktionalitet fra CSPL.Common til bla. loading af assemblies og fejehåndtering.

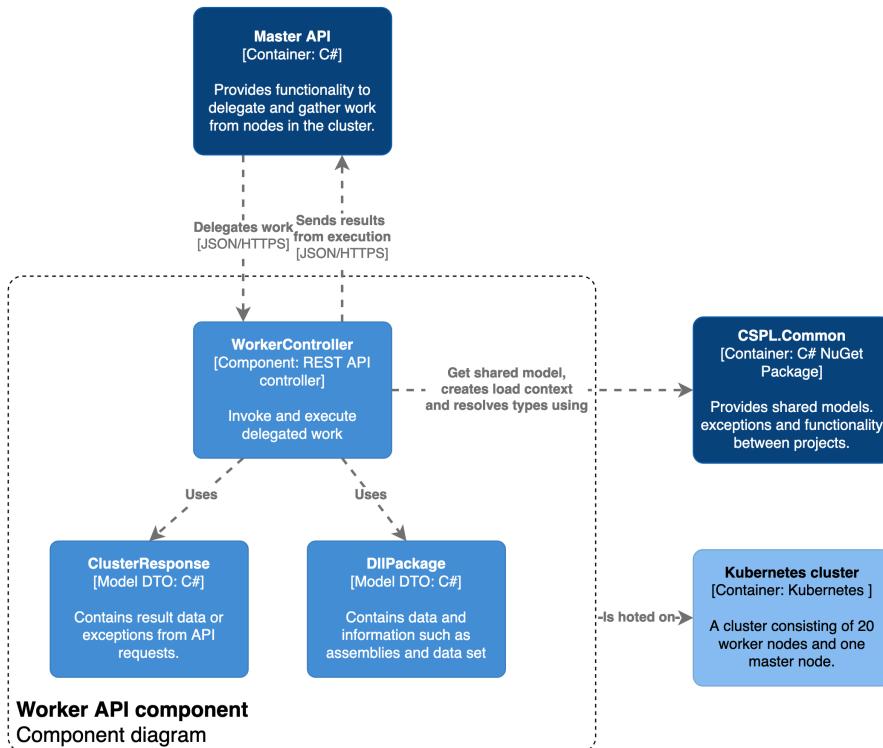


Figure 32: Worker API komponentdiagram, niveau tre.

5.5.3.2 Kodediagrammer

Klassediagram

Der er på figur 33 zoomet ind på worker API-komponenterne fra niveau tre, klassediagrammet viser de metoder der ligger i komponenterne.

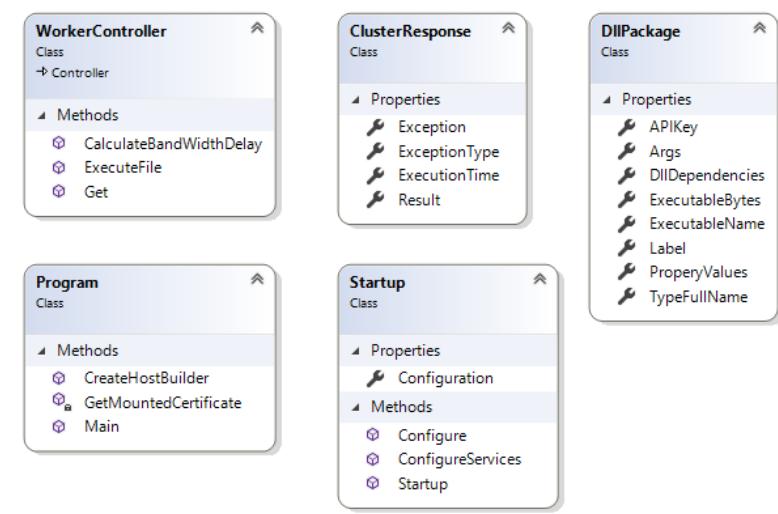


Figure 33: Klassediagram for worker API, niveau fire.

Code Map

Figur 34 udgør et Code Map som viser de interne afhængigheder mellem komponenterne.

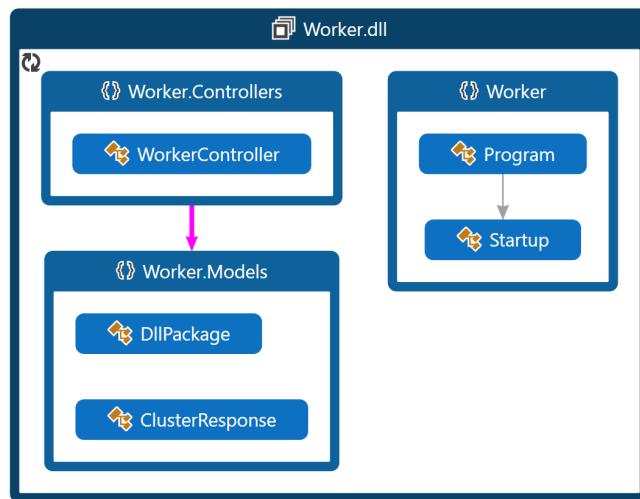


Figure 34: Code Map diagram for worker API, niveau fire.

5.5.4 CSPL.Common

5.5.4.1 Komponentdiagram

Niveau tre for CSPL.Common-containeren ses på figur 35. CSPL.Common er en NuGet-pakke som indeholder delte modeller og funktionalitet mellem containere. Komponenten benyttes bla. til at "resolve" generiske typer på runtime og loading af assemblies.

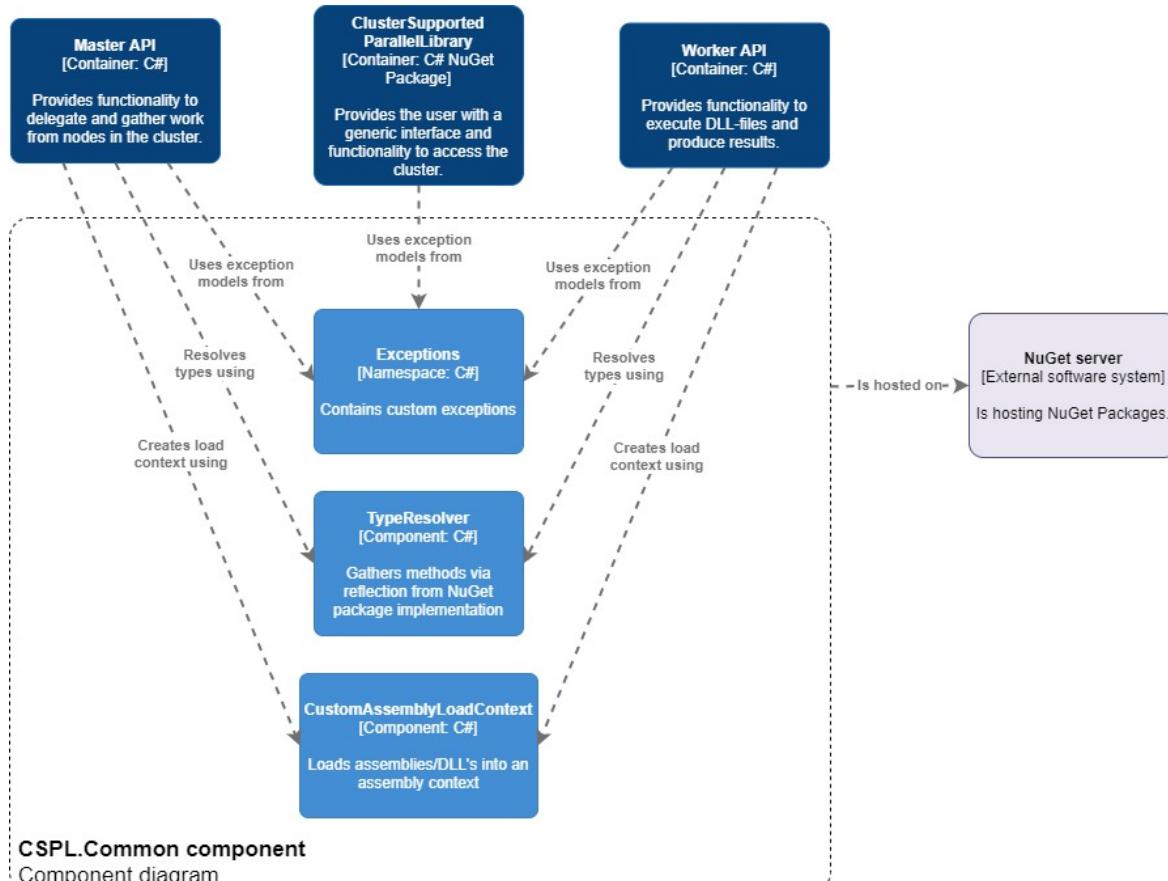


Figure 35: CSPL.Common komponentdiagram, niveau tre.

5.5.4.2 Kodediagrammer

Klassediagram

Figur 36 viser et niveau fire klassediagram for CSPL.Common-komponenterne, bla. de exception-modeller der bliver benyttet i systemet.

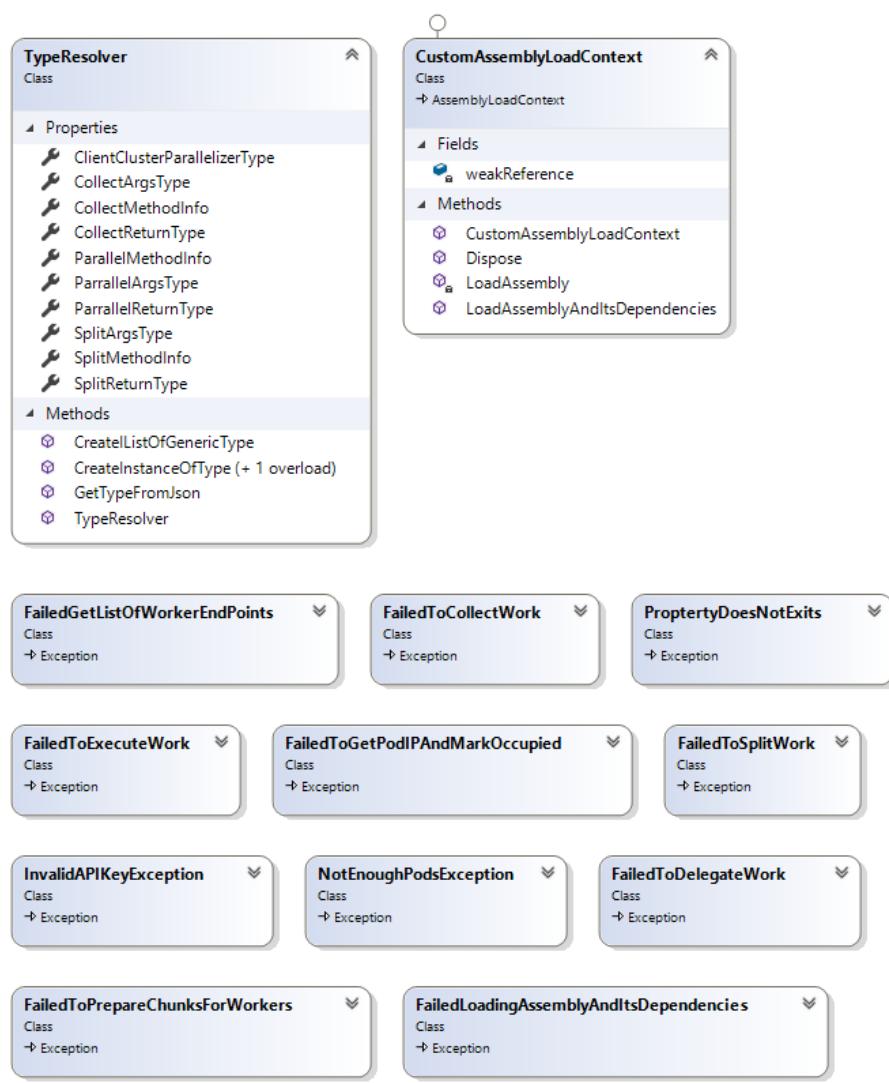


Figure 36: Klassediagram for CSPL.Common, niveau fire.

Code Map

De interne afhængigheder for CSPL.Common-komponenterne ses på Code Map'et på figur 37.

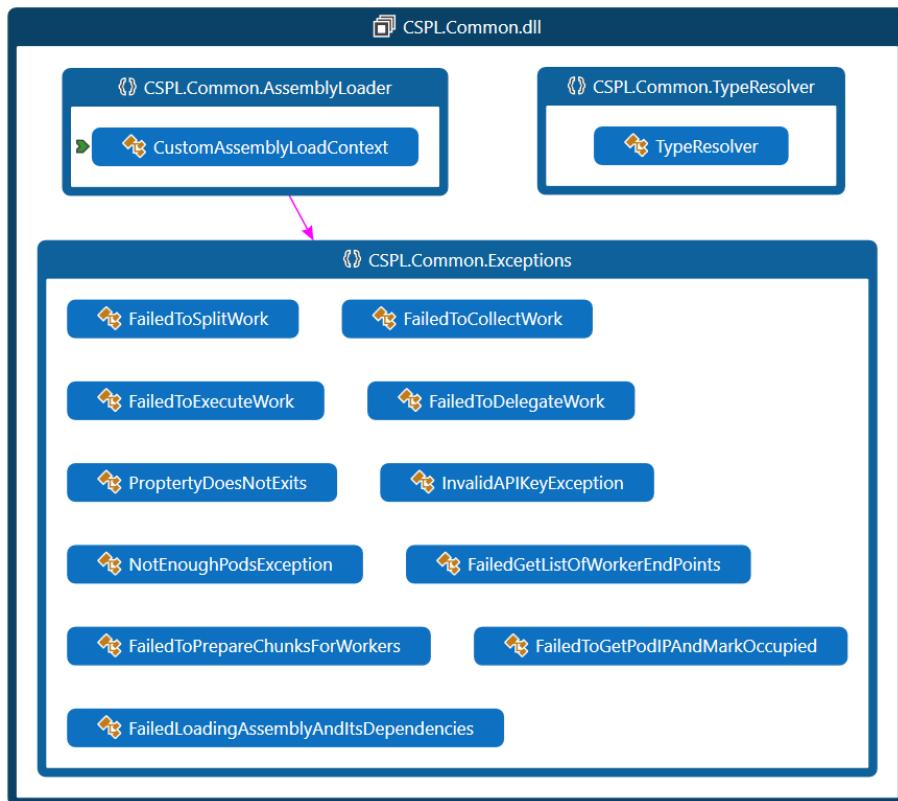


Figure 37: Code Map diagram for CSPL.Common, niveau fire.

5.5.5 Hjemmesiden

Hjemmesiden er en single page applikation skrevet i React og Javascript. Det er en lightweight hjemmeside hvor hovedopgaven er, at fremvise dokumentation for NuGet pakken og gøre det muligt at få en API-nøgle til brug af systemet. Derudover er der et basic login og create account system, samt en side hvor brugeren kan blive overbevist om at systemet virker, med grafer og tabeller til at vise systemets performance gain. Til sidst kan brugeren også kunne se data omkring de kald som er lavet til systemet samt eksekveringstid og antal af noder.

På figur 38 ses et komponent diagram for hjemmesiden. Hjemmesiden bruger Googles Firebase database "Cloud Firestore" til persistering af bruger data.

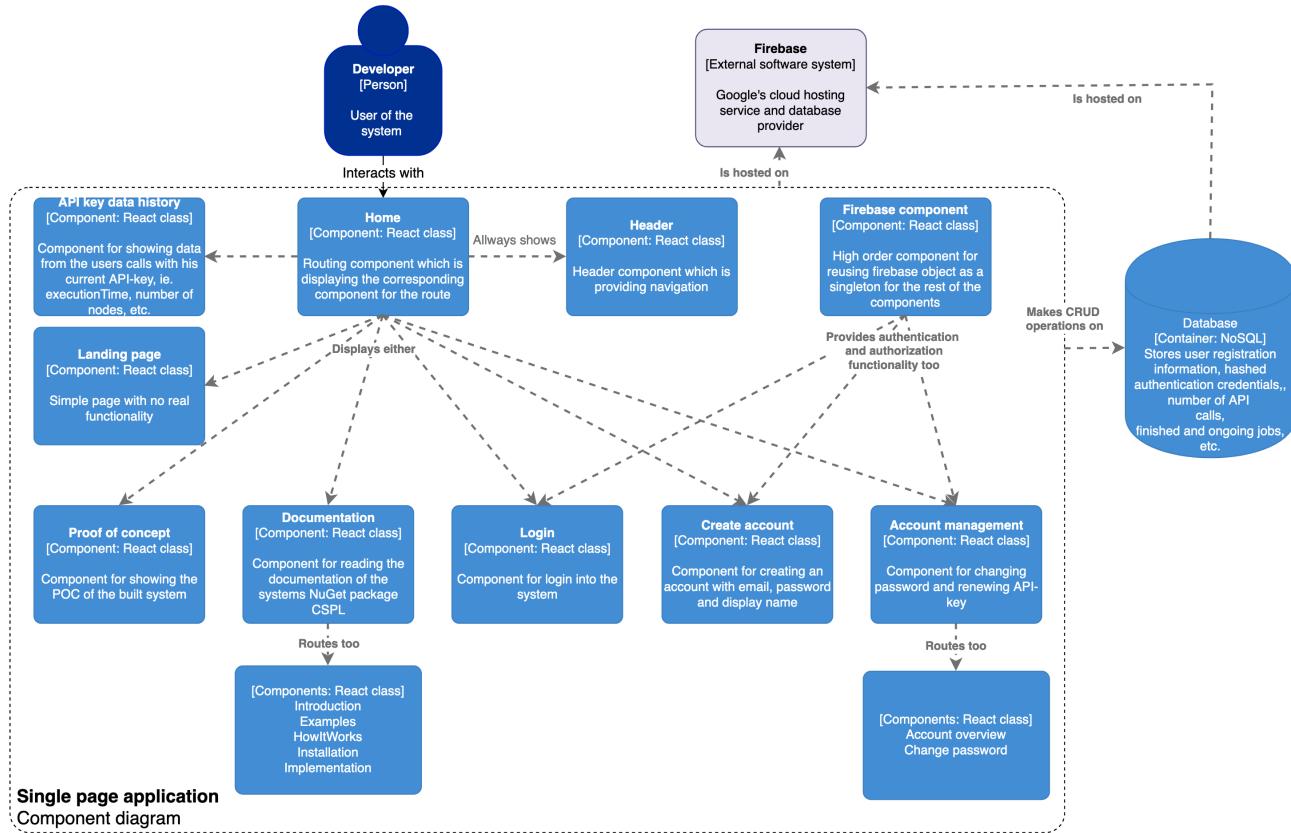


Figure 38: Webside arkitektur, niveau tre.

5.5.5.1 Home

Home komponenten er et routing komponent som renderer det komponent, som passer til den route man står på. Kasserne i midten er de forskellige komponenter som kan renderes af Home. Derudover vises header komponenten.

5.5.5.2 Firebase

Firebase komponenten er en HOC (High order component), som er et genbrugeligt komponent, som forsyner de resterende komponenter med et objekt som er en abstraktion til Googles firebase. Den indeholder funktionelitet til at snakke med firebase authentication og cloud firestore [13]. Man bruger en HOC for at alle komponenter ikke skal lave sit eget objekt af firebase klassen(Singleton principippet). Derudover bliver det indviklet at holde styr på forskellige firebase objekter, da et firebase objekt holder state, om den bruger som er logget ind. En HOC komponent er blot et komponent som returnerer et komponent med nogle specifikke properties. Komponenten holder styr på hvem der er logget ind samt nogle oplysninger omkring den person, samt hans API-key.

5.5.5.3 Dokumentation

Dokumentations komponenten forsyner brugeren med oplysninger om, hvordan systemet fungerer, bruges og implementeres. Komponenten har sin egen interne routing til forskellige sider. på figur 38 ses det at komponenten har ruter til følgende:

- **Introduction** som er en side med en kort beskrivelse af systemet
- **Installation** med en beskrivelse af hvordan man får fat i NuGet pakken.
- **Implementation** som beskriver interfacet og de forskellige funktioner som skal implementeres. Der vises også små forklarende eksempler.

- **HowItWorks** er en side hvor der beskrives teknisk hvordan systemet på et overordnet niveau, snakker med hinanden. Det er "the big picture".
- **Example** er en side hvor der vises et eksempel på det implementeret interface.

5.5.5.4 Login og create account

Login siden er en klassisk login side med et email - og password felt. Create account siden er et view hvor man indtaster email, password, repeat password og displayname. Begge komponenter gør brug af HOC komponenten firebase som er stillet til rådighed.

På figur 39 se et simplificeret sekvensdiagram over hvordan firebase benyttes fra hjemmesiden. Hoc komponenten Firebase, bruger blot en metode fra Firebase api'et med email og password. Det samme sker ved login på figur 40.

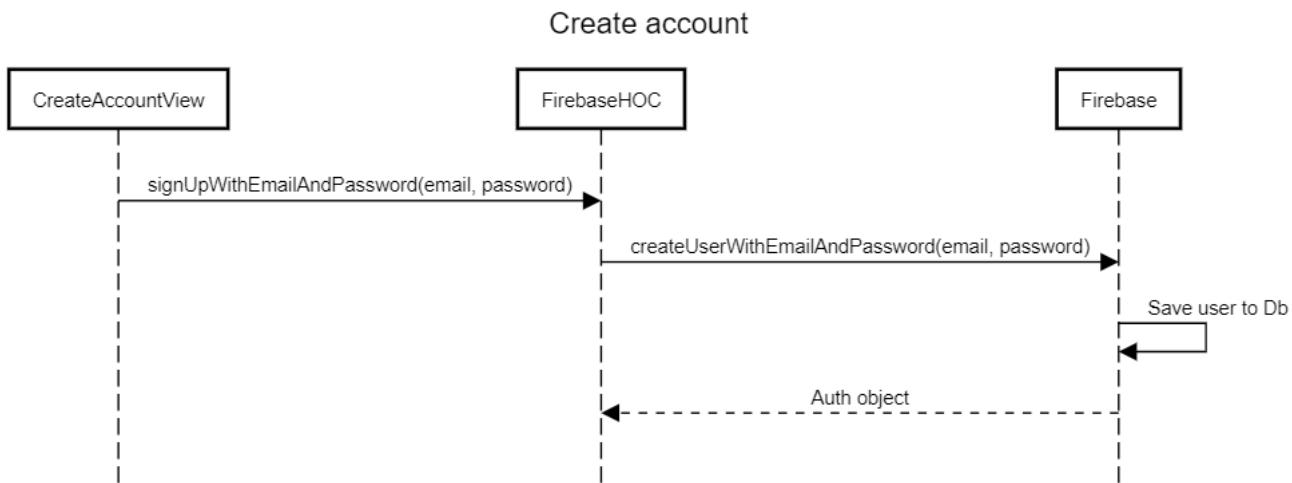


Figure 39: Sekvensdiagram af hvordan en klient lavet en account

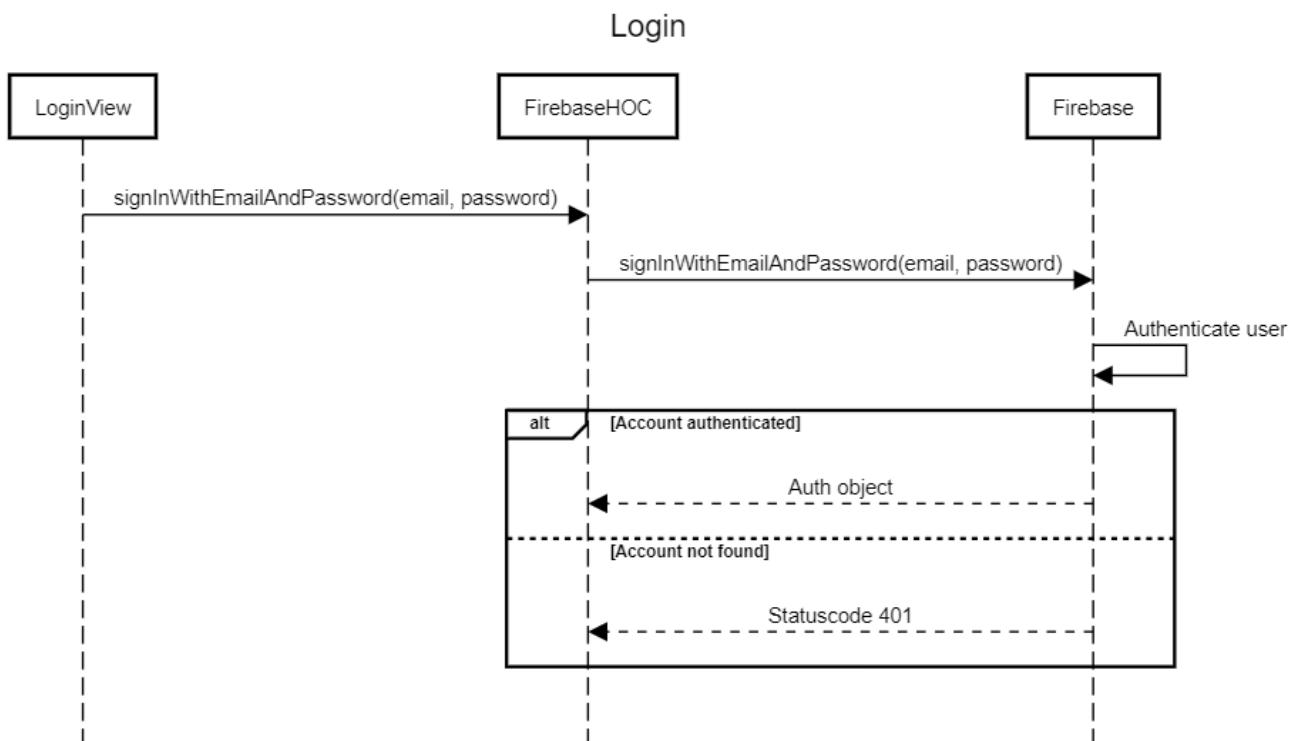


Figure 40: Sekvensdiagram af hvordan en klient logger ind på cspl.dk

5.5.5.5 Account Management

Account management er der hvor man kan få et overview over sin konto. Man kan yderligere skifte password og forny eller købe en API-nøgle. Siden har ligesom dokumentationen sin egen lille routing som ruter til de færdnævnte sider med change password mm. Account management gør også brug af firebase komponenten. Set på figur 41 er der vist et sekvensdiagram af en klient som køber en API-Key.

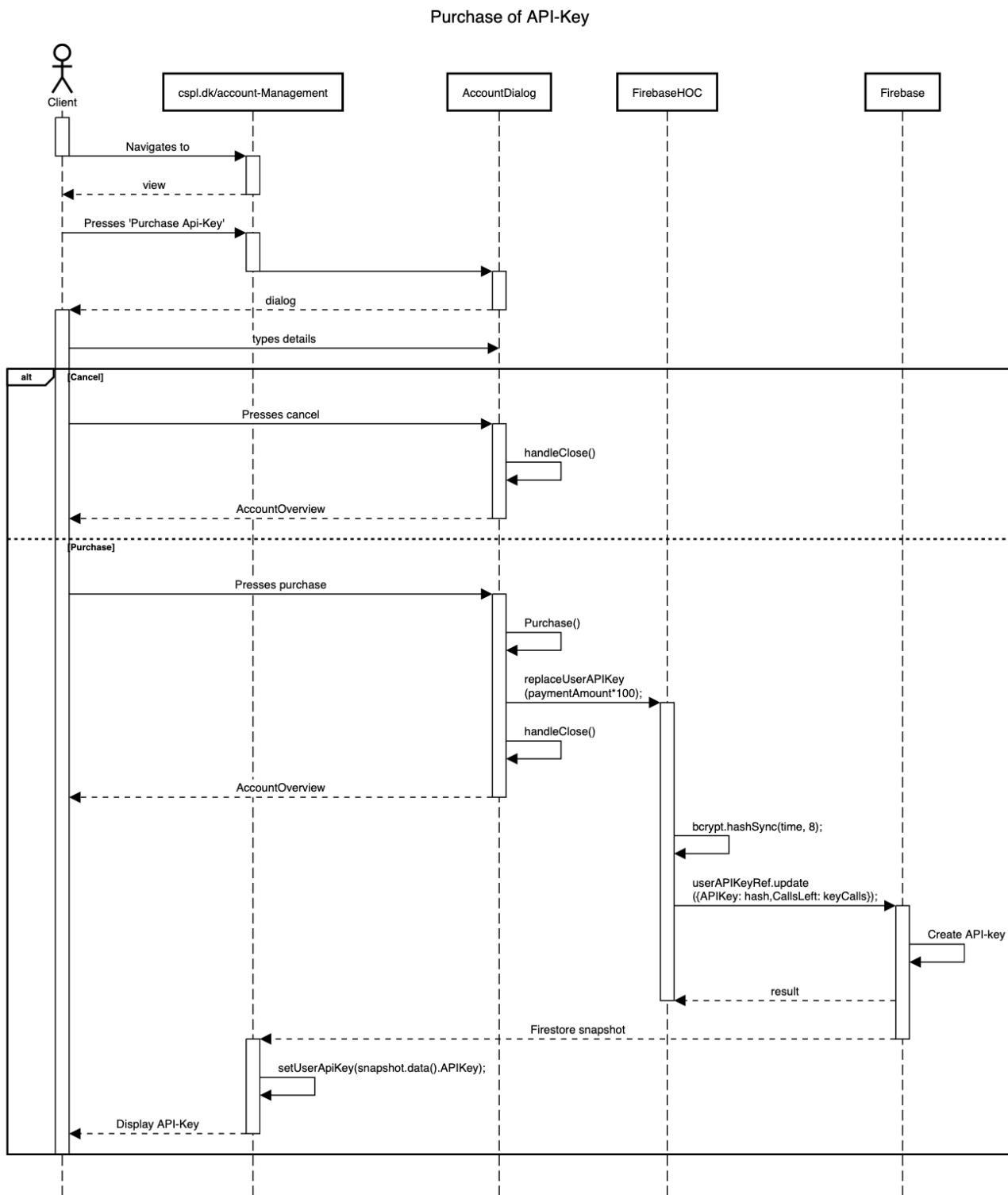


Figure 41: Sekvensdiagram af klient som køber en API-Key

5.5.5.6 API-Key History

API-Key history viewet har til formål at vise live data fra kald ind mod clusteret. De forskellige jobs bliver vidst i 4 forskellige states; in Queue, Running, Completed eller failed. På figur 42 er der vist et sekvensdiagram for hvordan API-Key history tabellen får vist det passende data.

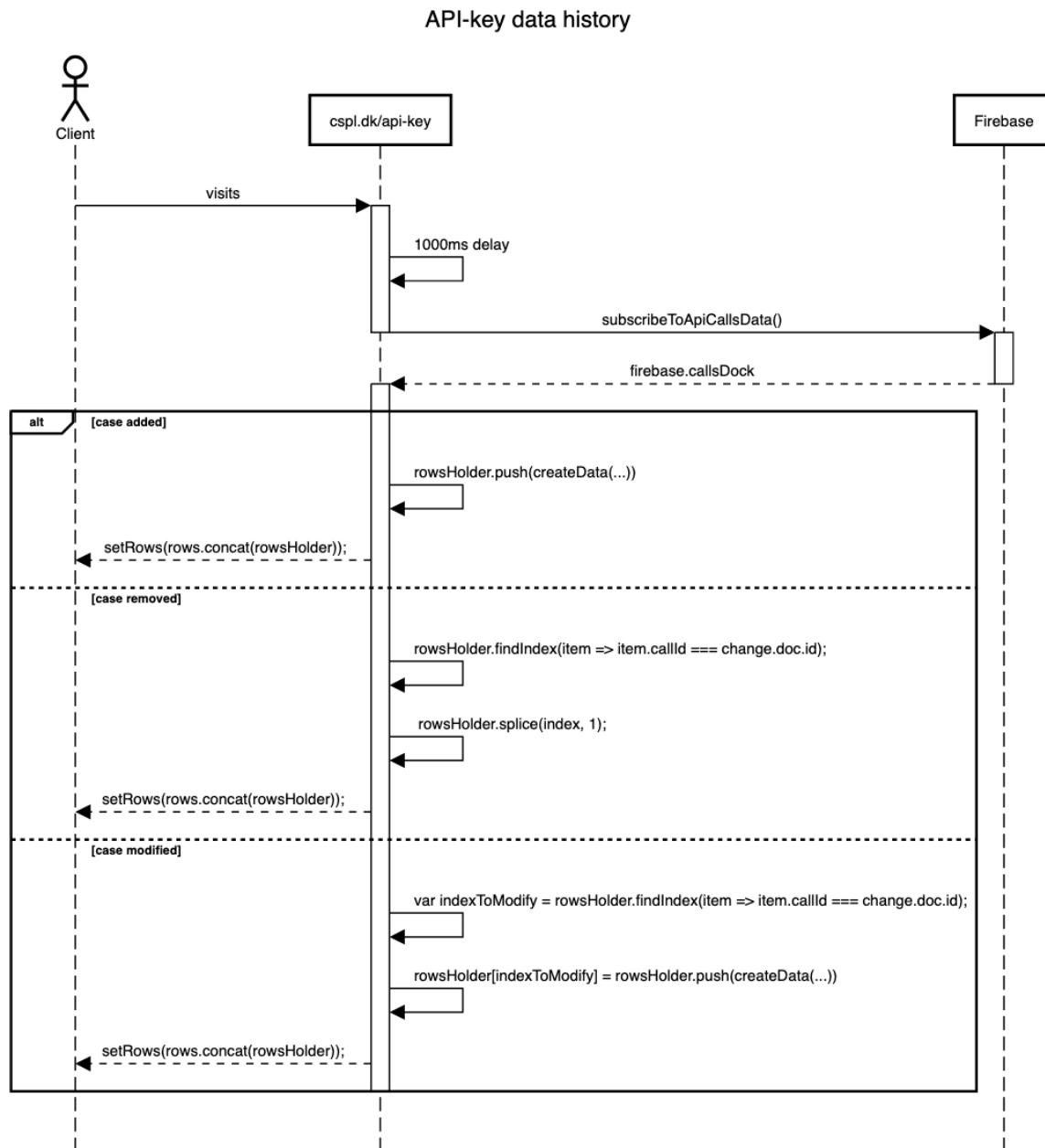


Figure 42: Sekvensdiagram for API-Key history tabel

5.6 Hosting af applikationer

Systemet består to applikationer, to NuGet-pakker og en database. I følgende afsnit gives en beskrivelse af, hvilke arkitektoniske valg der er taget i forbindelse med hosting af systemets artefakter.

5.6.1 Hjemmesiden

Der er mange forskellige muligheder til hosting af hjemmesider. To af dem er henholdsvis hosting med en tredje part service eller hosting på egen server. I dette system er der mulighed for hosting på kubernetes clusteret. I

stedet er beslutningen faldet på at hoste hjemmesiden på Googles Firebase [8]. Firebase tilbyder hosting med mulighed for gratis SSL certificering for hjemmesider med domæner. Derudover har hjemmesiden funktion alitet som skal fortælle brugeren status omkring clusteret. Det vil sige, at hvis clusteret er nede, skal hjemmesiden kunne fortælle brugeren det. Det kan hjemmesiden ikke, hvis den er hosted på et cluster som er nede. Derudover vil hosting på clusteret kræve at systemet afsætter en pod til hosting. Den pod er bedre givet ud til parallelisering af kode, for at give bedre performance. Hjemmesiden er blot support til det primære produkt som er clusteret, og derfor gives alt den CPU kræft som er til rådighed, til Master og Workers.

5.6.2 CSPL NuGet-pakken

Systemets to NuGet-pakker "ClusterSupportedParallelLibrary" og "CSPL.Common" er hostet henholdsvis på NuGet.org og en privat NuGet Server. NuGet.org hoster den publiceret "CSPL.Common" wrappet i "ClusterSupportedParallelLibrary", som brugere har adgang til, hvor den private buildserver hoster de samme NuGet pakker, men til brug for udviklere.

5.6.3 Master API

Master API'et er hostet i clusteret. Masterens loadbalancing gør brug af workernes virtuelle ip-adresser til at skedulere og uddelegere arbejde. Har masteren ikke adgang til clusteret, kan de virtuelle Ip'er ikke rammes og arbejde kan dertil ikke skeduleres.

5.6.4 Worker API

Worker API'erne er hostet i clusteret. Det er der for at kunne gøre brug af kubernetes orkestrations services som eksempelvis *restart policy* [26], som gør at pods restarts hvis de crasher. Derudover er det let at skalere antallet af replicas.

5.7 Kommunikation mellem frontend og Master

I og med at firebase er valgt som komponent til at imødekomme kravet om data i realtid, så forgår kommunikationen imellem frontend og master ved subscription på data i tabellerne i Cloud Firestore databasen. Et sekvensdiagram som viser dette flow ses på figur 43.

frontenden subscriber på noget data, som ved modificering fra master, tilføjelse eller sletning, notificere frontenden og opdatere viewet.

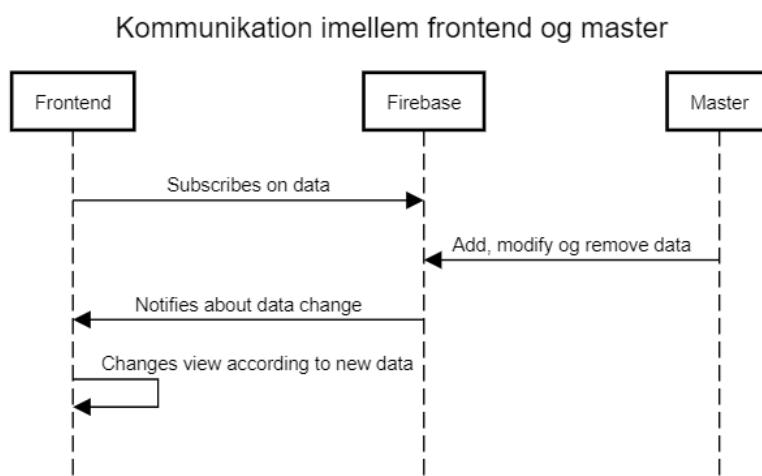


Figure 43: Kommunikation imellem frontend og master

5.7.1 Hardware oversigt

Set på figur 44, er der vist en oversigt over alle hardware komponenter der danner grundlag for CSPL clusteret. Clusteret får internet igennem et Technicolor modem, hvor routeren i kombination med switchen giver

internet til systemet. Master modtager klient requests om arbejde, hvor master uddeleger arbejdet videre til Raspberry Pi clusteret. En mere detaljeret forklaring om hvert komponent henvises til afsnit 4.1.

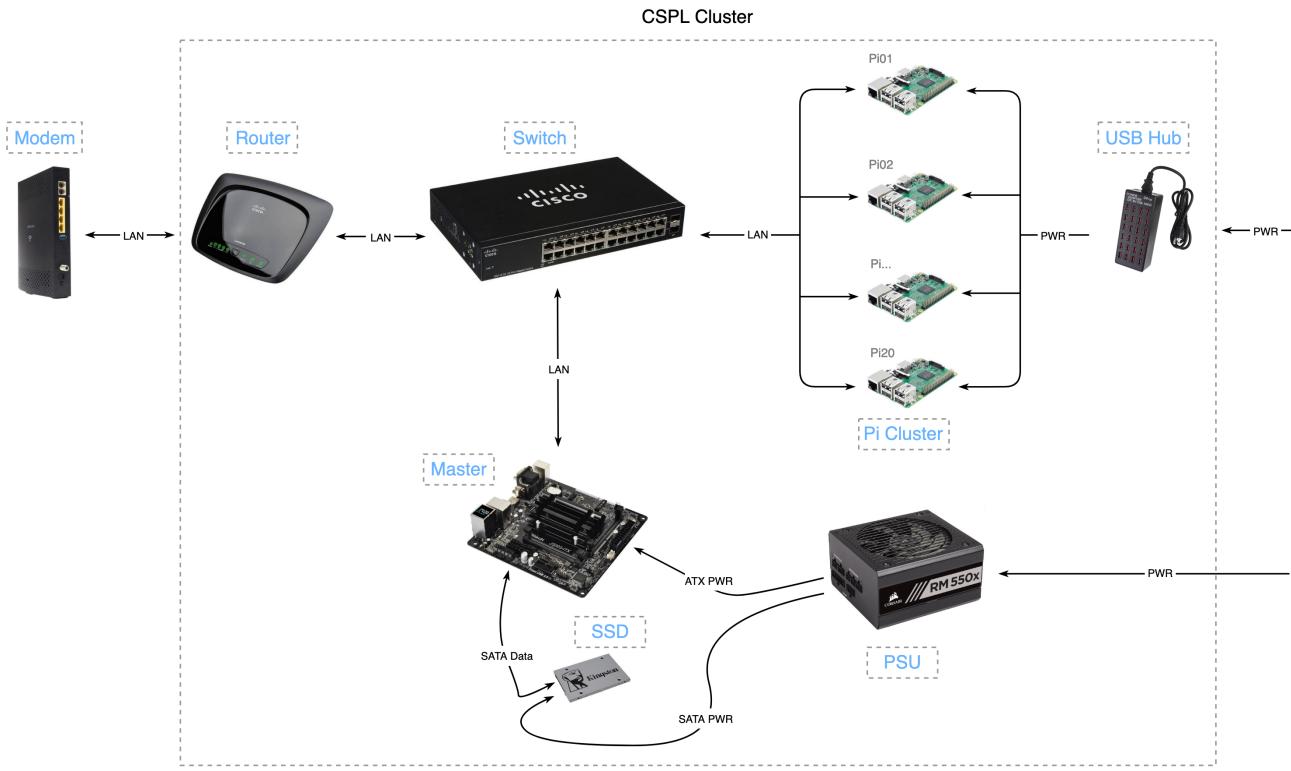


Figure 44: Oversigt over HW. komponenter for CSPL

6 Design

6.1 Load balancing

Dette afsnit tager udgangspunkt i løsninger på problemstillingerne omkring distribuering af arbejde til workers som er beskrevet i afsnit 4.11.

Da det umiddelbart ikke er muligt at konfigurere worker-servicen, som vælger tilfældige pods, til altid ligeligt at fordele arbejde mellem kerner, kan denne funktionalitet løses/kodes vha Kubernetes API'et. Hvis de rigtige rettigheder er til stede, kan der på runtime konfigureres Kubernetes indstillinger/elementer via Kubernetes API'et. Hertil kan worker-servicen modificeres til ikke at skedulere arbejde til pods, som allerede har fået et stykke arbejde tildelt.

6.1.1 Løsning 1 - Label switching

En mulighed for at løse dette kan være ved brug af labels i kubernetes. Labels er meget simple, men kan have en stor funktionalitet. Worker-servicen vælger hvilke worker-pods den skal være reverse proxy for ud fra en label-selector. Label-selectoren i dette tilfælde vil være "status: free" på worker-servicen, således at den kun vil vælge pods med dette label. Hvis en pod er ledig, vil den have et label "status: free" og hvis den ikke er ledig, vil den have label "status: occupied". Disse labels skal altså på figur 45 ses det hvorledes worker-servicen vælger de pods som har label "status: free".

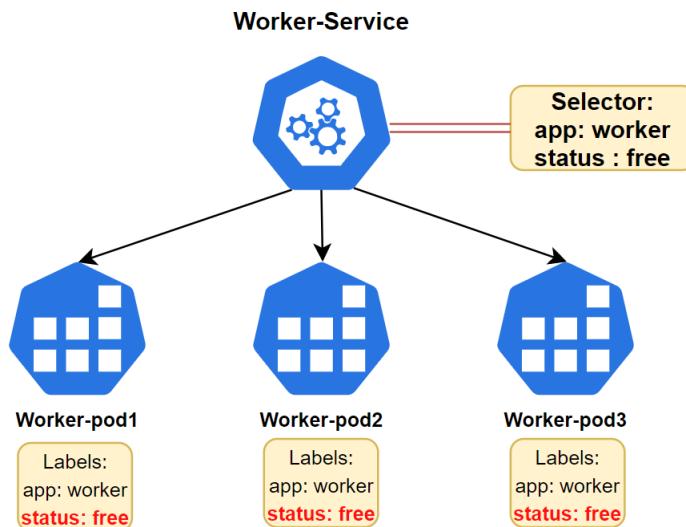


Figure 45: Worker-service med label-selector, tre pods er ledige.

Når en worker-pod har fået tildelt et stykke arbejde, skal den kontakte Kubernetes API'et for at sætte dens label til at være "status: occupied". Dette gør at worker-servicen dermed ikke har mulighed for at videresende til en pod der ikke er ledig, se figur 46.

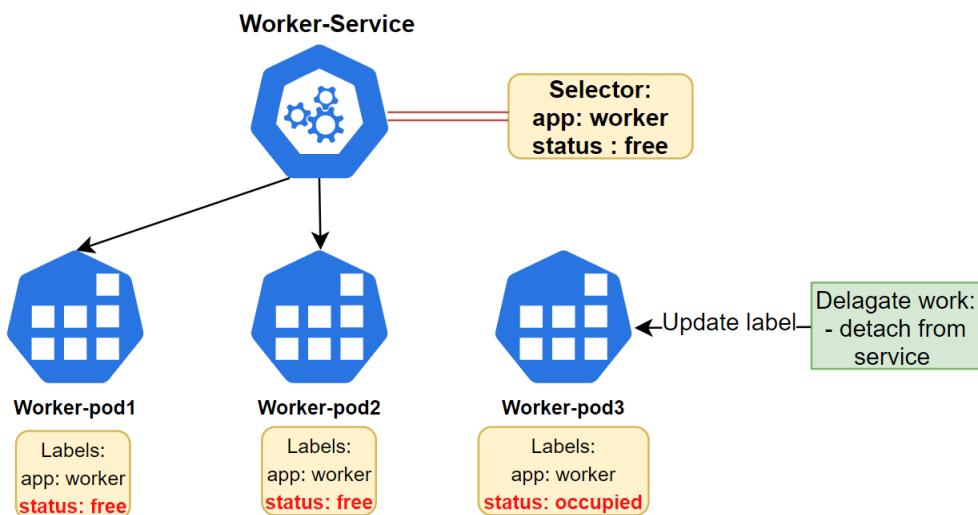


Figure 46: Worker-service med label-selector, to ledige pods og en ikke ledig pod.

- **Pros:**

- Sikring af ligeligt fordelt arbejde.
- Statisk endpoint (intern kubernetes DNS) for master at kontakte workers med.
- Hvis pods dør pga. node-restart eller at kubernetes flytter dem til en anden node, vil de aldrig blive genoplivet. Der vil altid starte en ny frisk pod, med en ny IP. Selvom dette sker vil worker-servicen altid være opdateret med de nyeste pod IP'er, automatisk.

- **Cons:**

- Hvis der skal requestes 10 worker-pods gennem worker-servicen skal der mellem hvert request ventes på at master-pod'en har modtaget et acknowledge, for at sikre at worker-pod status-label er skiftet fra "status: free" til "status: occupied". Hvis der ikke ventes på dette acknowledge, er der risiko for at sende data til en ikke ledig pod ("status: occupied").

- Det tager tid at skifte label for en worker-pod.
- Master bliver sekventiel nød til at vente på acknowledge en efter en, imens den uddeler arbejde.

I og med at master sekventiel bliver nød til at vente på et acknowledge, imens den uddeler arbejde, er denne løsning dyr ift. performance. Efter nogle tests har det vist sig at skiftning af label, plus et acknowledge, kan tage op til 500ms pr. worker-pod. Hvis en klient requester at få 10 worker-pods, vil der være følgende performance tab for uddelegering af arbejdet:

$$DelegationDelay = NumberOfPods * LabelDelay \quad (2)$$

$$= 10 * 500ms \quad (3)$$

$$= 5 \text{ sec} \quad (4)$$

6.1.2 Løsning 2 - Custom load balancer

En anden løsning vil være at lave en custom load balancer, for at sikre selv at kunne definere hvorledes arbejde skal fordeles på worker-pods. Load balanceren skal holde en cache i memory, som skal holde styr på hvilke virtuelle worker-pod IP'er som er i live og om de er ledige. Grundet workerens specielle livs cyklus i clusteret skal registreringen og fjernelse af workers, ske med præcision, hvilket gøres via. matematiske mængdeoperationer.

6.1.2.1 Worker pod's livscyklus i Kubernetes

En pod består af én eller flere containere (typisk én). Pod'en er indkapslet i et virtuelt netværkslag, således at der kan kommunikeres internt imellem pods i Kubernetes clusteret over det virtuelle netværkslag. En pods virtuelle IP er statisk i hele dens levetid. Hvis en pod enten crasher, flyttes eller hvis noden hvorpå den kører genstarter eller crasher, så dør pod'en. Kubernetes vil automatisk starte en helt ny frisk pod, som vil have en anden virtuel IP [26].

6.1.2.2 Custom Loadbalancer Cache

I stedet for at masteren skal snakke igennem worker-servicen for at få en worker-pod, skal den nu snakke direkte til en worker-pod via pod'ens virtuelle IP, som kun eksisterer internt i clusteret. Kubernetes holder styr på hvilke worker-pod IP'er der er i live, f.eks. hvis nogle worker-pods flyttes til en anden node eller genstarter. Load balanceren kan kontakte Kubernetes API'et og udfra det data som bliver returneret kan man finde information om alle worker-pods virtuelle IP'er. Load balanceren kan ud fra den information registrere de virtuelle worker-pod IP'er i memory og dermed holde styr på hvilke worker-pods som er "free" eller "occupied". Load balanceren bliver nød til at holde sig selv opdateret i forhold til det data Kubernetes API'et returnere, da det altid er Kubernetes API'et, der har den nyeste status for clusteret. Derfor bliver load balanceren nød til at opdatere sin cache i memory, hver gang den har modtaget et nyt request fra en klient. Når load balanceren skal opdatere cachen, kan der siden sidst være kommet nye worker-pods til eller der kan være nedlagte worker-pods, load balanceren skal opdatere cachen derefter.

6.1.2.3 Håndterering af døde worker-pods

For at detektere døde worker-pods i cachen, kan der ud fra det nyeste data fra Kubernetes API'et, laves mængdeoperationen **difference**. Beregning af døde workers pods, ses i equation 5.

Notation:

Lad **A** være den nuværende cache og lad **B** være det nyeste information fra Kubernetes API'et, endelig lad **C** være mængden af døde worker-pods.

$$C = A - B \quad (5)$$

Eksempel:

Det antages at cachen A indholder tre virtuelle worker-pod IP'er, og at det nyeste datasæt fra Kubernetes API'et B, indholder en ny pod samt en pod der ikke findes mere. Døde pods kan findes ved at tage alt der ligger i A som ikke ligger i B:

$$A = \{10.244.1.100, 10.244.1.101, 10.244.1.102\} \quad (6)$$

$$B = \{10.244.1.101, 10.244.1.102, 10.244.1.103\} \quad (7)$$

$$C = A - B \quad (8)$$

$$C = \{10.244.1.100\} \quad (9)$$

Alle døde worker-pod IP'er fjernes fra LoadBalancerens cache.

6.1.2.4 Håndterering af nye worker-pods

Ligesom at der kan bruges mængde-operationen **difference**, for at finde døde worker-pod IP'er, kan der også bruges **difference** til at finde nye worker-pod IP'er. Dette gøres omvendt ift. at finde døde worker-pods.

Notation:

Lad A være den nuværende cache og lad B være det nyeste information fra Kubernetes API'et, endelig lad C være mængden af nye worker-pods.

$$C = B - A \quad (10)$$

Example:

Det antages at cachen A indholder tre virtuelle worker-pod IP'er, og at det nyeste data fra Kubernetes API'et B, indholder en ny pod samt en pod der ikke findes mere. Nye pods kan findes ved at tage alt der ligge i B som ikke ligger i A:

$$A = \{10.244.1.100, 10.244.1.101, 10.244.1.102\} \quad (11)$$

$$B = \{10.244.1.101, 10.244.1.102, 10.244.1.103\} \quad (12)$$

$$C = B - A \quad (13)$$

$$C = \{10.244.1.103\} \quad (14)$$

Alle nye pods registreres i cachen med status "free".

- **Pros:**

- Sikring af ligeligt fordelt arbejde.
- Det er hurtigt hente nyt data, da en acknowledge-sekvens ikke er nødvendigt.
- Load balanceren kan understøtte at der kan requestes worker-pod IP'er i bulks.
- Ingen sekventiel ventetid.
- Load balanceren kan holde styr på fejl-scenarier, når der f.eks. ikke er flere worker-pods ledige.
- Det er nemt at holde cachen opdateret via set operationer som **difference** [50].

- **Cons:**

- Cachen skal vedligeholdes således den altid er up-to-date.
- Der skal integreres mere med load balanceren i forhold til worker-servicen.

6.1.2.5 Valgte løsning

Grundet at custom load balancing (se løsning 2 afsnit 6.1.2) understøtter kontrolleret uddelegering af arbejde og dermed resultere i en bedre performance, er dette den valgte løsning. I og med at løsning 1 (se afsnit 6.1.1), kræver en sekventiel acknowledgement og dette dermed påvirker performance negativt i form af ventetid, blev denne løsning nedprioriteret. Custom load balancing er hurtigere og kan sende requestes i bulks, derudover kan load balanceren bla. holde styr på scenarier hvor alle pods er optaget, dette er oplagt i forhold til design af køstrukturerne i systemet, se afsnit 6.3. Det er også oplagt at load balanceren kan holde styr på de eksakte pods der er blevet benyttet ift. oprydning af worker-pods, se afsnit 6.2.

6.2 Forurening af pods i clusteret

I og med der ikke er særlig stor kontrol af, hvad en klient kan få eksekveret af kode i clusteret, er det nødvendigt at tage stilling til evt. forurening af master- og worker-pods. Klienten sender egne assemblies til clusteret og da disse loades og eksekveres på både master- og worker-pods, kan der nemt opstå forurening af de containere API'erne bliver hostet i. Dette kan forårsage at pods crasher og dermed mister state/data, dette kan clusteret ikke håndtere. I forsøg på at håndtere denne problemstilling, er der blevet udarbejdet nogle forskellige funktionaliteter. Først og fremmest er der blevet designet et komponent, som rydder op i de assemblies som loades ind i programmerne, se afsnit 7.7.1. Ligeledes er der udviklet funktionalitet, der nedlægger worker-pods efter brug, således at hver gang en klient får tildelt en worker-pod, er denne pod helt frisk og har ikke være i brug før, se afsnit 7.7.2.

6.3 Køstruktur

Når klienter requester systemet om at få x-antal workers, kan der opstå en situationen, hvor der på det tidspunkt ikke er nok workers tilgængelig. Da systemet primært står for at optimere, er det ikke ønsket at komme i kø, da køen vil forsinke klientens arbejde. Dog kan der på tidspunkter alligevel opstå scenarier hvor der vil blive modtaget forskellige requests fra klienter, hvor der ikke vil være nok workers til at uddeleger alt arbejdet. Det kan for eksempel være, at en klient requester 40 workers, men hvor der på det tidspunktet kun er 20 pods ledige i clusteret, hertil er det nødvendigt at supportere en køstruktur. Hvis en klient requester med en API-key der ikke er valid, bliver klienten ikke sat i kø og får med det samme en fejlmeddeelse tilbage om at klientens API-key ikke er valid.

6.3.1 Hvornår får klienter tildelt workers?

Køstrukturen er designet som en FIFO-kø (First in first out), således hver klient får et nummer, som indeholder det antal workers de requester. På den måde kan systemet regne ud hvornår det kan blive dens tur. Der kan argumenteres for at klienter som requester små antal af workers kan få lov til at komme foran i køen, hvis ikke de forsinke de forreste i køen. Hvis der er 20 workers ledige i clusteret og den forreste klient i køen ventet på 60 workers og klienten bag ved skal bruge 10 workers, kunne denne klient godt få tildelt de workers, som den skal bruge, i og med at antallet er ledigt. Dette kan dog være et problem, da det først og fremmest udsætter arbejdet for den der ligger forrest i køen. Derudover kan der opstå starvation, hvis der hele tiden kommer klienter, som requester små antal af workers, og dermed "snyder" foran den klient, der ellers ligger forrest i køen - den forreste i køen kan risikere aldrig at blive taget ud af køen. Køen er derfor designet således at den blokerer, hvis den klient som ligger forrest i køen requester flere workers end der er ledige. På figur 47 ses en illustrering af køstrukturen, det kan ses at der ligger tre klienter i ClusterQueue'en med henholdsvis worker requests

(*Workers* : 60, *Nr* : 1), (*Workers* : 10, *Nr* : 2) og (*Workers* : 10, *Nr* : 3). Det ses også at 48 workers er optaget, og 32 er ledige. Da den forreste i køen requester 60 workers, kan den ikke få de workers den requester, og vil derfor vente på at 60 workers bliver ledige. Først når et af de jobs der ligger InProgress er færdigt, vil workers blive ledige for den forreste klient i ClusterQueue'en.

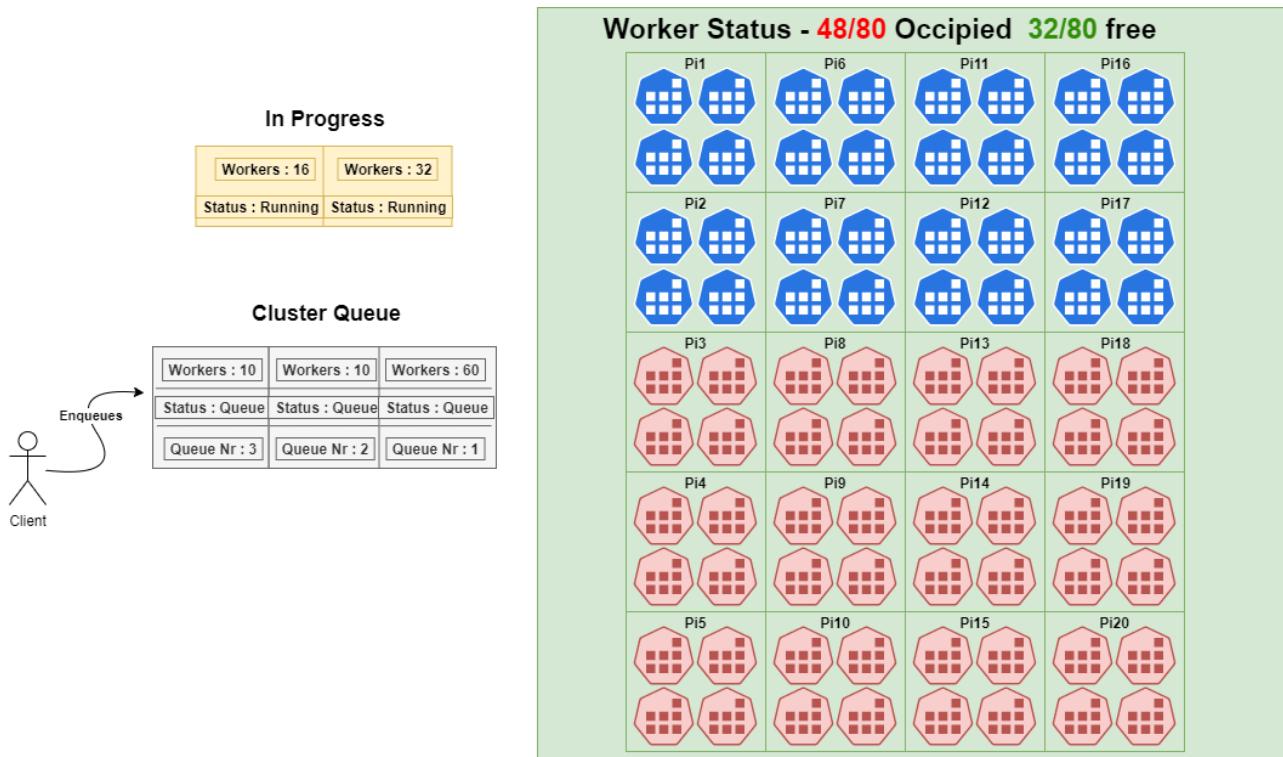


Figure 47: Køstruktur.

6.3.2 Thread safety

I og med at master servicen er udviklet i ASP .NET Core, vil de forskellige worker request blive håndteret samtidigt (concurrently). Dette betyder at flere tråde tilgår præcis den samme ClusterQueue. I og med at alle trådene læser og skriver den samme liste, kan der opstå uønsket fejlscenarier. Derfor er der valgt at bruge en ConcurrentQueue som ligger i .NET, som har en FIFO-struktur med indbygget thread-safety [33]. På den måde sikres uønskede fejlscenarier, muterende operationer på ClusterQueue'en automatisk lockes i en ConcurrentQueue.

6.3.3 Køstatus

Hver klient der requester workers vil have sin egen køstatus, som ændre sig undervejs i et jobbets forløb på clusteret. QueueStatus har en vigtig betydning for opdatering af firebase databasen, således frontend'en får live opdateringer omkring jobbets status og klienten kan følge forløbet. Live data opdatering af frontend'en er beskrevet i afsnit 6.3.5. QueueStatus bliver først tilgængelig når klients API-key'en er valid. På figur 48 ses de forskellige transitioner der kan være mellem de forskellige states.

- **InQueue:** Klienten er i kø. Hvis klienten ligger Forrest i køen, vil klienten vente på, at de requestede antal workers bliver ledige.
- **Failed:** Klientens job er fejlet. Dette sker når klientens kode kaster en exception.
- **Running:** Når en klient ligger Forrest i køen og der er nok ledige workers, vil klienten blive fjernet fra køen, og efterfølgende vil jobbet blive eksekveret.
- **Completed:** Hvis klientens arbejde gennemfører uden fejl vil status være "completed".

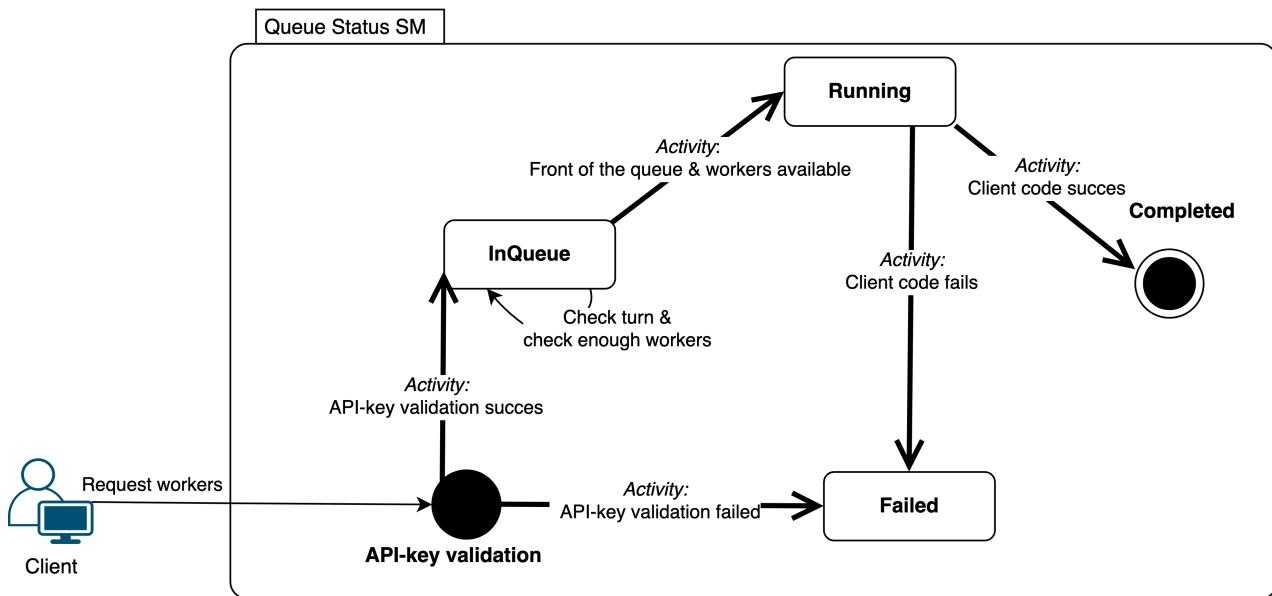


Figure 48: State machine Queue status

6.3.4 Hvornår er det ens tur?

Når en klient requester x-antal workers vil klienten blive sat i kø og få tildelt et unikt ID. Hvert element i køen vil derfor have sit eget unikke ID. Hver klient vil efterfølgende vente på at det bliver dens tur. Som det ses i koden på figur 49 vil hver klient poller i køen hvert 500 millisekund, og checke om det forreste elements ID, er lig dens egen ID. Hvis det forreste elements ID er lig klientens eget ID, vil funktionen terminere og jobbet igangsættes efterfølgende for den klient der terminerede. Hver klient der ligger i køen, vil hver især eksekvere `WaitTurnAsync` og vil derfor sidde fast i dette kald, indtil det bliver deres tur.

```

MasterController.cs

1  public async Task WaitTurnAsync(string documentId, CancellationToken token)
2  {
3      ...
4      while (true)
5      {
6          QueueElement currentWorkerInFront;
7          clusterQueue.concurrentQueue.TryPeek(out currentWorkerInFront);
8          if (currentWorkerInFront != null)
9          {
10              if (currentWorkerInFront.DocumentId == documentId)
11              {
12                  Console.WriteLine($"Its my turn {currentWorkerInFront.DocumentId}, starting..");
13                  return;
14              }
15          }
16          ...
17
18          await Task.Delay(500);
19
20      }
21  }
22  ...
23 }

```

Figure 49: WaitTurnAsync metode.

Hvornår er der ledige workers?

Når en klient ligger forrest i køen, er det hans tur til at requeste workers. LoadBalancer'en holder styr på hvilke pods der er ledige, og gennem den vil klienter forespørge x-antal workers. Da LoadBalancer'ens cache ikke altid er i synkronisering med de nyeste virtuelle IP'er fra kubernetes vil LoadBalancer'en altid opdatere dens interne cache/memory før den udleverer ledige workers. Hvis ikke der er nok workers, når klienten requester LoadBalancer'en, vil LoadBalancer'en hvert 1500 millisekund opdatere den interne cache/memory og checke om der er nok workers til klienten. Dette vil den blive ved med indtil der er nok workers til klienten. Når klienten endelig får det requestede antal af workers, igangsættes dens arbejde og ændre status til **Running**.

6.3.5 QueueUpdater

Når køen ændrer sig, er det vigtigt, at der kun er ét objekt som opererer ad gangen. Problemet er at der findes flere instanser af MasterController'en, og hver instans vil gerne ændre på køen, når der kommer et nyt job eller en status ændrer sig. For at imødekommme problemet er der udviklet en QueueUpdater. QueueUpdater er en baggrunds tråd som hele tiden lytter på køen for at tjekke statusser fra de forskellige klienter. Klienterne vil løbende opdatere deres status til enten **Running**, **Completed**, **Failed** og derved signalere til QueueUpdater. QueueUpdater holder styr på at opdatere køen og derefter give frontenden besked om at data har sendret sig. Når QueueUpdater fjerner elementer fra køen, vil den have en sekundær liste med alle de jobs som er i **Running** og som senere vil ende i **Completed** eller **Failed**. Grunden til den sekundære liste er, at selve køen er en fifo kø. Det vil sige at når et job bliver sat i **running**, så skal den fjernes, for at den næste i køen kan startes. Som det ses på figur 50 vil Cluster Queue'en kun have elementer som har status **InQueue** eller **Running**. JobInProgressList vil derimod kun have elementer som er **Running**, **Completed** eller **Failed**.

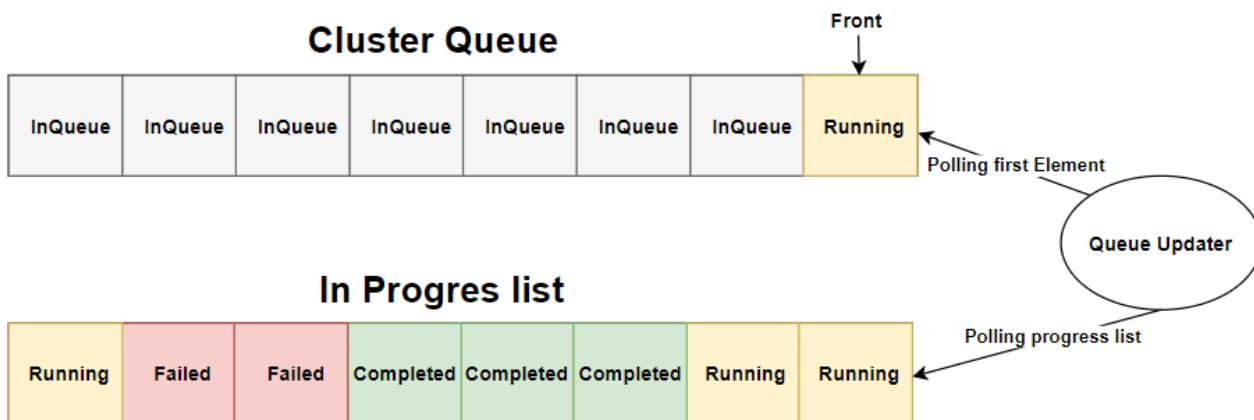


Figure 50: Queue Updater håndterer Cluster Queue & Jobs in Progress

QueueUpdateren tjekker hvert 100 millisekund om de forskellige klienter som ligger i køen har ændret status. Det er kun det forreste element i Cluster Queue'en som bliver checket. Men JobsInProgressList skal løbes helt igennem, da flere jobs kan være termineret. QueueUpdateren vil med Cluster Queue'en og JobsInProgressList'en lave følgende updateringer:

- **Cluster Queue polling**

- **Running:** Når det forreste element i Cluster Queue'en går i **Running**, betyder det at den forreste i køen, har fået det antal workers som den har requestet. QueueUpdateren vil derfor fjerne det forreste element fra Cluster Queue og ligge jobbet i JobsInProgress Listen. Når det sker vil den næste klient i køen opdage at det nu er dens tur, og vil begynde at vente på workers. QueueUpdateren vil opdatere kø numrene på firebase samt skrive til firebase at jobbet nu er i gang.
- **InQueue:** Hvis det forreste element i Cluster Queue har status **InQueue** betyder det at den forreste klient i køen stadig afventer ledige workers. Derfor gør QueueUpdateren ingen ting.

- **JobsInProgressList polling**

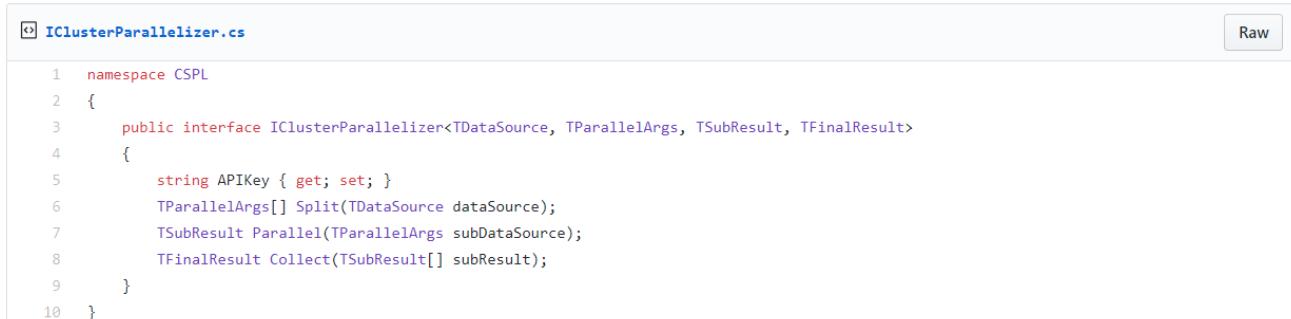
- **Running:** Hvis et element har status Running, er jobbet ikke termineret endnu, og QueueUpdateren vil derfor ikke gøre noget.
- **Failed:** Hvis et element har status Failed betyder det at et job har termineret med en fejl. Queue-updateren vil derfor opdatere firebase om at jobbet er fejlet.
- **Completed:** Hvis et element har status Completed betyder det at et job har termineret med success. QueueUpdateren Opdaterer firebase med at jobbet er gået godt.

6.4 NuGet-pakke

Når en klient skal implementere kode til CSPL systemet, gør han brug af ClusterSupportedParallelLibrary NuGet pakken. Nuget pakken er designet med henblik på at gøre det brugervenligt for klienten. Da hver raspberry Pi har fire workers/pods som beskrevet i afsnit 4.9, skal klienten ikke tænkte på selv at starte tråde, han skal blot implementere de retningslinjer interfacet har. Parallelisering handler om at starte tråde eller processer og køre dem samtidig. I nogle tilfælde skal der samles delresultater til et endeligt resultat.

Præcis denne tankegang om parallelisering er taget i brug for at designe ClusterSupportetParallelLibrary interfacet, også navngivet **IClusterParallelizer**. Som det ses i koden på figur 51 skal klienten implementere 3 forskellige metoder.

- **Split(dataSource)** definerer hvordan det givne arbejde skal splittes op i delmængder som skal proceseres parallelt på hver worker/pod. Antallet af delmængder som Split returnere, resultere i hvor mange workers klienten får tildelt.
- **Parallel(subDataSource)** definerer hvad hver worker/pod skal processere for at generere et delresultat til det endelige resultat.
- **Collect(subResults)** definerer hvorledes de forskellige delresultater fra Parallel-metoderne skal samles for at producere det endelige resultat.
- **API-key** skal indeholde klientens nøgle og den bliver valideret hver gang klienten requester CSPL systemet.



```

IClusterParallelizer.cs Raw

1  namespace CSPL
2  {
3      public interface IClusterParallelizer<TDataSource, TParallelArgs, TSubResult, TFinalResult>
4      {
5          string APIKey { get; set; }
6          TParallelArgs[] Split(TDataSource dataSource);
7          TSubResult Parallel(TParallelArgs subDataSource);
8          TFinalResult Collect(TSubResult[] subResult);
9      }
10 }

```

Figure 51: Interface for ClusterSupportedParallelLibraries

IClusterParallelizer er generisk hvilket gør at klienten kan bruge alle forskellige typer i hans implementering. Det eneste krav for typerne er at de kan serialiseres, hvilket de fleste typer i C# kan. Her er en liste over typerne som skal specificeres når interfacet skal implementeres.

- **TDataSource** Klienten definere her, hvilket dataset hans kode skal arbejde på. Denne type vil bliver input argumentet til Split.
- **TParallelArgs** Klienten definere her, hvilken type som hver parallel metode skal kaldes med. Splits retur-type er et array af typen TParallelArgs, hvert element i array'et bliver givet til en Parallel-metode.
- **TSubResult** Klienten definere returtypen for hver Parallel metode. Argumentet til Collect er et array af TSubResult typer, som indeholder returværdier fra Parallel-metoder, der hver især producerer et object af type TSubResult.

- **TFinalResult** Klienten definere hvilken type arbejdet skal ende med at returnere i hans/hendes program.

6.5 Compile time properties problem

Når interfacet implementeres har klienten mulighed for at definere sine egne metoder og properties. En vigtig og meget kritisk problemstilling, som kan ligge skjult for klienten er runtime properties vs compile properties. Det følgende vil demonstrere problemet.

Eksempel på problem med custom properties

Det antages at en klient vil implementere en algoritme der parallelt kan generere x-antal tilfældige tal og summere resultatet. Klienten har implementeret en custom property `RandomRange` som definere hvilke tilfældige tal der skal genereres. Som det ses på figur 52 sætter klienten `RandomRange` til at være 50 i hans program. Når DLL pakken bliver sendt til master servicen, vil den via. reflection finde hans implementerede klasse op fra DLL'en, og eksekvere kodden i interfacet. Problemets er her, at klienten tror at hans kode kører med `RandomRange` lig 50, men den har faktisk stadig har default værdien 10, da det er en compile time property, da den har default værdi 10 nede i DLL pakken.

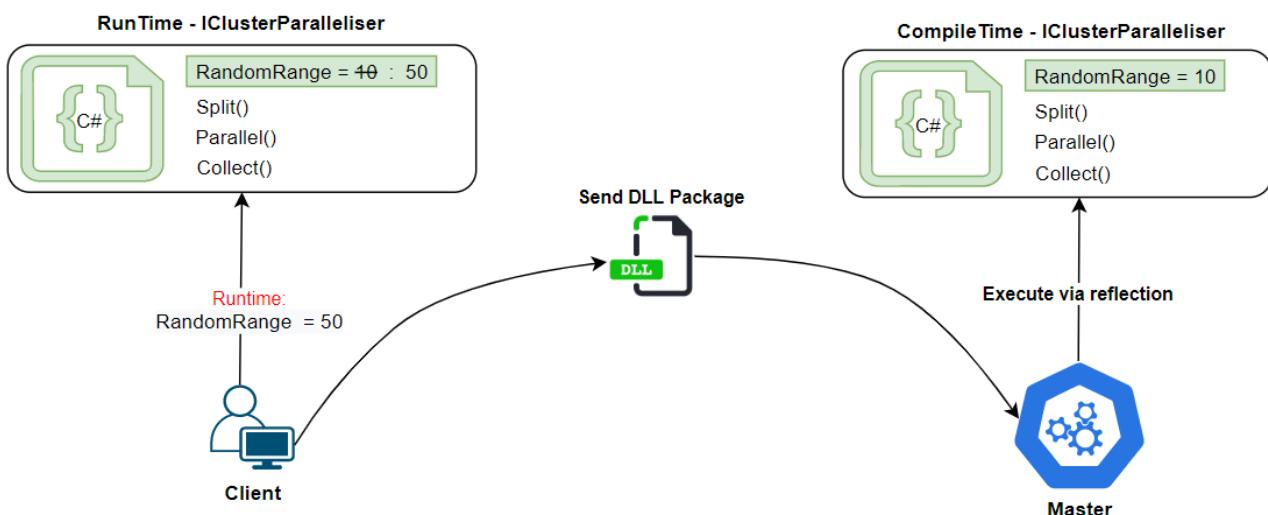


Figure 52: Example: property does get assigned.

Løsning til custom properties

For at undgå problemet, er der designet funktionalitet til at sætte properties på runtime, ovre i master servicen i stedet for at gøre det i klientens eget program. Som det ses i koden på figur 53, gemmer NuGet pakken de requestede runtime properties og deres value i et dictionary, som sendes med i DLL pakken. Master servicen kan via. reflection på runtime sætte de requestede properties på klientens IClusterParallelizer.

```

 SetRunTimeProperty.cs
Raw
1  public void SetRunTimeProperty(string propertyName, object PropertyValue)
2  {
3      properties.Add(propertyName, System.Text.Json.JsonSerializer.SerializePropertyValue));
4  }

```

Figure 53: SetRunTimeProperty method

Som det ses på figur 54 kaldes `SetRunTimeProperty` i stedet for selv at sætte propertien manuelt. Master servicen kan derfor på runtime via reflection sætte de requestede properties på klienten implementerede `IClusterParalleliser`. Som det ses på figur 54 ender masterservicen med at sætte værdien til 50, hvilket var det som klienten requestede med metoden `SetRunTimeProperty`.

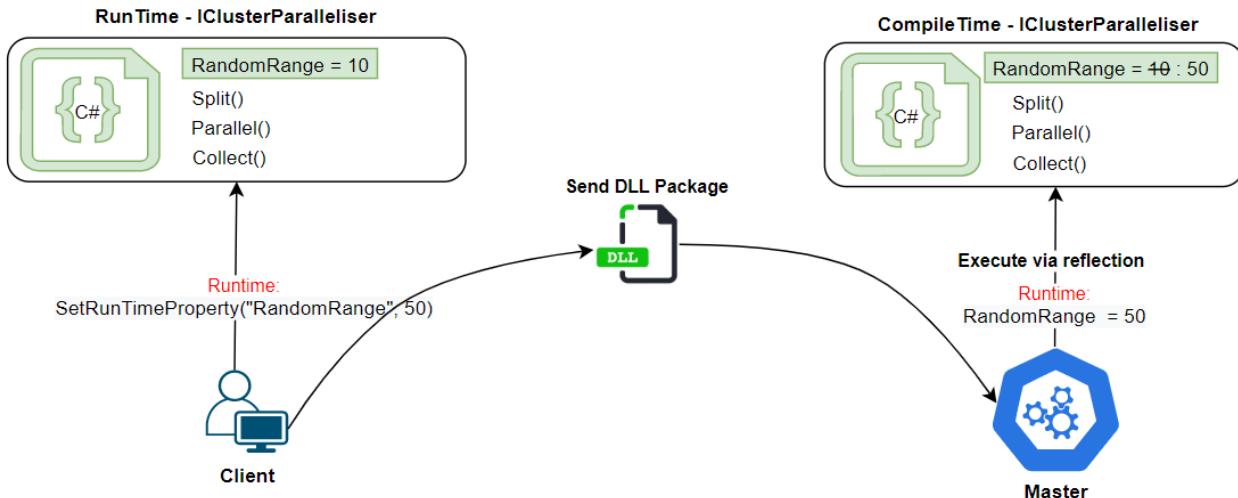


Figure 54: Example : property gets assigned correctly

7 Implementering

I følgende afsnit beskrives udarbejdelsen af de vigtigste komponenter i produktet, samt de mest essentielle algoritmer som er udviklet til at løse specifikke opgaver for enten produktet eller udviklingsprocessen.

7.1 Sekvensdiagrammer

Følgende sekvensdiagrammer tager udgangspunkt i systemsekvensdiagrammet i afsnit 5.2 og omfatter brugen af CSPL NuGet-pakken, Master API'et og Worker API'et, samt viser kommunikationen derimellem. Sekvensdiagrammerne viser også hvordan hvert komponent håndtere assemblies/DLL'er, serialisering/deserialisering, datapakker, reflection, type resolving, delegering af jobs mm.

7.1.1 NuGet-pakken

Sekvensdiagrammet på figur 55 viser den sekvens som eksekveret i CSPL NuGet-pakken når en udvikler/client implementere interfacet, starter eksekveringen af dette på clusteret og får svar derfra. Sekvensen for Master API'et ses på figur 56.

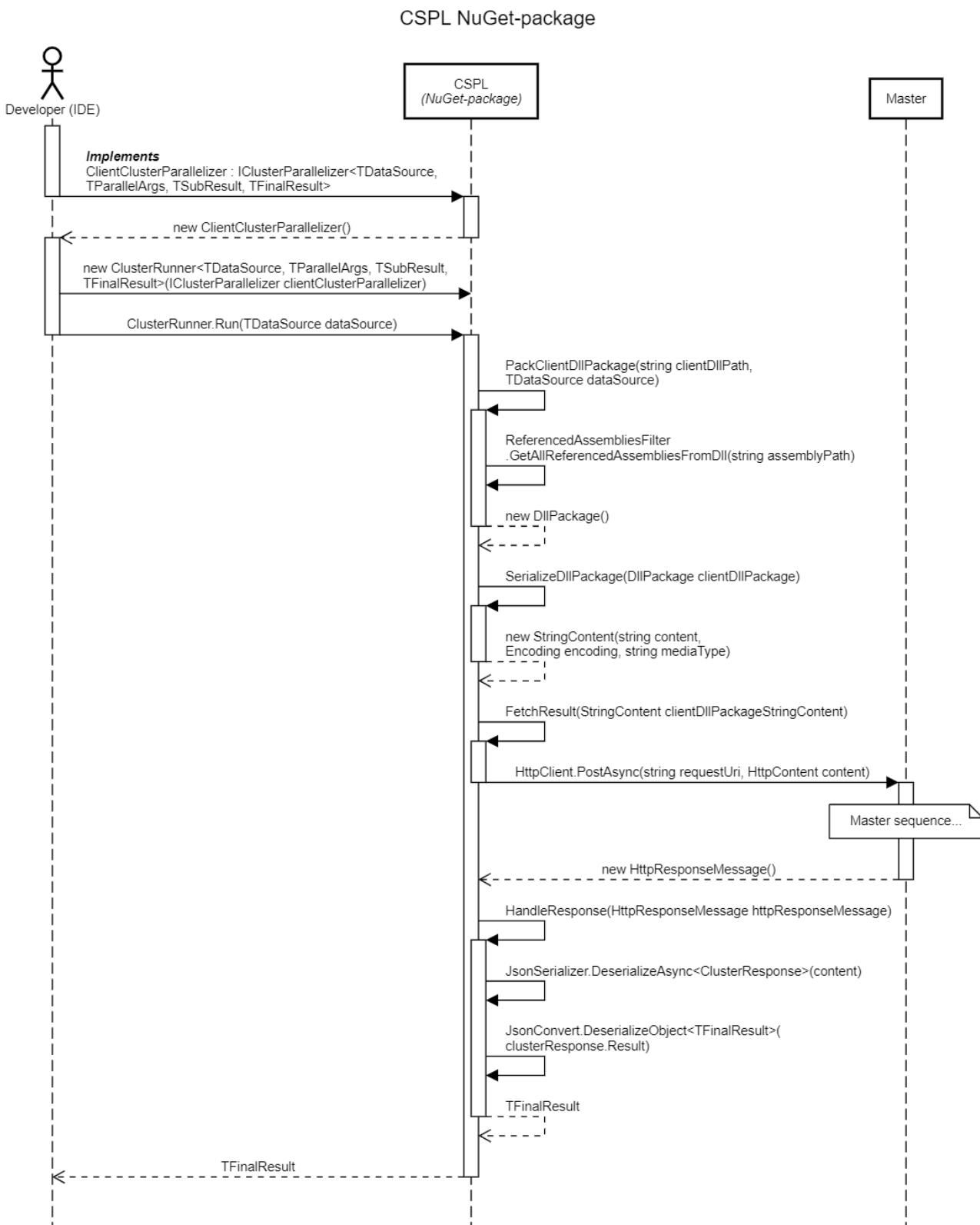


Figure 55: Sekvensdiagram for CSPL NuGet-Pakken.

7.1.2 Master API

Figur 56 viser dele af den sekvens som eksekveres på Master API'et når der modtages et job fra en udvikler/klient (se figur 55). Sekvensen for Worker API'et ses på figur 57.

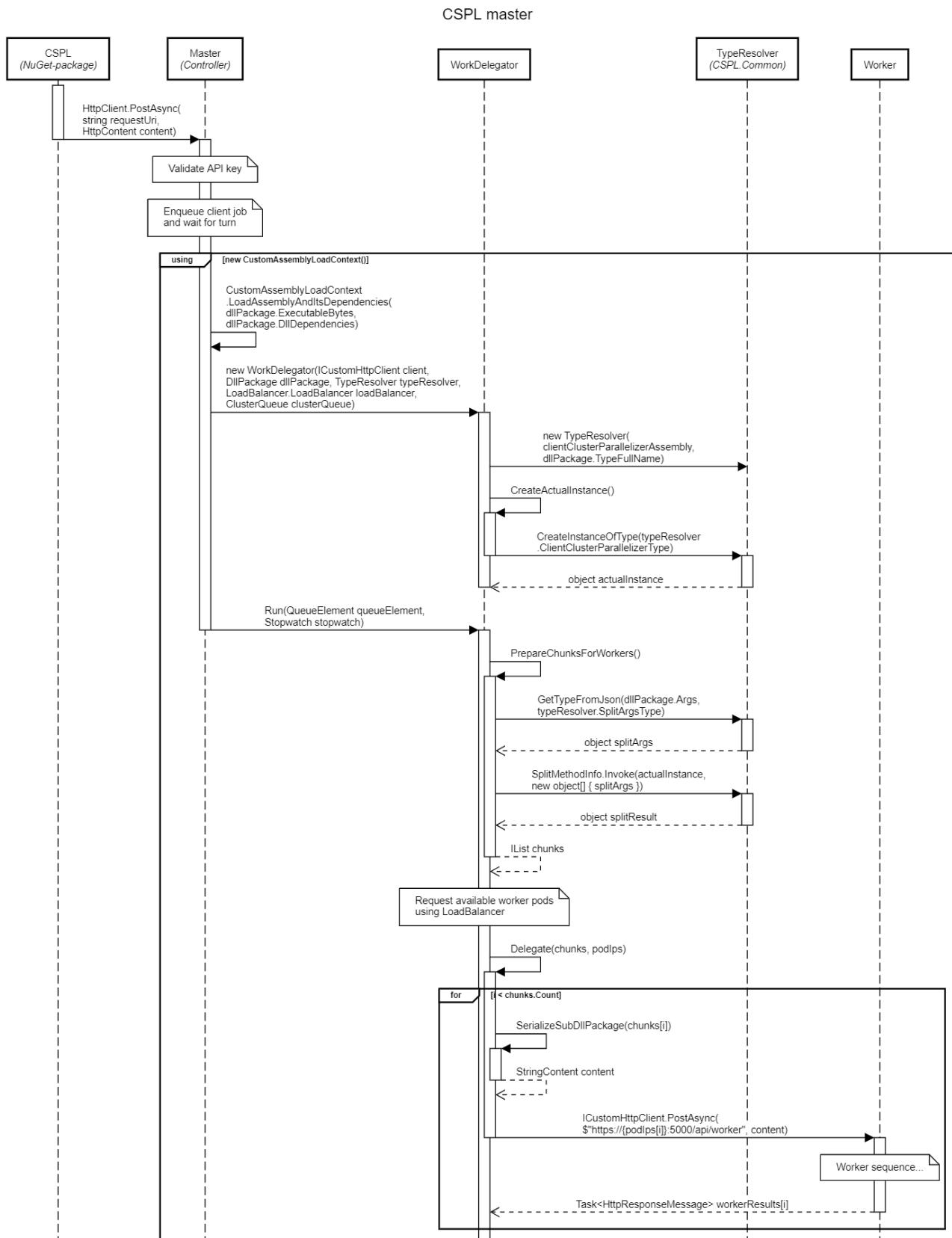


Figure 56: Del 1 - Sekvensdiagram for CSPL Master.

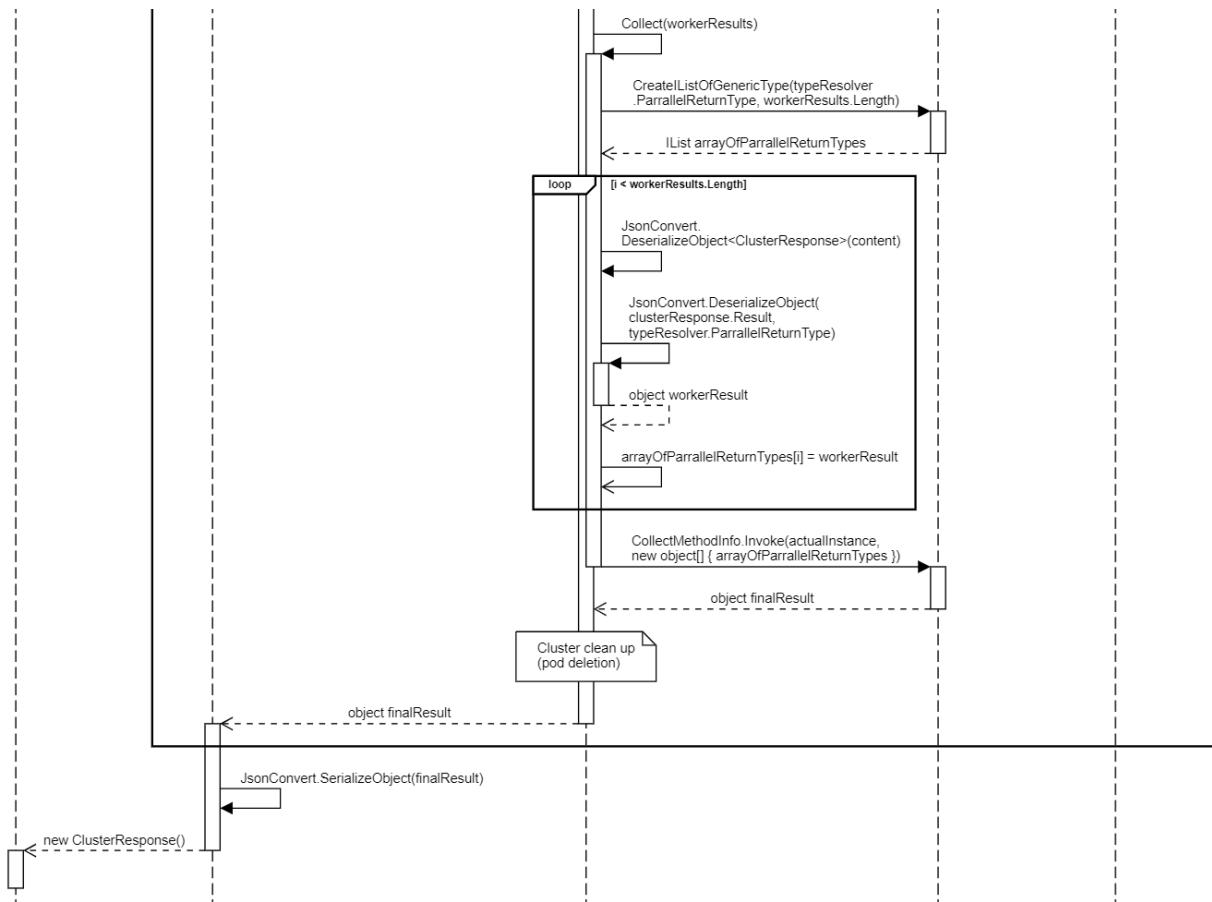


Figure 56: Del 2 - Sekvensdiagram for Master API.

7.1.3 Worker API

Sekvensdiagrammet på figur 57 viser den sekvens som eksekveret i Worker API'et, når API'et modtaget en del af et job fra Master API'et (se figur 56).

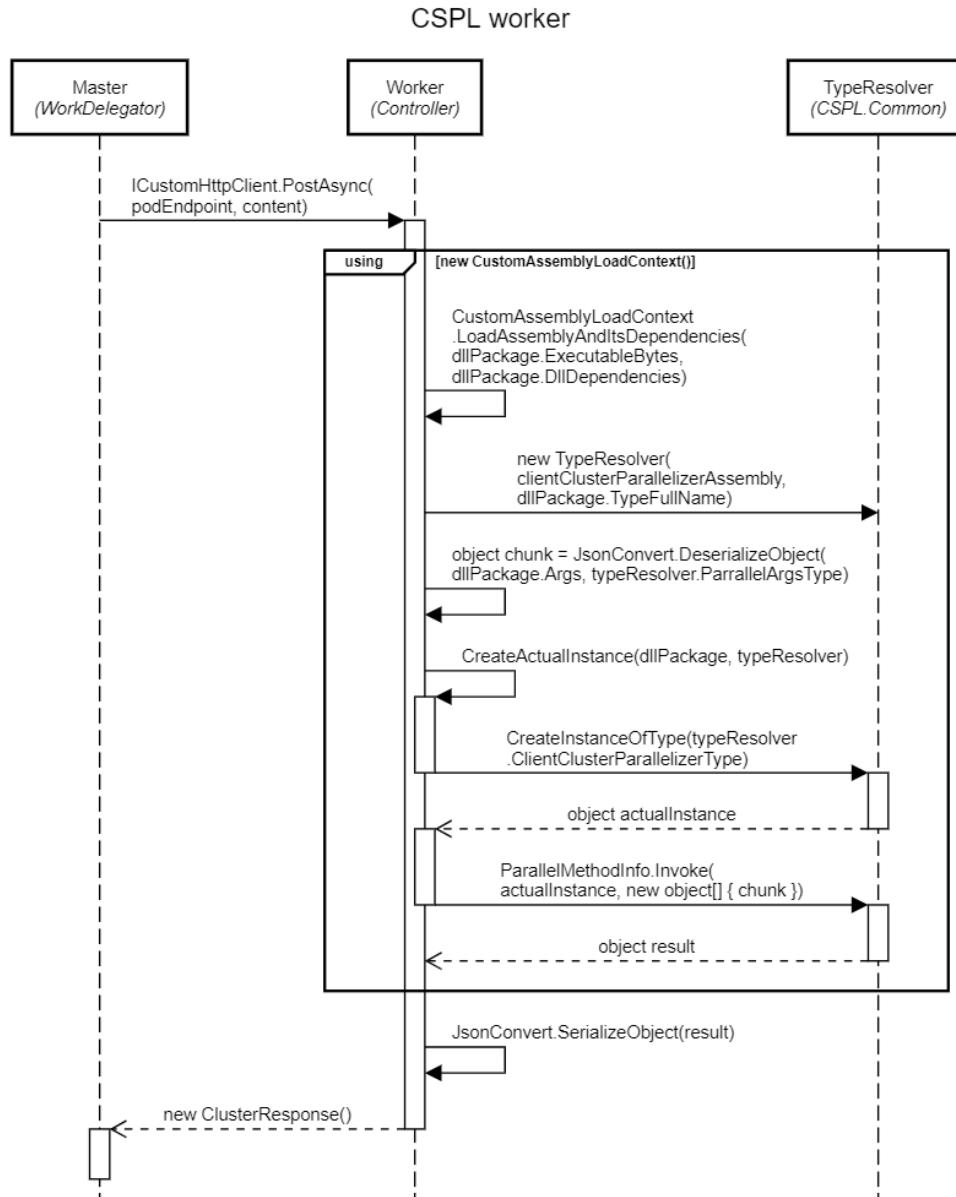


Figure 57: Sekvensdiagram for Worker API.

7.2 Delt NuGet-pakke

Grundet at der er delt funktionalitet imellem nogle komponenter i CSPL systemet, kan det til tider være svært at undgå kodeduplikering. For at undgå dette, er der oprettet en NuGet pakke, hostet på en privat NuGet-server som udviklerne fra CSPL kan tilgå. De indkluderede services i CSPL master & worker, er begge afhængig af funktionalitet, såsom type-kalkulation til kalkulation af klientens selvdefinerede typer via reflection, samt assembly loading af klient DLL'er og CSPL specificerede exceptions.

Grundet de to services deler denne funktionalitet, er NuGet-pakken `CSPL.Common` blevet lavet. Koden er derved nemmere at vedligeholde, da det ligger ét sted. NuGet pakken understøtter også semantisk versionering, således at der kan holdes styr på hvornår der sker bugfixes, minor feature implementations eller breaking changes. `CSPL.Common` har også sin helt egen pipeline på build-serveren, hvor den bygger, pakker og publicere NuGet-pakken, efter at have sikret grønne unit- og integrationstests.

7.3 DLL dependencies og filter

For at muliggøre eksekvering af en klients kode på en ekstern maskine, skal klientens DLL'er og dependencies samles og sendes til den eksterne maskine. Der er derfor implementeret en metode som rekursivt finder en alle dependencies for en given DLL-fil, samt de subdependencies der måtte være. En DLL kan altså have multiple DLL-dependencies, hvoraf disse DLL'er også kan have yderligere dependencies og dette kan visualiseres vha. et "dependency-træ" som ses på figur 58.

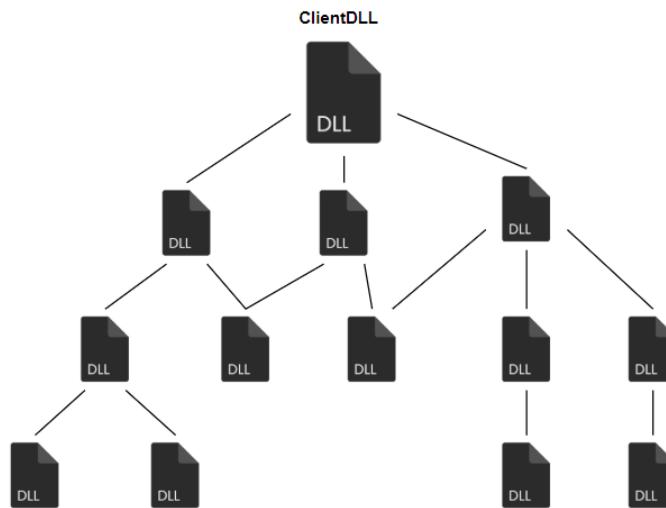


Figure 58: Dependency-træ.

Metoden gør det muligt at samle og senere hen sende clientens DLL-program samt alle de DLL'er programmet er dependent på. Dette gør at det på clusteret er muligt at load disse assemblies og dermed eksekvere clientens program via reflection. Da clientens program kan have dependencies som stammer fra .NET Core og/eller et 'third-party'-library (f.eks. en NuGet-pakke), er det derfor ikke alle DLL der skal samles og sendes med til clusteret, da workers/Pi's allerede har installeret .NET Core og de dertilhørende DLL'er. Derfor bliver alle dependencies hold op imod en liste af alle de DLL'er der ligger i .NET Core og dermed bliver de sorteret fra. Listen af .NET Core-specifikke DLL'er er lavet og udledt vha. PowerShell, se figur 59.

```

powershell_deps.ps1
dir -Path 'C:\Program Files (x86)\dotnet\' -Filter *.dll -Recurse | %{$_.Name} | Out-File C:\dependencies.txt
  
```

Figure 59: PowerShell command til at lave en .txt-fil med alle navne på de DLL-filer som ligger i dotnet-mappen.

7.4 DLL-pakke

7.4.1 Sending af DLL-pakke

De vigtigste attributter er henholdsvis den DLL-fil som skal eksekveres, dens afhængigheder (som består af andre DLL-filer), samt de argumenter eller det datasæt som DLL-filens metoder skal have som argument. Derudover skal der bruges en type på den klasse, hvori de metoder som skal eksekveres ligger. Modellen for hvordan en sådan pakke ser ud ses på figur 60.

```

DLLPackage.cs

1  internal class DllPackage
2  {
3      public string ExecutableName { get; set; }
4      public byte[] ExecutableBytes { get; set; }
5      public Dictionary<string, byte[]> DllDependencies { get; set; }
6      public string TypeFullName { get; set; }
7      public string Args { get; set; }
8      public string APIKey { get; set; }
9      public string Label { get; set; } = "";
10     public Dictionary<string, string> PropertyValues { get; set; }
11 }

```

Figure 60: DLL-pakke til sending af data.

Attributten `executableFile` udgør det kode som brugeren har skrevet, som han gerne vil have eksekveret på clusteret, `executableName` er navnet på denne fil. DLL-filen som skal eksekveres, bliver sammen med alle dens afhængigheder (typisk andre DLL-filer) streamet til byte arrays. DLL-filens afhængigheder bliver lagt i `dllDependencies` som er et dictionary, hvilket er en key/value-pair liste, hvor navnet på DLL'en bliver lagt som key og dens assembly i form af et byte array som value. `TypeFullName` er en attribut som fortæller helt præcist hvilket namespace og klasse navn brugeres implementerede klasse ligger under. Hvis brugeres klasse lå under namespace'et Client og han klasse hedder Client, så vil `TypeFullName` indeholde `Client.Client`. Dette er nødvendigt for at worker-norderne kan dykke ned i DLL'en og finde den rigtige klasse, hvori de metoder som skal kaldes ligger. Det sidste argument `args` udgør det datasæt brugeren gerne vil have eksekveret sin metode på. Dette er egentlig en arbitrær type T men den bliver sendt som en string. Alle disse dele kan pakkes i et objekt og sendes til clusteret.

7.4.2 Eksekvering af DLL-pakke

Når en DLL fil fra en klient bliver modtaget af en node, er der forskellige forudsætninger der skal tages hen-syn til, før noden kan gøre brug af den sendte DLL pakke. DLL'erne bliver sendt via HTTPS og ligger som rå bytes i det sendte JSON formaterede objekt. For at noden kan gøre brug af DLL'erne skal de først og fremmest loades ind i en context hvori processens kan eksekveres. Disse assemblies/DLL'er loades altså ind i en ny assembly context som er et slags scope hvori man kan load, resolve og unload DLL'er [30]. Dette er nødvendigt for at eksekvere de metoder der er afhængige af andre DLL'er, dette er illustreret på figur 61. Det er også nødvendigt at oprette en ny assembly context således, at når eksekveringen er færdig skal context'en unloads således at garbage collectoren kan samle disse assemblies. Hvis ikke der bliver oprettet en ny assembly context og DLL'erne istedet bliver loadet ind i worker-programmets/nodens context, vil de ikke blive unloadet efter endt eksekvering af kilentens DLL og der vil derfor være en risiko for, at programmet kaster en exception, næste gang den modtager en klient-DLL som har samme DLL-dependency og forsøger at loade den ind i en context hvor den assembly allerede eksisterer.

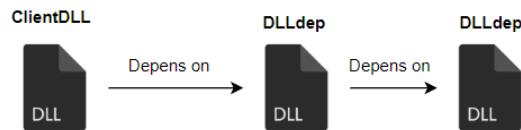


Figure 61: Client DLL dependencies

Når alle DLL'er er blevet loadet, kan noden via reflection dykke ned i DLL'en og finde de metoder der skal kaldes. Vha. et striks interface som klienten har lovet at implementere, ved noden lige præcis hvilke metoder den skal lede efter. Da der er flere forskellige klasser som kan implementere dette strikse interface, er det vigtigt at der ligger noget typeinformation i den sendte DLL-pakke. Som beskrevet i tidligere afsnit 7.4.1 sendes attributten `TypeFullName` som holder information om hvilket namespace klientens klasse ligger i, samt navnet på den implementerede klasse. For at illustrere dette, antages det at brugeren lover at implementere følgende funktionsdefinition som ses i listing 1

Listing 1: Generic Parallel klient Function

```
1 public T ParallelFunc(T someGenericTypeArg);
```

Der gøres brug af `TypeFullName` som klienten har sendt, for at kunne finde frem til klientens klasse. Når klient klassen er fundet, kan der søges efter metoden "ParallelFunc". En vigtig observation er her at "ParallelFunc" tager en generisk type, hvilket medfører at den noden ikke umiddelbart har nogen mulighed for at kunne konvertere de sendte argumenter til de korrekte typer. Dette skyldes at "ParallelFunc" tager en generisk type `T`, som der ikke kendes til. Dette problem løses med reflection som kan finde typer på runtime. På figur 62 ses de step der er nødvendige for at finde frem til den rigtige "ParallelFunc" argument type.

1. Brug reflection for at finde en konkrete type af `T`
2. Konverter det sendte argument fra JSON til den konkrete type.
3. Invoker ParallelFunc function med instansen af den konkrete type.

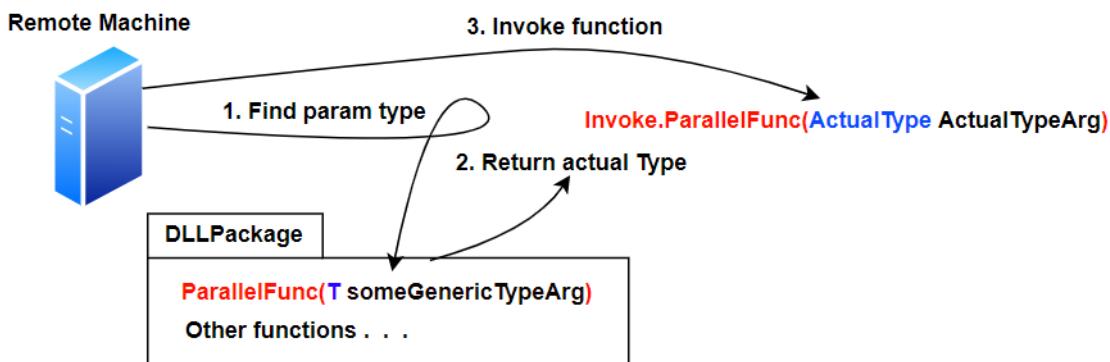


Figure 62: Find den konkrete type for ParallelFunc argumentet.

7.5 Opsætning og problemer med Kubernetes cluster

Implementeringen af et Kubernetes cluster med en J5005 mini-ITX master-node og Raspberry Pi's some worker-nodes har ikke været uden problemer.

7.5.1 Kubeadm

Kubeadm , værktøjet som bruges til at 'bootstrap' et hurtigt Kubernetes cluster, gav problemer med dens join kommando [24]. Master-noden som efter at have kørt `kubeadm init` generere et token som worker-nodes bruger til at forbinde til clusteret [21]. Et problem her er, at master-noden skal have dens firewall åbnet til portene 6443, 2379 – 2380 og 10250 – 10252 eller være deaktivert. Vis ikke dette er gjort vil worker-noden forsøge at forbinde til clusteret og blive stoppet. Specielt for Raspberry Pi's/Raspbian er, at de ikke har en firewall som fungere på samme måde som på Ubuntu OS'et. Ønsker man at route et request fra en master-node til en worker-node, skal man sætte iptables til at acceptere alle forwarded requests.

7.5.2 Pod network

For at nodes og pods kan kommunikere og nodes gå fra state `<NotReady>` til `<Ready>`, skal et pod network installeres. Der findes mange forskellige pod networks, hvor nogle af de mere kendte såsom Weave Net, Calico og Flannel anbefales i Kubernetes dokumentationen [19].

Det viste sig, at Weave Net som skal installeres på master-noden, kræver SEL (Security Enhanced Linux) installeret. SEL som er et Linux kernel sikkerhedsmodul som tilbyder en mekanisme for understøttelse af adgangskontrol og sikkerhedspolitikker [65]. Denne tilføjelse til master-noden resultere dog i problemer med kubeadm join kommandoen. Løsningen til dette problemer er at installere Flannel som pod netværk istedet. Flannel er mere 'light weight' og kræver dertil nogle argumenter ved `kubeadm init`. Her er det vigtigt at give argumentet `-pod-network-cidr=10.244.0.0/16` med, da de nye deployed pods vil logge warnings for forkert IP-konfiguration.

7.5.3 Dynamiske IP-adresser

Et essentiel forudsætning er at clusteret har statiske IP- adresser. Ved `kubeadm init`, sættes clusteret op til at route trafik til dens nuværende IP-adresse. Genstartes clusteret og master- eller worker-nodes får tildelt nye IP-adresser, vil masters API server, worker-nodes kubelet og docker daemon ikke kunne kommunikere sammen. Løsningen til dette problem er ikke at have DHCP (Dynamic Host Configuration Protocol) IP-adresser. Løsningen er en switch og en router som kan administrere hvilken IP-adresse en given MAC-adresse får tildelt, se figur 63.

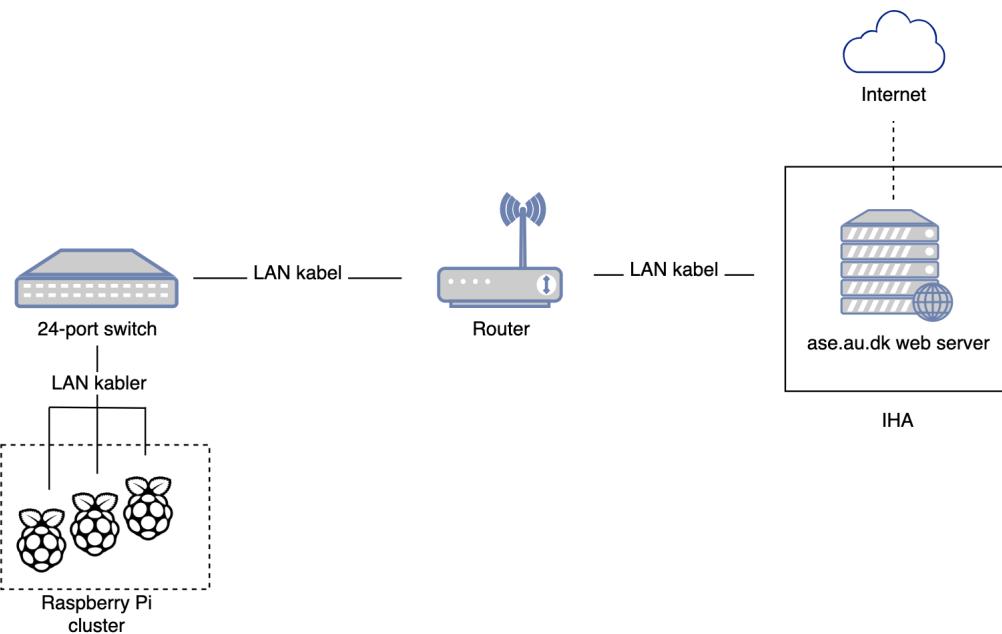


Figure 63: Opsætningen af det kontrollerbare netværk

7.6 Deployment og konfigurering

Efter at Kubernetes clusteret er sat op, kan der deployeres master og worker-pods ud på clusteret. Det bliver gjort ved at apply en .yaml-fil til clusteret vha. `kubectl` på master-noden (se listing 2) [25].

Listing 2: `kubectl apply`.

```
1 $ kubectl apply -f masterDeploymentAndService.yaml
2 $ kubectl apply -f workerDeploymentAndService.yaml
```

I .yaml-filen bliver den nødvendige information for deployment specificeret, se figur 64 og 65. Bla. navn, namespace, replicas og label på deploymenten. Labels bliver bla. benyttet til at identificere pods, replication controllers og services [23], f.eks. så den service som pods skal tilgåes igennem kan target de rigtige pods vha. selector, se service i figur 64 og 65. Derudover bliver den container som skal køre i pod'en specificeret, det container image bliver automatisk hentet fra Docker Hub af pod'en. Container porten fortæller hvilken port pod'en kan tilgåes på internt i clusteret. Der bliver også lavet en service som pods skal tilgåes igennem, se service i figur 64 og 65, her specificeres bla. hvilken port pod'en skal tilgåes på indefra (port) og udefra (nodePort), i dette tilfælde er den ikke specificeret og der bliver tildelt en tilfældig f.eks. 31711), samt den eksterne IP-adresse, se figur 69. TargetPort er den port som servicen i pod'en udstiller [48].

 **workerDeploymentAndService.yaml**

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: worker
5    namespace: cspl
6    labels:
7      app: worker
8  spec:
9    selector:
10      matchLabels:
11        app: worker
12    replicas: 16
13    template:
14      metadata:
15        deletionGracePeriodSeconds: 0
16        labels:
17          app: worker
18          status: free
19    spec:
20      containers:
21        - name: worker
22          image: docker.io/cspl/worker:1.1.33.61
23          volumeMounts:
24            - name: csplworkerapiconfig
25              mountPath: /etc/ssl/certs/CsplWorkerApiCertificate.pfx
26              subPath: CsplWorkerApiCertificate.pfx
27              readOnly: false
28          resources:
29            limits:
30              memory: "120Mi"
31            requests:
32              memory: "120Mi"
33          ports:
34            - containerPort: 5000
35          volumes:
36            - name: csplworkerapiconfig
37            configMap:
38              name: csplworkerapiconfig
39          restartPolicy: Always
40          nodeSelector:
41            disktype: worker
42          terminationGracePeriodSeconds: 0
43  ---
44  apiVersion: v1
45  kind: Service
46  metadata:
47    name: worker-service
48    namespace: cspl
49    labels:
50      app: worker
51  spec:
52    type: ClusterIP
53    ports:
54      - port: 8021
55        targetPort: 5000
56    selector:
57      app: worker
58      status: free

```

Figure 64: Worker .yaml for Kubernetes

```

☒ masterDeploymentAndService.yaml

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: master
5    namespace: cspl
6    labels:
7      app: master
8  spec:
9    selector:
10      matchLabels:
11        app: master
12    replicas: 1
13    template:
14      metadata:
15        deletionGracePeriodSeconds: 0
16        finalizers:
17          - foregroundDeletion
18        labels:
19          app: master
20    spec:
21      containers:
22        - name: master
23          image: docker.io/cspl/master:2.1.333.203
24          volumeMounts:
25            - name: csplmasterapiconfig
26              mountPath: /etc/ssl/certs/CsplMasterApiCertificate.pfx
27              subPath: CsplMasterApiCertificate.pfx
28              readOnly: false
29          ports:
30            - containerPort: 5000
31          volumes:
32            - name: csplmasterapiconfig
33            configMap:
34              name: csplmasterapiconfig
35          tolerations:
36            - key: "node-role.kubernetes.io/master"
37              effect: "NoSchedule"
38              operator: "Exists"
39          restartPolicy: Always
40          nodeSelector:
41            disktype: master
42  ---
43  apiVersion: v1
44  kind: Service
45  metadata:
46    name: master-service
47    namespace: cspl
48    labels:
49      app: master
50  spec:
51    type: NodePort
52    ports:
53      - port: 8020
54        targetPort: 5000
55    selector:
56      app: master
57    externalIPs:
58      - 192.168.1.100

```

Figure 65: Master .yaml for Kubernetes

Pods kan deployeres ud i flere replicas alt efter hvor mange instanser af pod'en der skal være i clusteret. I dette tilfælde er der 16 pod replicas af worker-servicen, fordelt på fire Raspberry Pi-nodes i clusteret, se figur 66, 67 og 68.

```
master@master-node:~$ kubectl get nodes
NAME      STATUS  ROLES   AGE     VERSION
master-node  Ready   master  3d21h  v1.17.3
pi01      Ready   <none>  3d21h  v1.17.3
pi02      Ready   <none>  3d21h  v1.17.3
pi03      Ready   <none>  3d21h  v1.17.3
pi04      Ready   <none>  3d21h  v1.17.3
```

Figure 66: kubectl get nodes.

```
master@master-node:~$ kubectl get deployment -n cspl
NAME      READY  UP-TO-DATE  AVAILABLE  AGE
worker    16/16  16          16          16m
```

Figure 67: kubectl get deployment.

```
master@master-node:~$ kubectl get pods -n cspl -o wide
NAME        READY  STATUS   RESTARTS  AGE     IP           NODE
worker-dd57699c-5kwd8  1/1   Running  0          19m   10.244.1.43  pi01
worker-dd57699c-bf825  1/1   Running  0          19m   10.244.1.41  pi01
worker-dd57699c-bwczj  1/1   Running  0          19m   10.244.4.43  pi04
worker-dd57699c-crvzt  1/1   Running  0          19m   10.244.3.34  pi03
worker-dd57699c-dzz22  1/1   Running  0          19m   10.244.4.42  pi04
worker-dd57699c-fjbvl  1/1   Running  0          19m   10.244.3.31  pi03
worker-dd57699c-fqpd9  1/1   Running  0          19m   10.244.4.44  pi04
worker-dd57699c-mdk77  1/1   Running  0          19m   10.244.2.41  pi02
worker-dd57699c-n98vj  1/1   Running  0          19m   10.244.2.43  pi02
worker-dd57699c-pb54w  1/1   Running  0          19m   10.244.2.42  pi02
worker-dd57699c-qlxzf  1/1   Running  0          19m   10.244.2.40  pi02
worker-dd57699c-s9chm  1/1   Running  0          19m   10.244.3.33  pi03
worker-dd57699c-skhvc  1/1   Running  0          19m   10.244.3.32  pi03
worker-dd57699c-trcs7  1/1   Running  0          19m   10.244.4.45  pi04
worker-dd57699c-wzwww  1/1   Running  0          19m   10.244.1.44  pi01
worker-dd57699c-xg792  1/1   Running  0          19m   10.244.1.42  pi01
```

Figure 68: kubectl get pods.

```
master@master-node:~$ kubectl get service -n cspl
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)        AGE
worker    NodePort  10.98.116.221  192.168.1.100  8020:31711/TCP  18m
```

Figure 69: kubectl get service.

Antallet og fordelingen af worker-pods er bestemt ud fra at der skal være én worker-pod pr. CPU kerne. På hver Raspberry Pi-node er der sat en begrænsning på hvor mange pods noden kan få tildelt. Denne restriktion er sat vha. flaget 'maxPods' i config.yaml for kubelet på hver Raspberry Pi, config-filen ligger under /var/lib/kubelet/config.yaml. Hvis en Raspberry Pi mister forbindelsen til clusteret bliver disse 'døde' pods altså ikke scheduleret ud på de andre nodes, så der er mere kontrol i clusteret i forhold til ægte parallelitet. På figur 70 og 71 ses resultatet af at pi01 har mistet forbindelsen til clusteret. Der er nu fire færre replicas i tilgængelige i deploymenten og Kubernetes forsøger at terminere de 'døde' pods og venter på at schedulerne fire nye replicas af worker-pods ud på en node der har plads. Genskabes forbindelsen til pi01 bliver de nye replicas scheduleret ud på den og deploymenten vil igen have 16 replicas tilgængelige.

```
master@master-node:~$ kubectl get deployment -n cspl
NAME      READY  UP-TO-DATE  AVAILABLE  AGE
worker    12/16  16          12          22m
```

Figure 70: kubectl get deployment, ingen forbindelse til pi01.

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
worker-dd57699c-5kwd8	1/1	Terminating	0	21m	10.244.1.43	pi01
worker-dd57699c-7l29v	0/1	Pending	0	82s	<none>	<none>
worker-dd57699c-7tl79	0/1	Pending	0	82s	<none>	<none>
worker-dd57699c-bf825	1/1	Terminating	0	21m	10.244.1.41	pi01
worker-dd57699c-bwczj	1/1	Running	0	21m	10.244.4.43	pi04
worker-dd57699c-crvzt	1/1	Running	0	21m	10.244.3.34	pi03
worker-dd57699c-dzz22	1/1	Running	0	21m	10.244.4.42	pi04
worker-dd57699c-fjbjv1	1/1	Running	0	21m	10.244.3.31	pi03
worker-dd57699c-fqpd9	1/1	Running	0	21m	10.244.4.44	pi04
worker-dd57699c-mdk77	1/1	Running	0	21m	10.244.2.41	pi02
worker-dd57699c-n98vj	1/1	Running	0	21m	10.244.2.43	pi02
worker-dd57699c-pb54w	1/1	Running	0	21m	10.244.2.42	pi02
worker-dd57699c-qlxzf	1/1	Running	0	21m	10.244.2.40	pi02
worker-dd57699c-qv4lj	0/1	Pending	0	82s	<none>	<none>
worker-dd57699c-s9chm	1/1	Running	0	21m	10.244.3.33	pi03
worker-dd57699c-skhvc	1/1	Running	0	21m	10.244.3.32	pi03
worker-dd57699c-trcs7	1/1	Running	0	21m	10.244.4.45	pi04
worker-dd57699c-wzwww	1/1	Terminating	0	21m	10.244.1.44	pi01
worker-dd57699c-x85k2	0/1	Pending	0	82s	<none>	<none>
worker-dd57699c-xg792	1/1	Terminating	0	21m	10.244.1.42	pi01

Figure 71: kubectl get pods, ingen forbindelse til pi01.

7.7 Forurening af pods i clusteret

7.7.1 CustomAssemblyLoadContext

Med udgangspunkt i problemstillingen beskrevet i afsnit 6.2 omkring forurening af pods, er der implementeret en CustomAssemblyLoadContext. Klassen nedarver fra AssemblyLoadContext [31], som wrapper de assemblies som loades ind i programmet ind i denne AssemblyLoadContext. Dette giver mulighed for at unloads denne AssemblyLoadContext efter endt arbejde, se figur 72. Klassen nedarver også fra IDisposable, så den i f.eks. MasterController'en er nemt at benytte vha. using, se figur 73.

CustomAssemblyLoadContext klassen er gjort collectible så AssemblyLoadContext'en kan unloads/garbage collector'en kan indsamle assembly'en [32]. Unload af assembly'en sker først når garbage collector'en indsamler alle objekter fra assembly'en [36]. Derudover er AssemblyLoadContext'en sat som en WeakReference [43], således at unload af assembly'en stadig kan ske, selvom der er en reference til AssemblyLoadContext'en, samt at man senere hen kan tjekke for om assembly'en er blevet garbage collected. Dette bliver gjort ved at trigger garbage collectoren, og vente på at den WeakReference ikke længere er i live. Først der er AssemblyLoadContext'en unloaded og dermed også alle de assemblies der er loaded ind i contexten, se Dispose() på linie 19 i figur 72.

```
CustomAssemblyLoadContext.cs

1  internal class CustomAssemblyLoadContext : AssemblyLoadContext, IDisposable
2  {
3      WeakReference weakReference;
4      public CustomAssemblyLoadContext() : base(isCollectible: true)
5      {
6          weakReference = new WeakReference(this, true);
7      }
8
9      public Assembly LoadAssemblyAndItsDependencies(byte[] assembly, Dictionary<string, byte[]> dependencies)
10     {
11         ...
12     }
13
14     private Assembly LoadAssembly(byte[] assembly)
15     {
16         ...
17     }
18
19     public void Dispose()
20     {
21         Unload();
22         for (int i = 0; weakReference.IsAlive && i < 10; i++)
23         {
24             GC.Collect();
25             GC.WaitForPendingFinalizers();
26             GC.Collect();
27         }
28     }
29 }
```

Figure 72: CustomAssemblyLoadContext.

```
MasterController.cs

1  using (CustomAssemblyLoadContext assemblyLoader = new CustomAssemblyLoadContext())
2  {
3      Assembly clientClusterParallelizerAssembly =
4          assemblyLoader.LoadAssemblyAndItsDependencies(dllPackage.ExecutableBytes, dllPackage.DllDependencies);
5
6      ...
7
8  } // Unload assemblies
```

Figure 73: Brug af CustomAssemblyLoadContext i MasterController.

Dette betyder at de assemblies og dermed det kode som kommer fra klienten og bliver eksekveret i clusteret, ikke forurener systemet men derimod bliver indsamlet og ryddet op.

7.7.2 Oprydning af worker-pods

I forbindelse med problemstillingen beskrevet i afsnit 6.2 omkring forurening af pods, er der i master API'et inkorporeret funktionalitet, der ved hjælp af data fra Kubernetes API'et og cache i memory fra LoadBalancer'en (se afsnit 6.1), nedlægger worker-pods efter hver gang, pod'en har værer i brug. Kubernetes sørger selv for at opretholde det antal af replicas der skal være og så snart en pod bliver nedlagt spawner kubernetes automatisk en ny.

For at teste dette i starten blev worker-pods slettet manuelt vha. en kommando i terminalen på master-noden, se listing 3. Der blev tilføjet tagget `deletionGracePeriodSeconds: 0` for worker deployment i Kubernetes, for at sikre hurtig nedlægning af pods og dermed også hurtigere forbindelse til nye pods.

Listing 3: Slet alle worker pods command.

```
1 $ kubectl -n=cspl get pods | grep worker | awk '{print $1}' | xargs kubectl delete ←
  pod --grace-period=0 -n cspl
```

Ved hjælp af data fra Kubernetes API'et og cache i memory fra LoadBalancer'en kunne dette implementeres i master API'et, således at det var muligt at slette lige præcis de eksakte pods som har været i brug. Dette er vigtigt, for ikke at slette pods som er ved at eksekvere et andet stykke arbejde for en anden klient, samt for ikke unødig at slette pods som ikke har været i brug. På figur 74 ses den metode som sletter worker-pods i clusteret ud fra pod-ID'er vha. Kubernetes API'et.

KubernetesFetcher.cs

```
1  public void DeleteBulkPods(string[] podNames, string kubeNamespace)
2  {
3      Task<V1Status>[] tasks = new Task<V1Status>[podNames.Length];
4      for (int i = 0; i < podNames.Length; i++)
5      {
6          tasks[i] = DeletePod(podNames[i], kubeNamespace);
7      }
8      Task.WaitAll(tasks);
9  }
10 }
11
12 public Task<V1DeploymentList> ListNamespacedDeployment(string kubeNamespace)
13 {
14     return kubeClient.ListNamespacedDeploymentAsync(kubeNamespace);
15 }
```

Figure 74: Metode i KubernetesFetcher til at slette delete pods vha. Kubernetes API'et.

Metoden på figur 74 bliver efter endt arbejde, invokeret af LoadBalancer'en som mapper de pod-IP'er, som et klient kald oprindeligt har fået tildelt, til de dertilhørende pod-ID'er, og dermed de worker-pods som er blevet benyttet under eksekveringen af klientens arbejde, se figur 75.

LoadBalancer.cs

```
1  public void DeleteBulkPods(string[] podIps)
2  {
3      string[] podNames = new string[podIps.Length];
4
5      for (int i = 0; i < podIps.Length; i++)
6      {
7          if (currentPodsAvailable.TryGetValue(podIps[i], out Pod pod))
8          {
9              podNames[i] = pod.Name;
10         }
11     }
12
13     kubernetesFetcher.DeleteBulkPods(podNames, "cspl");
14 }
```

Figure 75: Metode i LoadBalancer til at omsætte de pod-IP'er som skal slette til pod-ID'er og derefter invokerer sletning via KubernetesFetcher.

7.8 SSL

SSL, den nu nyere version TLS [66], er det blevet en industristandard, som alle professionelle hjemmesider og dermed datalinjer bruger til at sikre sikkerhed mellem part A og B. Et 'must-have' indenfor de fleste IT brancher, og et krav for CSPL systemet.

Hvordan fungere det?

Http kald med TLS skal først lave et initial 'handshake'. I dette handskake udveksler klient og serveren hver deres public nøgle. Serveren kryptere en symmetrisk session-key med klientens public nøgle og sender den tilbage

til klienten. Klienten åbner denne krypteret pakke via. hans private key. Nu kan data mellem server og klient begynde, da begge parter har den samme symmetriske session-key.

Denne udveksling af nøgler tillader sikker data kommunikation.

7.8.1 TLS tradeoff

TLS tilbyder sikkerhed, men på hvilken bekostning?

HTTP-kald uden TLS starter sendingen af datapakker efter forbindelsen mellem klient og host er lavet. Her er der ingen forsinkelse på sendingen, og data er sendt som de er, rå. Http kald med TLS skal først lave et initial handshake. Dette initial handshake og kryptering af hver datapakke giver sikkerhed, på bekostningen af tid. Tiden at sende pakker derefter afhænger af to ting, CPU-kræft og internet hastighed. Den ene afgørende faktor til sendingen, er hvor hurtigt klienten kan kryptere det data som skal sendes. Har klienten en god CPU, vil krypterings processen ikke tage lang tid, og dermed ikke være en flaskehals. Omvendt, har klienten en mindre god CPU, vil krypterings tiden være længere og dermed forlænge sendings tiden.

For hvad angår sendingen af data, kan det til tider være hurtigere at sende med TLS frem for uden. Når datapakkerne krypteres, sker der en slags komprimering, en form for 'zipping' af filerne, hvilket gør den overordnet pakke mindre [2, 44]. Ved sending af små data pakker, kan TLS blive også blive et overhead da klient og server skal asymmetrisk kryptere og uddele den symmetriske session-key og derefter sende den lille pakke. Argumentet for ikke at bruge TLS, grundet hastighedsforsinkelserne, er dog så marginale, at sikkerhedsfordelene opvejer enhver potentiel påvirkning af hastigheden.

7.8.2 TLS hjemmeside, master og workers

Efter at have undersøgt fordele og ulemper ved TLS, skal de forskellige komponenter; hjemmesiden, master og workers, have vurderet hvilke der skal sikres med TLS.

Hjemmesiden skal have et TLS certifikat, grundet en bedre Google page ranking [64], for at opfylde kravene om at være en progressive webapp og give kunder en forskring om sikre kommunikationslinjer.

Master service skal også have TLS, grundet den sensitive data mellem klient og master. Vil en virksomhed ønske at bruge CSPL, vil de være forsikret om, at deres data ikke bliver sendt råt over HTTP.

For de forskellige worker services, er de utilgængelig for alle andre end master servicen da de kører på et virtuelt netværk i clusteret som kun master servicen kan tilgå. Af denne grund er argumentet at TLS ikke var nødvendigt.

Mere til, vil alle workers kun få en brøkdel af den originale pakke fra klienten. Disse mindre datapakker kan risikere at skabe et overhead, da kryptering og udveksling af nøgler for hvert kald er nødvendigt.

Problemet med ikke at anvende TLS er dog den fremtidig udvikling af produktet. Hvis CSPL SW løsningen sælges videre, skal der sikres TLS hele vejen igennem, fra klient til worker. Yderligere vil der være mulighed for at tilslutte flere worker-nodes fra hele verdenen rundt, ikke kun en fysisk begrænsning på lokation til det virtuelle netværk.

7.8.3 Opsætning

7.8.3.1 Hjemmeside

Hjemmesiden er hosted med Firebase og et TLS certifikat er givet på forhånd. Dette TLS certifikat er cineret af CA'en Lets' Encrypt. Hjemmesiden som oprigtigt har følgende dns navn: <https://cspl-d8b81.web.app/> har fået anvendt et nyt dns navn: cspl.dk, hvilket peger på det givet Firebase dns navn. Set på figur 76, er der vist certifikatet for cspl.dk.

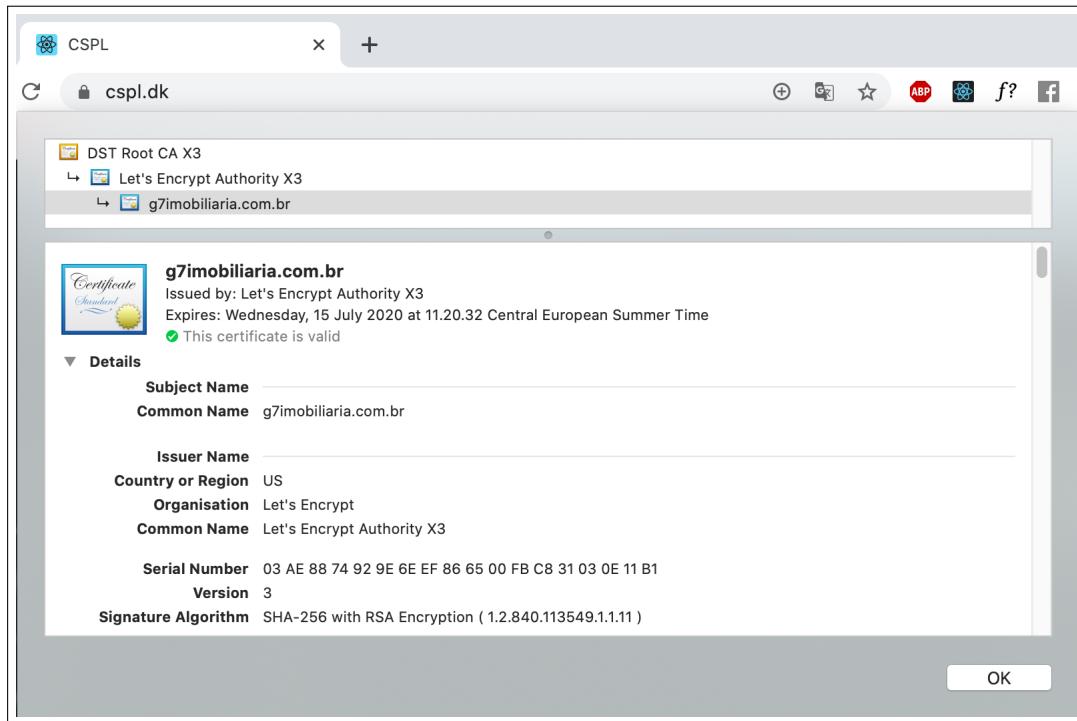


Figure 76: TLS certifikat fra CSPL.dk

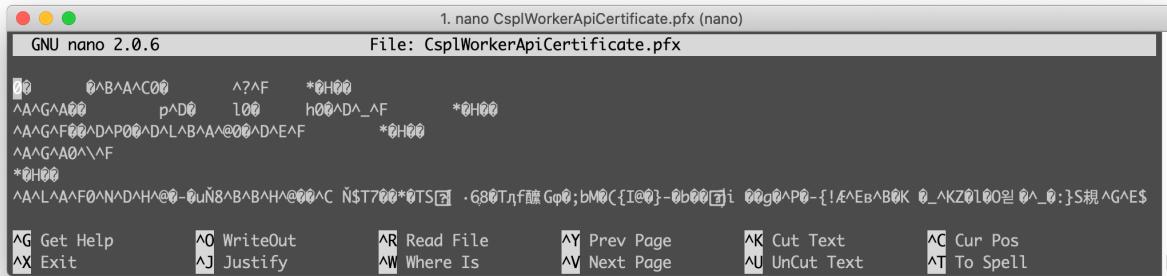
7.8.3.2 Master & Workers

For at få TLS certifikater til worker services og master servicen, skal der først genereres et certifikat. Et certifikat består af en *.crt fil og en private key. Disse filer er henholdsvis nøglerne til kryptering og dekryptering, men også informationer omkring hvem certifikatet tilhører og hvem har signeret det. Disse filer kan genereres fra en værktøj som Openssl. Openssl er et kryptografibibliotek, som bla. kan lave *.pfx filer. Disse filer formår at samle privat nøglen og *.crt filen i én *.pfx fil, som Kubernetes gerne vil acceptere.

Set på figur 77, er der vist hvordan Openssl med input domain.key (private key) og domain.crt (public key og informationer omkring certifikatet) generere en *.pfx fil af typen pkcs12. Denne fil er selv krypteret og et ulæselig, se figur 78.

```
1. pmh@mac: ~/SSL (zsh)
+ SSL openssl pkcs12 -export -out CsplWorkerApiCertificate.pfx -inkey domain.key -in domain.crt
```

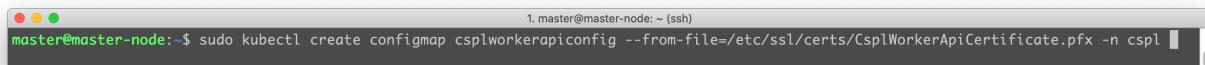
Figure 77: Generering af *.pfx fil



The screenshot shows a terminal window titled "File: CsplWorkerApiCertificate.pfx" with the text "1. nano CsplWorkerApiCertificate.pfx (nano)". The file contains binary certificate data represented as hex values. Below the editor are various keyboard shortcuts for navigating and editing the file.

Figure 78: Indholdet af den genereret *.pfx fil

Når *.pfx filen er genereret, skal der laves et configmap i Kubernetes [49]. Et configmap, giver mulighed for at afkoble konfigurations artefakter, hvilket hjælper med at holde containeren porterbare [20], således fungere den som en slags service, som tillader pods at hente filer fra configmap ressourcen. Hertil kan man via en deployment.yaml fil referer til ens configmap og dermed få adgang til *.pfx filen, se figurene 79, 80.



```
1. master@master-node: ~ (ssh)
master@master-node:~$ sudo kubectl create configmap csplworkerapiconfig --from-file=/etc/ssl/certs/CsplWorkerApiCertificate.pfx -n cspl
```

Figure 79: Kubernetes, kreation af et configmap, som peger på openssl *.pfx filen



```
1. master@master-node: ~ (ssh)
volumeMounts:
- name: csplworkerapiconfig
  mountPath: /etc/ssl/certs/CsplWorkerApiCertificate.pfx
  subPath: CsplWorkerApiCertificate.pfx
  readOnly: false
resources:
  limits:
    memory: "120Mi"
  requests:
    memory: "120Mi"
ports:
- containerPort: 5000
volumes:
- name: csplworkerapiconfig
  configMap:
    name: csplworkerapiconfig
```

Figure 80: worker.yaml filen, som tilgår configmap ressourcen

For master servicen opsættes ASP.NET core serveren til at lytte på port 5000, med det signeret certifikat fra Let's Encrypt [16], *set på figur 81*. Url adressen for CSPL NuGet pakken sættes til at pege på CSPL DNS'en frem for IP'en *set på figur 82*. Port 443 på det lokale netværk er ligeledes også blevet åbnet, så TLS trafik kan ramme master servicen, *se afsnit portforwarding 7.12*.



```
1 //private readonly string baseMasterUrl = "http://localhost:5000";
2 //private readonly string baseMasterUrl = "https://62.107.0.222:6443/api/v1/namespaces/cspl/services/master-service/proxy";
3 private readonly string baseMasterUrl = "https://cluster.cspl.dk"; //SSL cert expires in 90 days
```

Figure 81: Master kestral server lytter til https forbindelser

```
EnableHttps.cs

1 options.Listen(IPAddress.IPv6Any, 5000, listenOptions =>
2 {
3     //Enable https
4     listenOptions.UseHttps(GetMountedCertificate());
5 });


```

Figure 82: Forrige og nuværende MasterUrl strings

Når master og worker applikationerne er blevet ændret til at bruge https og certifikatet er blevet signeret af et trukket CA, vil man via programmer såsom Wireshark, kunne lytte til netværks trafikken og se hvordan data er krypteret.

På figur 83, er der vist hvordan man uden en https forbindelse kan finde frem til hvad klienten har modtaget. Set på linje nr. 395 er det vist at klienten kalder en GET-request på endpoint /api/master. Nogle acknowledgments senere, modtager klienten et response i form af json (linje nr. 404), hvor det ellers fortrolige data kan læses.

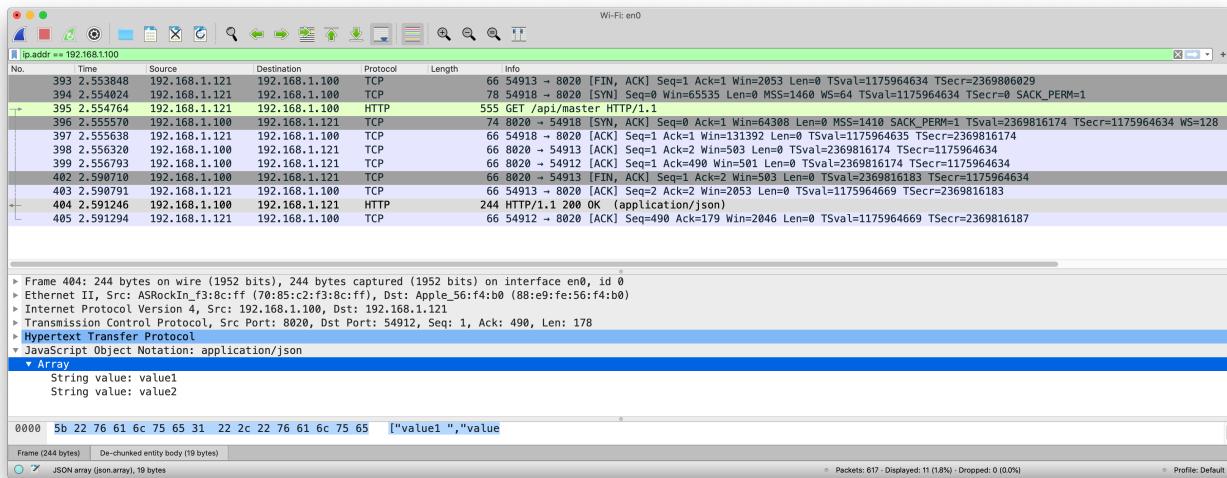


Figure 83: WireShark lytter til netværkstrafik uden TLS

Ser man derimod på figur 84, på linje nr. 67, modtages der data, men i krypteret form. Her er det vist, hvordan udefrakommende ikke længere har muligheden for at lytte med på netværkstrafikken og læse brugerfølsom data.

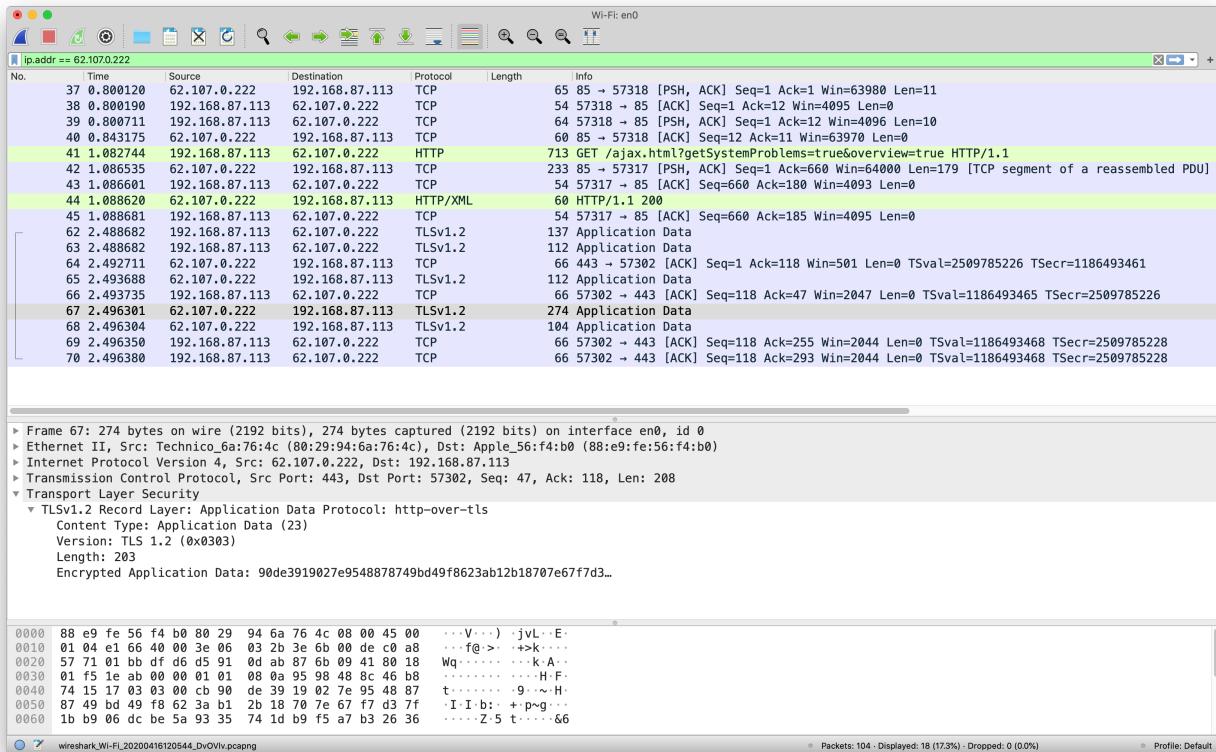


Figure 84: WireShark lytter til netværkstrafik med TLS

7.9 API-Key history view

En af siderne på hjemmesiden er API-Key history, hvor man kan se oversigten over de kald klienten har lavet til clusteret. API-Key komponenten er bestående af en react material tabel [27], som igennem en firebase komponent, subscriber på "calls" collectionen i firebase databasen. Målet med denne tabel, er at kunne vise data i realtid, således at klienter kan se og spore deres jobs foruden at opdatere siden forny. Når der manipuleres med en komponents state i react, bliver komponenten automatisk re-rendered. Dette koncept gøres der brug af, hver gang der kommer ny data i den tabel som bliver subscriptet på.

På figur 86 er der vist et sekvens diagram for hvordan de diverse kald fra klient, til API-Key komponenten udarbejdes.

Klienten starter med at besøge cspl.dk/api-key og får vist tabellen med API-Key historik. Komponenten med tabellen i, subscriber på `firebase.callsDoc`. CallsDoc er variabel som indeholder en reference til et database dokument som indeholder information om brugerens API-Key data. Det ses på figur 85 at callsDoc bliver sat, når en bruger logger ind.

④ **FirebaseCallsDoc.js**

```
1  class Firebase {
2    constructor() {
3      firebase.initializeApp(firebaseConfig);
4      this.db = firebase.firestore();
5      ...
6      this.auth.onAuthStateChanged(user => {
7        ...
8        this.userDoc = this.db.collection('Users').doc(user.email);
9        this.fetchUserAPIKey(user.email).then((res) => {
10          this.apiKey = res.APIKey;
11          this.callsLeft = res.CallsLeft;
12          this.callsDoc = this.db.collection('APIkeys').doc(this.apiKey).collection("Calls");
13        });
14        ...
15      });
16    }
}
```

Figure 85: kode udsnit af et event som sker på login og log ud.

CallsDoc er en reference til en collection som hedder 'Calls'. Calls ligger i et dokument som har navnet på den API-Key som brugeren har fået genereret. Det dokument ligger inde i en collection af API-Keys for alle brugere.

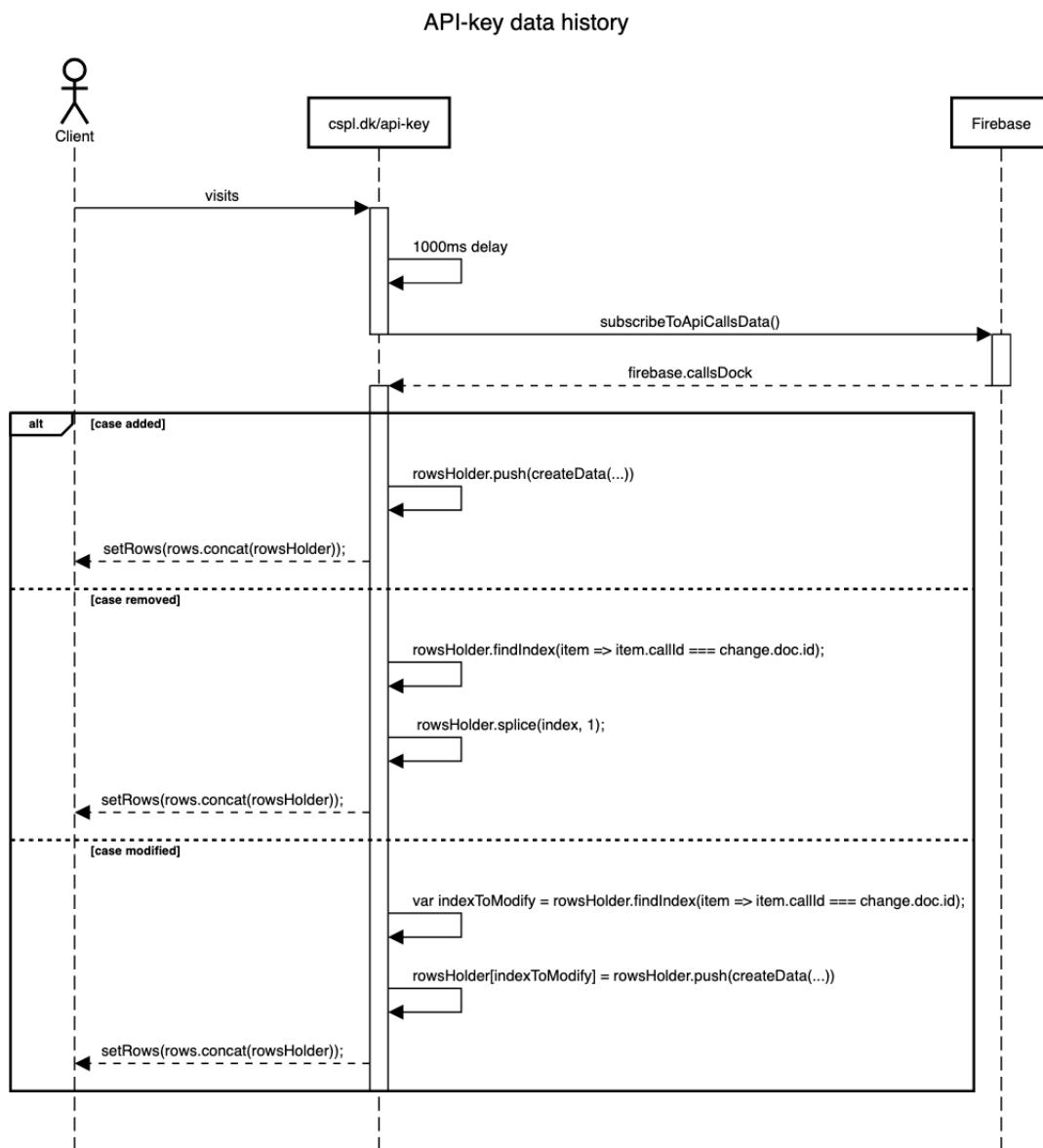


Figure 86: SD. for API-Key data history

Efter at have subscriptet på en collection, får man et snapshot, hver gang et stykke data blive tilføjet, ændret eller fjernet. Der er med andre ord mulighed for at modtage 'realtime updates' [14]. Tabellen skal nu opdatere sig selv, hver gang callsDoc ændre sig.

For at muliggøre dette, skal tabellen have dens state ændret for hver gang callsDoc får ændret data, således at tabellen kan re-render. For at være sikker på at firebase komponenten faktisk har lavet callsDoc, sættes et timer på 1000ms således at programmet ikke crasher. På figur 87 ses et eksempel på det den collection som bliver subscriptet på.

\$2a\$08\$SrFSocTIAYQqjjBhuJbhOqncyA... ::	Calls	GObGpmIf3TdHUhyHbf4A ::
+ Start collection	+ Add document	+ Start collection
Calls >	GObGpmIf3TdHUhyHbf4A >	+ Add field
	ReYPtCPptRWMwobnswFe afQD3U9RRwSf2Jtg0gZf xd7lGYKgDgII0B5HaQHn zWJpJvRC1jNJF3HB9yrs	ClassName: "MonteCarlo" ExceptionMessage: <i>null</i> ExecutionTime: 8.723 Label: "10-Pods-4M-Iterations" NumberOfNodes: 10 QueueNumber: 0 Status: "Completed" TimestampEnd: April 21, 2020 at 1:30:51 PM UTC+2 TimestampStart: April 21, 2020 at 1:30:35 PM UTC+2

Figure 87: Firestore dokument database collection for jobs

Måden hvorpå staten for tabellen ændres er via et hjælpe array som masker ændringer til det oprindelige array, dette hjælpe array hedder **rowsHolder**. Tanken er, at der skal maskes ændringer til det oprindelige array ved at konkatener forskellene sammen, og dernæst kalde **setState**. Dette vil tvinge det oprindelige array til at re-render. Når et snapshot fremkommer, kan det data der følger med, være i tre states; cases added, case removed eller case modified.

For alle cases undtaget removed, gør de brug af **createData()** funktionen, som set på figur 88.

Ved case added, kalder rowsHolder **createData** funktionen og sætter de forskellige parametre til at være lig dataen modtaget fra callsDoc snapshottet. Når rowsHolder har lavet et data object, bliver hele rowsHolder og rows (som er den originale tabel) konkatineret, for at maske forskellene sammen.

```
createData.js
1 function createData(callId, numberOfNodes, startTime, endTime, executionTime, className, status, icon, queueNumber, label) {
2     return { callId, numberOfNodes, startTime, endTime, executionTime, className, status, icon, queueNumber, label };
3 }
```

Figure 88: createData function

Case removed, findes der det object som matcher callsDoc snapshot Id'et. Herefter slettes objektet på det specifikke index, og ændringerne maskes over i rows.

Case modified, er en blanding af de to forrige cases. Der findes først det specifikke index som skal modificeres, hvorefter et nyt objekt laves og erstatter det gamle. Dette maskes så over i rows.

7.10 Firebase abstraktion

For at gøre brug af Firebase og tillade andre komponenter at kalde ind til firestore, skal firebase installeres og opsættes. Det installeres igennem CLI'en med node package manager (npm). Derefter kan man importere alle de nødvendige firebase biblioteker. [12].

Firebase.js, som er den primære klasse, importerer de nødvendige komponenter, såsom **firebase/app** som giver mulighed for at lave et firebase objekt. Der er også mulighed for at importere ekstra features som autentifikation og adgang til firestore er også givet.

Da CSPL hjemmesiden skal bruge disse ekstra features til login system, og hente data fra firestore, importeres **firebase/auth** og **firebase/firestore**.

For at fortælle firebase hvilken database, data skal hentes fra, skal en **firestoreConfig** opsættes. Værdierne til dette **firestoreConfig** er frarådet at sættes i samme klasse. Ville en person ønske selv at lave kald til cspl fire-

store api'et, kunne han via informationerne i firebaseconfigen gøre det. Af denne grund gemmes alle firebasemconfig informationerne i en .env fil.

Til komponenter som skal genbruges mange gange, som firebase, kan man lave en HOC (higher-order component). En HOC er en avanceret teknik i React for at kunne genbruge komponent logik. HOCs er ikke en del af React API'et, men er et design pattern, som er opstået fra Reacts måde at have komposition over komponenter[53]. Mere konkret sagt, er en HOC en funktion som tager et komponent og returnerer et nyt komponent, med nye properties.

I stedet for, at alle komponenter, som ønsker at gøre brug af firebase skal tage en firebase prop, kan komponenter lade sig wrappe i en HOC-komponent og bruge firebase funktionalitet, *se figur 89*.

```
loginComponent.jsx
1 export default withRouter(withFirebase(withSnackbar(CreateAccount)));
```

Figure 89: Create account komponent bliver wrappet i firebase HOC

Set på figur 90, er der vist context.js klassen, som exporterer `withFirebase` funktionen som HOC. `WithFirebase` tager en komponent som argument, og wrapper det så i en firebase context, hvor firebase proppen gives med, og returnerer det nye component.

```
context.js
1 export const withFirebase = Component => props => (
2   <FirebaseContext.Consumer>
3     {firebase => <Component {...props} firebase={firebase} />}
4   </FirebaseContext.Consumer>
5 );
```

Figure 90: Context.js klassen som exportere `withFirebase` som HOC

7.11 Autorisation

For brugeren, er API-nøglen deres måde at kunne interagere med clusteret på. API-nøglen bliver lavet på frontenden, men valideres på backenden. API-nøglen laves for første gang, når klienten køber en nøgle inde på cspl.dk/account-management.

Hjemmesiden tilbyder mulighed for at forny sin API-nøgle eller erstatte den med den gamle. Hvis en virksomhed har mange ansatte til at bruge én nøgle, ville det være uhensigtsmæssigt at få en ny ved hvert køb. Klienten kan derfor vælge bare at tilkøbe API-kald og derved beholde den gamle nøgle. Hvis nøglen uheldigvis skulle blive lækket, er der også mulighed for at erstatte den nuværende nøgle, og modtage en ny.

Api-nøglen bliver som sagt lavet på frontenden. For at nøglen bliver unik hver gang, laves nøglen ved at hashe den nuværende dato i millisekunder med klokkeslettet og tilføje salt. Biblioteket som er brugt til dette er `bcrypt` [45], som er en npm pakke som bruges til at hashe.

Den hashet værdi er nu klientens API-nøgle. Denne sendes op til firestore via en reference til klientens primærenøgle, hans e-mail.

På backenden, før klientens efterspørgsel om ar få arbejde sat i kø, bliver hans API-nøgle blive valideret. API-nøglen skal både være valid, altså at den findes i databasen, men også have nok kald. Er en af disse to krav ikke opfyldt, ville klienten blive mødt med en fejl-besked om, at hans nøgle ikke er valid eller at der ikke er nok kald tilbage.

I master controlleren, på det endpoint klienten rammer, vil der som noget af det første, kaldes `ValidateAPIKey(dllPackage.APIkey)`. Denne metode får api-nøglen fra CSPL NuGet-pakkens API-Key proptery, hvor der fra UserValidator klassen hentes et snapshot fra firebase og læser værdierne på nøglen, *se figur 91*.

```

@UserValidator.cs
1  public async Task<bool> CheckIfUserHasAValidAPIKey(string APIKey)
2  {
3      Console.WriteLine("Checking API-key validity...");
4
5      Query user = fb.usersRef.WhereEqualTo("APIKey", APIKey);
6
7      QuerySnapshot snapshot = await user.GetSnapshotAsync();
8      if (snapshot.Count == 0 )
9      {
10         return false;
11     }
12     Dictionary<string, object> document = snapshot.Documents[0].ToDictionary();
13     if (document["APIKey"] != null && Convert.ToInt64(document["CallsLeft"]) >= 1)
14     {
15         return true;
16     }
17     else
18     {
19         return false;
20     }
21 }

```

Figure 91: Backend validering af API-nøglen

7.12 Portforwarding

I takt med at clusteret kunne tilgåes udefra af klienter og ikke kræve lokal netværksforbindelse, skal der åbnes nogle porte på det modem som clusteret er forbundet til. De fleste modems og routers har admin paneler, som tillader at åbne porte igennem deres 'portforwarding feature'. I forbindelse med opsætning og udvikling af CSPL systemet, skulle i alt seks porte åbnes bla. til at kunne tilgå Kubernetes clusteret og udviklingsværktøjer.

- Port 443 for HTTPS trafik
- Port 8020 for master servicen
- Port 6443 for Kubeadm admin panel
- Port 85 for TeamCity build server
- Port 3389 for Microsoft Remote Desktop
- Port 22 for SSH

Port 22 er åbnet grundet muligheden for at kunne ssh ind på master- og worker-nodes. Debugging og logs for de diverse pods er nu tilladt via ssh, hvilket betyder en lokal forbindelse ikke er nødvendigt. Da CSPL build-serveren til tider skal have Docker for Desktop opdateret eller kræver nødvendige opdatering, giver port 3389 Windows mulighed for at bruge Remote Desktop Connection. Port 85 er også åbnet og bruges af TeamCity, hvor de forskellige builds/pipelines kan ses. Port 6443 er porten for Kubernetes, hvor de forskellige deployments, services og node status'er kan ses og modificeres. Port 8020 er master-servicens port, hvor 443 er porten for HTTPS. Disse porte router til hindanden og det er her at clusteret kan tilgåes udefra af klienter.

Ses der på figur 92, er der en oversigt over hvordan master-servicen tilgåes. Klienten tilgår via NuGet-pakken domænet cluster.cspl.dk, DNS'en router til IP'en 62.107.0.222 og rammer dermed modem firewall'en på port 443. Normalt er denne port åben som standard, men grundet at denne port skal route til en service, portforwardes den til router IP'en 192.168.87.126 på port 8020. Når denne request rammer routeren, skal der igen opsættes en regel for, hvor trafikken skal dirigeres hen. Hertil portforwardes den til master IP'en 192.168.1.100 på port 8020. Grundet at master-noden og de dertilhørende worker-nodes har deres firewalls deaktivert, rammer klientens request direkte det ønskede endpoint.

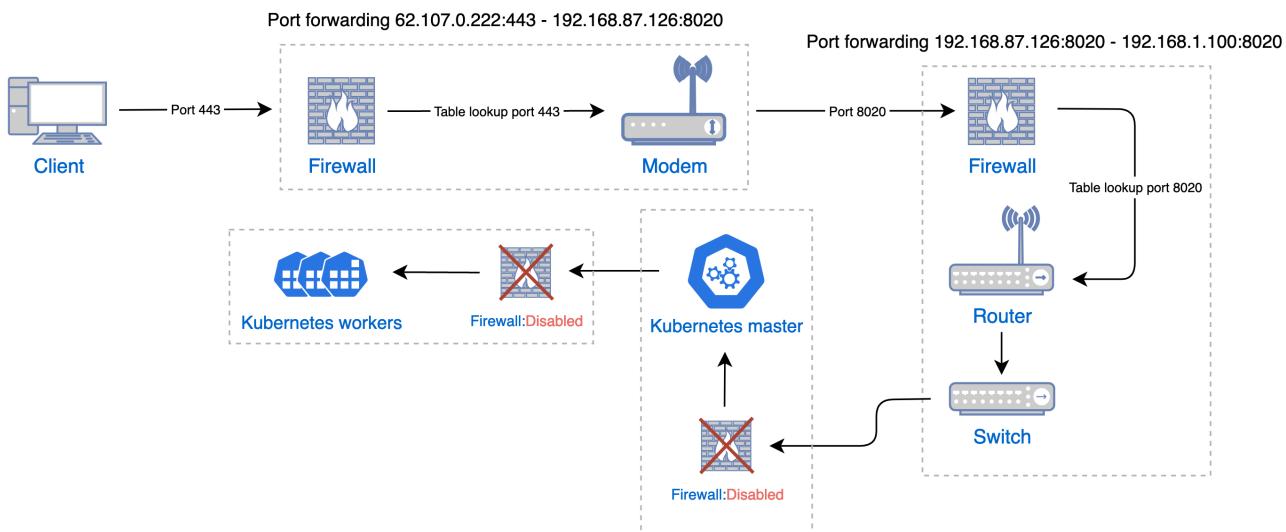


Figure 92: Portforwarding fra klient til master.

7.13 Fejlhåndtering

7.13.1 Generel fejlhåndtering

For et velfungerende program er håndtering af data og brugerinputs vigtigt. CSPL-systemet bygger på at være generisk med type-kalkulationer, type-casting, data bliver serialiseret/deserialiseret og data bliver holdt og håndteret i flere arrays. Alt dette tillader en bred vifte af programmer, at blive eksekveret på clusteret, men vil samtidig åbne døren for risici i form af crashes og fejl.

Igennem master-, worker- og NuGet-komponenterne skal der derfor håndteres fejl på kode sektioner hvor en handling, hvis den skulle fejle, ville crash systemet. Alle controllers og specifikke udvalgte funktioner er blevet wrappet i `try-catch-statements`, hvor funktionaliteten blive eksekveret. Fejler koden, gribes fejlen og den kastede exception delegeres tilbage til brugeren.

Selvom `try-catch-statements` er gode til at fange runtime-fejl, så koster de også system ressourcer og eksekveringstid. Det at kaste/håndtere en exception burde kun benyttes til ekstraordinære situationer, ikke til at håndtere events eller generelt flow-kontrol. Eksempelvis skal en exception ikke kastes hvis brugeren kommer med et forkert input - det forventes at brugere til tider giver forkerte argumenter med.

Da systemet er generisk og `try-catch-statements` findes over flere komponenter, kan det være svært for brugeren at vide hvor det gik galt, hvis ikke der sker et re-throw af exceptions. Af denne grund gribes der ikke kun exceptions, men de kastes også videre, så den først/sidst-kaldte funktion (controllers), kan håndtere de akkumulerede fejl og sende dem til brugeren.

Til yderligere hjælp, er der blevet lavet custom exceptions der bedre fortæller brugeren hvad problemet er. De forskellige custom exceptions nedarver fra `System.Exception` og har passende overloaded constructors, til ønsket brug. Disse custom exceptions bliver kastet sammen med en inner exception, for at give brugeren bedst mulig chancer for at fikse fejlen. Set på figur 93, er vist en af de mange custom exception klasser.

CustomException.cs

```

1 [Serializable()]
2 public class InvalidAPIKeyException : System.Exception
3 {
4     public InvalidAPIKeyException(string message) : base(message) { }
5     public InvalidAPIKeyException(string message, Exception inner) : base(message, inner) { }
6
7     protected InvalidAPIKeyException(SerializationInfo info, StreamingContext context) : base(info, context) { }
8
9     [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
10    public override void GetObjectData(SerializationInfo info, StreamingContext context)
11    {
12        Source = Source.ToUpperInvariant();
13        base.GetObjectData(info, context);
14    }
15 }
```

Figure 93: En custom exception klasse

7.13.2 Klient-exceptions

For at håndtere de exceptions der evt. måtte opstå i det kode som bliver eksekveret i clusteret, som klienten implementerer, er der blevet lavet fejlhåndtering således, at de exceptions bliver "driblet" tilbage til klienten og kastet igen i klientens program. Dette bliver gjort vha. data-modellen ClusterResponse, se figur 94.

ClusterResponse.cs

```

1 internal class ClusterResponse
2 {
3     public object Result { get; set; }
4     public string Exception { get; set; }
5     public string ExceptionType { get; set; }
6     public long ExecutionTime { get; set; }
7 }
```

Figure 94: ClusterResponse

Bliver der kastet en exception i klientens implementation af metoden IClusterParallelizer.Parallel på en worker i clusteret, bliver denne exception fanget, se figur 95. Derefter bliver denne exception (eller en anden) fanget og serialiseret ned i ClusterResponse, inklusiv exception-typen. ClusterResponse bliver returneret/sendt tilbage til master-programmet, se figur 96.

exceptionFromParallel.cs

```

1 try
2 {
3     result = typeResolver.ParallelMethodInfo.Invoke(actualInstance, new object[] { chunk });
4 }
5 catch (System.Reflection.TargetInvocationException e)
6 {
7     throw new FailedToExecuteWork("An error occurred while executing work, exception thrown in"
8                                     "IClusterParallelizer.Parallel, see InnerException for more information."
9                                     , e.InnerException); // Rethrow the exception from the clients parallel method
10 }
```

Figure 95: Fejlhåndtering af exception fra IClusterParallelizer.Parallel i worker-programmet

```
catchExceptionPackClusterResponse.cs

1  try
2  {
3      ...
4  }
5  catch (Exception e)
6  {
7      var clusterResponse = new ClusterResponse
8      {
9          Exception = JsonConvert.SerializeObject(e, new JsonSerializerSettings { TypeNameHandling = TypeNameHandling.All }),
10         ExceptionType = JsonConvert.SerializeObject(e.GetType())
11     };
12
13     return BadRequest(clusterResponse);
14 }
```

Figure 96: Returnering af exception i worker-programmet

Når master-programmet modtager svar i form at et ClusterResponse fra kaldet til worker-API'et, valideres ClusterResponse for om der har været en exception hos worker-programmet. Hvis der bliver en exception kastes den igen, se figur 97.

```
rethrowExceptionFromWorker.cs

1 // Rethrow exception from worker, if present
2 if (clusterResponse.Exception != null)
3 {
4     Type exceptionType = JsonConvert.DeserializeObject<Type>(clusterResponse.ExceptionType);
5     var exception = JsonConvert.DeserializeObject(clusterResponse.Exception, exceptionType,
6                                                 new JsonSerializerSettings { TypeNameHandling = TypeNameHandling.All });
7     throw (Exception)exception;
8 }
```

Figure 97: Validering i master-programmet af evt. exception i ClusterResponse

Denne sekvens af returnering og rethrow af exceptions er også implementeret fra master-programmet til klienten-programmet (ClusterSupportedParallelLibrary-NuGet-pakken). I klientens program hvor klienten eksekverer ClusterRunner.Run, kaster denne metode altså de eventuelle exceptions der måtte være i klientens implementering af IClusterParallelizer.Split, IClusterParallelizer.Parallel og IClusterParallelizer.Collect. Klienten har dermed mulighed for selv at fange og håndtere disse fejl, der bliver kastet i clusteret. Se den fulde sekvens på figur 98.

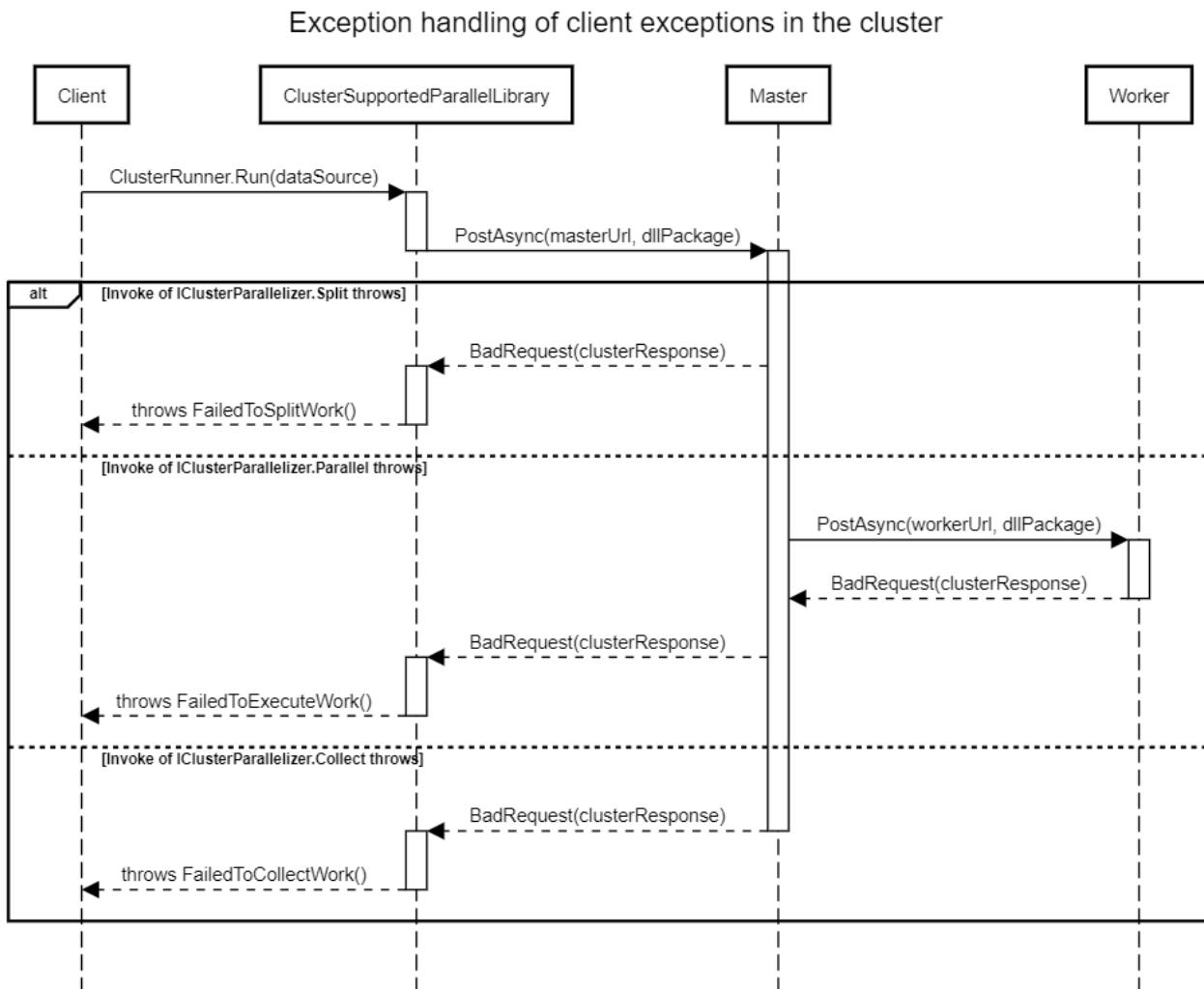


Figure 98: Sekvensdiagram for fejlhåndtering af klient-exceptions i clusteret.

For information om integrationstest af denne fejlhåndtering se afsnit 9.5.3.

8 Kubernetes installation and opsætningsguide

Denne guide viser hvordan man kan installere og opsætte et fungerende Kubernetes cluster bestående af en amd64 maskine som master-node og flere arm Raspberry Pi's som worker-noder.

8.1 Prerequisite

- All the devices/nodes must be assigned static IP addresses on the network that are being used.

8.2 Install OS

Master

- Install Ubuntu 18.04.4 LTS on the master node (amd64).

Raspberry Pi

- Install Raspbian Buster Lite February 2020 on the worker nodes/Pi's (arm).

8.3 Enable SSH

Master

- Run the following commands.

```
$ sudo systemctl enable ssh
$ sudo systemctl start ssh
```

Raspberry Pi

- Create an empty file on the microSD card named ‘ssh’ or run the following commands.

```
$ sudo systemctl enable ssh
$ sudo systemctl start ssh
```

8.4 Disable firewall

Master

- Run the following command.

```
$ sudo ufw disable
```

- Reboot

```
$ sudo reboot
```

Raspberry Pi

- Create a startup script on the Pi’s.

```
$ sudo nano /etc/init.d/iptablesScript.sh
```

- Copy the following to the file and save.

```
#!/bin/sh
sudo iptables -P FORWARD ACCEPT
```

- Make the file executable.

```
$ sudo chmod 777 /etc/init.d/iptablesScript.sh
```

- Create this file.

```
$ sudo nano /lib/systemd/system/startup.service
```

- Copy the following to the file and save.

```
[Unit]
Description=My Startup Service After=multi-user.target

[Service]
Type=idle
ExecStart=/etc/init.d/iptablesScript.sh

[Install]
WantedBy=multi-user.target
```

- Run the following commands.

```
$ sudo chmod 644 /lib/systemd/system/startup.service
$ sudo systemctl daemon-reload
$ sudo systemctl enable startup.service
```

- Reboot

```
$ sudo reboot
```

8.5 Edit host name

Master

- Edit this file and change the hostname to ‘master-node’ (you may choose a different name) and save the file.

```
$ sudo nano /etc/hosts
```

- Run this command to change the hostname in */etc/hostname*.

```
$ sudo hostnamectl set-hostname master-node
```

Raspberry Pi

- Edit this file and change the hostname to ‘pi01’ (or ‘pi02’, depending on how many Pi’s you are using) and save the file.

```
$ sudo nano /etc/hosts
```

- Run this command to change the hostname in */etc/hostname*.

```
$ sudo hostnamectl set-hostname pi01
```

8.6 Configure cgroup boot options

Master & Raspberry Pi

- Create or edit this file.

```
$ sudo nano /boot/cmdline.txt
```

- Append this to the end of the file.

```
cgroup_enable=cpuset cgroup_memory=1 cgroup_enable=memory
```

8.7 Install updates

Master & Raspberry Pi

- Run the following command.

```
$ sudo apt update && sudo apt dist-upgrade
```

8.8 Permanently disable swap

Master

- Run the following command.

```
$ sudo swapoff -a && sudo sed -i.bak '/ swap / s/^(\.*\)$/#\1/g' /etc/fstab
```

- Reboot

```
$ sudo reboot
```

Raspberry Pi

- Run the following commands.

```
$ sudo dphys-swapfile swapoff  
$ sudo dphys-swapfile uninstall  
$ sudo apt purge dphys-swapfile
```

- Reboot

```
$ sudo reboot
```

8.9 Install Docker

Master & Raspberry Pi

- Install curl (if not already installed).

```
$ sudo apt install curl
```

- Run the following commands.

```
$ curl -sSL get.docker.com | sh  
$ sudo usermod -aG docker <username>
```

8.10 Configure Docker daemon options

Master & Raspberry Pi

- Create or edit this file.

```
$ sudo nano /etc/docker/daemon.json
```

- Copy this to the file.

```
{  
  "exec-opts": ["native.cgroupdriver=systemd"],  
  "log-driver": "json-file",  
  "log-opt": {  
    "max-size": "100m"  
  },  
  "storage-driver": "overlay2"  
}
```

- Reboot.

```
$ sudo reboot
```

8.11 Setup Kubernetes repository

Master & Raspberry Pi

- Create or edit this file.

```
$ sudo nano /etc/apt/sources.list.d/kubernetes.list
```

- Append this to the file.

```
deb http://apt.kubernetes.io/ kubernetes-xenial main
```

- Add GPG key.

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
```

8.12 Install Kubernetes.

Master & Raspberry Pi

- Update, run the following command. Make sure that this completes successfully, if not try to run the command again.

```
$ sudo apt update
```

- Install kubernetes.

```
$ sudo apt install kubeadm kubectl kubelet
```

8.13 Setup Kubernetes.

Master

- Set */proc/sys/net/bridge/bridge-nf-call-iptables* to 1. Run the following command.

```
$ sysctl net.bridge.bridge-nf-call-iptables=1
```

- Initialize kubeadm.

```
$ sudo kubeadm init --pod-network-cidr=10.244.0.0/16 --v=5
```

- Setup directories, run the following commands. If the output text of *kubeadm init* recommends different commands, run those instead and skip this step.

```
$ mkdir -p ~.kube
$ sudo cp /etc/kubernetes/admin.conf ~/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

8.14 Setup pod network.

Master

- Setup flannel pod network.

```
$ kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/2140ac876ef134e0ed5 \
→ af15c65e414cf26827915/Documentation/kube-flannel.yml
```

- Check if all pods are up and running.

```
$ kubectl get pods --all-namespaces -o wide
```

- Add anonymous role binding.

```
$ kubectl create clusterrolebinding cluster-system-anonymous --clusterrole=cluster-admin \
→ --user=system:anonymous
```

8.15 Join worker nodes to the Kubernetes cluster

Raspberry Pi

- Join all Pi's to the cluster using the join command from the text output of `kubeadm init`. It should look something like this.

```
$ kubeadm join <master-ip-address>:<port> --token <token>
→ --discovery-token-ca-cert-hash <token>
```

- Check if all nodes status are ready.

```
$ kubectl get nodes -o wide
```

8.16 Cluster SSH

Til opsætning af Raspberry Pi og Kubernetes er der blevet brugt XQuartz [67] til clustet SSH. I stedet for manuelt at ssh til hver eneste Raspberry Pi, kan et script skrives til at ssh ind på hele clusteret. På figur 99 er der vist en billede af alle 20 Raspberry Pi shells, som er forbundet til med cluster SSH.

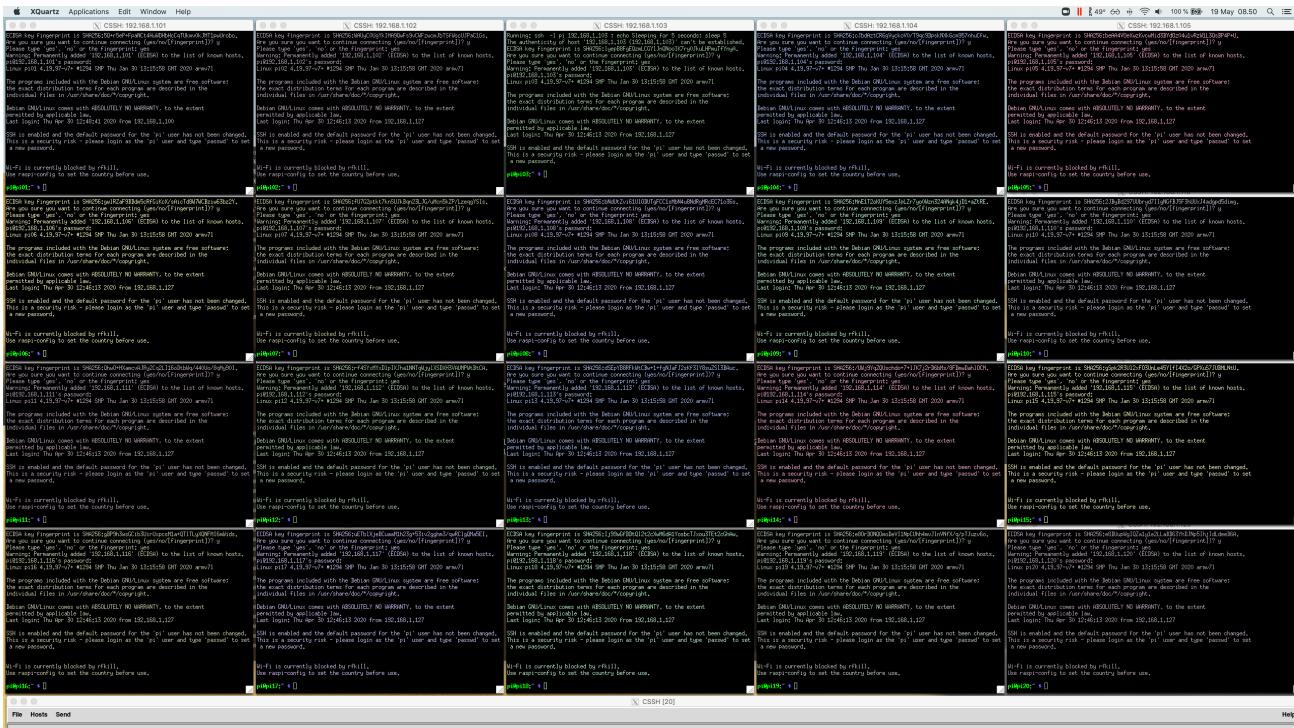


Figure 99: XQuartz brugt til cluster SSH

9 Test

9.1 Introduktion

Følgende afsnit beskriver forskellige former for test, der er blevet lavet for at sikre kvalitet af CSPL systemet, heriblandt unit tests, integration tests, maueltests, performance test og accepttest. Før CSPL systemets udvikling blev igangsat, var en TeamCity byggeserver allerede opsat med automatiske pipelines til at kører auto tests, således CSPL udviklerne via continues integration kunne sikre der løbende under udviklingen af ny funktionalitet ikke blev ødelagt allerede eksisterende funktionalitet. NUnit test frameworket er blevet brugt til implementere unitest for komponenter, samt integrationstest. Derudover er NSubstitute blevet brugt for at stubbe forskellige system dependencies ud under unittest og integrationstest.

9.2 ManuelTest

For at teste funktionalitet som har ikke har kunne blive dækket med automatiseret tests, har manuel testing været brugt.

9.2.1 Manuel test - Adminstrative produkt

Når nyt funktionalitet er blevet implementeret, har en af CSPL udviklerne som ikke har lavet implementationen, stået som tester. At have andre øjne på funktionaliteten vil typisk give et bedre udbytte med fund af kritiske fejl og diskussioner om ev.t forbedringer. Manuelt testing har af denne grund været et essentielt værktøj til kvalitets sikring af produktet.

9.2.2 Monkey tests

Da alle CSPL udviklerne har en bias til produktet og af denne grund forventes at produktet bruges på en specifik måde, kan det være en fordel at have en ekstern udvikler til at afprøve produktet. En ekstern udvikler vil kunne give feedback i form af ideer og forbedring på både hjemmesiden og backend. Af denne grund har en software udvikler fra 6. semester været med til at afprøve systemet.

9.3 Feedback

For at få konstruktiv feedback på det byggede CSPL system, er der indblandet en software ingenør from IHA på 6 semester, navnet angives anonymt gennem testen som "Test person". Test personen fik til opgave at bygge et simpelt program som kunne summere x-antal tal parallelt på clusteret gennem CSPL nuget pakken. Selve testen blev lavet over discord, således der kunne screenshares hvis det var en nødvendigehed.

Download Cluster Supported Parallel Libraries

Test personen kunne ikke download Cluster Supported Parallel libraries fra nuget.org da nuget pakken havde dependencies på andre nuget pakker fra private registries. Test personen fik derfor hurtigt udlevede DLL'erne til nuget pakken således test personen kunne starte implementationen af det simple summerings program.

Create account

Test personen havde svært ved at lave en account, da der under login ikke var en "Dont have an account yet create here" section, som test personen fremhævede at der ofte var på andre sider.

Implementation

Test personen fik hurtigt implementeret interfacet som var nødvendig for summerings programmet. Det givet example på cspl.dk hjemmeside var lidt svært at forstå, og kunne godt være mere simplet. Ellers var implementation ok når man først forstod hvordan systemet fungerede.

Test personen var forviret over at han ikke kunne lave custom properties, og sætte dem i hans program. Dette blevet forklaret til test personen som en known issue.

Documentaion

Test personen, var forvirret over hvad forskellen var på pods/workers/nodes under documentationen.

Final feedback Test personen var positivt overrasket, at det implementerede program bare virkede, da det blev distribueret ud på x-antal rasberry Pi's. Test personen syntes også det var super lækkert at han kunne se hans exceptionens live på hjemmesiden, da han requestede for mange workers i clusteret end hvad der var kapacitet til. Generelt var det lidt svært at finde ud af det forskellige typer i det generiske interface, men efter han have implementeret det gav det generelt god mening. Test personen mente at hvis han skulle implementet et nyt program med cluster supported parallel libraries ville det være langt nemmere.

9.4 Modultest

I takt med at programmerne og de dertilhørende komponenterne bliver udviklet, implementeret og refaktureret, bliver der skrevet unittests af de komponenter, der er blevet vurderet til at være nødvendige at teste på. Der bliver udført code coverage analyser for at have en pejling om hvor fyldestgørende disse test er. Ift. code coverage bliver der taget udgangspunkt i coverage af blocks, som er defineret ved de stykker af kode som har ét enkelt indgangs- og udgangspunkt [41].

De følgende afsnit er d. 18-05-2020 markeret som forældet, flere unittests er kommet til afsnittet blev skrevet, code coverage kan derfor være anderledes.

9.4.1 Master

9.4.1.1 LoadBalancer

Loadbalanceren er det komponenten der holder styr på hvilke pod IP'er der findes i CSPL systemet, samt også holderstyr på hvilke der er ledelige eller optaget. Her blandt filtre døde og nye pods til CSPL systemet. For at sikre disse avancerede funktionaliteter er af følgende auto test blevet lavet for LoadBalanceren set på figur 100. Det ses blandt andet hvordan testene sikre dediktering af døde pods, samt forskellige tests hvor der requeste workers pods.

	LoadBalancerTest (10)	
✓	CanInitPassPodsIpsWithStatusToDictionary	10 sec
✓	CanLoadBalancerDetectDeadPod	209 ms
✓	CanLoadBalancerReturnAvailablePodAndMarkItCorrectly	25 ms
✓	CanLoadBalancerReturnBulkOfPodIpsAndMarkAsOccupied	5 ms
✓	CanLoadBalancerReturnNullIfNoPodsAreAvailable	6 ms
✓	CanLoadBalancerSetStatusAvailableForBulksToMarkPodsAsFinnishedCorrectly	< 1 ms
✓	CanLoadBalancerSetStatusAvailableToMarkPodAsFinnishedCorrectly	< 1 ms
✓	CanLoadBalancerUpdatePodsAndRunWork	< 1 ms
✓	CanLoadBalancerWaitForAvailablePods	10 sec
✓	CanLoadBalancerThrowIfRequestedNrOfPodsIsTooBig	9 ms

Figure 100: LoadBalancer tests

Da LoadBalanceren gør brug af KubernetesFetcher til at hente live data fra kubernestes clusteret, er der brugt dependency injection i form at constructor injecting, således LoadBalanceren bliver et isoleret test komponent. På figur 101 ses det at gør LoadBalanceren brug af interface IKubernetesFetcher, således kan der stubbes en fake udgave ud af KubernetesFectheren som returner kendte data de forskellige tests kan operere på.

```
④ LoadBalancer.cs
1  public LoadBalancer(IKubernetesFetcher kubernetesFetcher, uint maxPodSize)
2  {
3      this.kubernetesFetcher = kubernetesFetcher;
4      MAXPODSIZE = maxPodSize;
5      Init();
6  }
```

Figure 101: LoadBalancer Constructor injection

9.4.1.2 KubernetesFecther

KubernetesFecther komponenten henter live data fra kubernetes API'et, samt laver de kommunikationer med kubernetes API'et så som er nødvendige. Heriblandt sendes requests til kubernestes API'er for at genstarte brugte worker pods som kan være forurende. På figur 102 test for restarting af worker pods.

	KubernetesFetcherTest (2)	
✓	CanKubernetesFetcherDeleteBulkOfPods	11 sec
✓	CanKubernetesFetcherDeleteOnePod	7 sec

Figure 102: Kubernetes Fetcher tests

9.4.1.3 UserValidator

UserValidatoren er det komponent der står for at validere klientes API-key. Derfor er test lavet for at sikre validering af både ikke gyldige API-keys og gyldige API-keys. På figur 103 ses de forskellige unit test som er blevet lavet for sikre API-key validering.

✓ UserValidatorTest (6)	2 sec
✓ TestAPIKeyEnvironmentVariablesAreNotSet	7 ms
✓ TestAPIKeyEnvironmentVariablesAreSet	2 ms
✓ TestAPIKeyFailsWhenCallsLeftIs0	522 ms
✓ TestAPIKeyIsInvalid	188 ms
✓ TestAPIKeyIsValid	411 ms
✓ TestConnectionWithGoogleFireBase	931 ms

Figure 103: UserValidator tests

9.4.1.4 ClusterQueue

ClusterQueue er selve køen hvor at klienter kommer i kø for at få workers til deres arbejde. Derfor er der lavet test for at klienter bliver lagt korrekt i køen, samt kommer korrekt ud af køen. På figur 104 ses testen for korrekt køstruktur.

✓ ClusterQueueTest (1)	9 ms
✓ CanQueueEnqueWorkers	9 ms

Figure 104: ClusterQueue test

9.4.1.5 WorkDelegator

WorkDelegator er det komponent på master-programmet som sørger for, at delegerer bidder af arbejde ud til de x-antal worker-programmer, der skal eksekvere det kode/program som brugeren har sendt til master-programmet. Det har i en testsammenhæng derfor været nødvendigt at implementere et interface, som omslutter HttpClient'en (forbindelsen til worker-programmerne) således at der i unit tests, kan laves et substitut for denne og dermed injecte den i WorkDelegator konstruktoren. Der er lavet en unit test for hver metode i WorkDelegator-klassen, som hver især er uafhængige af hinanden. Code coverage af WorkDelegator kan ses på figur 105.

Hierarchy	Covered (% Blocks)	Covered (Blocks)	Not Covered (% Blocks)	Not Covered (Blocks)
WorkDelegator	100,00 %	77	0,00 %	0
WorkDelegator(Master.Helpers.ICustomHttpClient, CSPL.Common.Models.DllPackage, ...)	100,00 %	4	0,00 %	0
Run()	100,00 %	5	0,00 %	0
PrepareChunksForWorkers()	100,00 %	7	0,00 %	0
Delegate(System.Collections.IList)	100,00 %	45	0,00 %	0
Collect(System.Threading.Tasks.Task<System.Net.Http.HttpResponseMessage>[])	100,00 %	16	0,00 %	0

Figure 105: Code coverage af WorkDelegator.

9.4.2 ClusterSupportedParallelLibrary (NuGet-pakke)

9.4.2.1 ReferencedAssembliesFilter

ReferencedAssembliesFilter er et komponent som samler alle DLL/assembly-dependencies for en given DLL-fil/assembly, dette gøres op imod et filter af assemblies som allerede er en del af .NET Core. Unit tests af komponenten tager udgangspunkt i et opstillet 'TestProject', som er et .NET Core projekt med diverse dependencier til andre projekter og NuGet-pakker. Code coverage af ReferencedAssembliesFilter kan ses på figur 106.

Hierarchy	Covered (% Blocks)	Covered (Blocks)	Not Covered (% Blocks)	Not Covered (Blocks)
{ } CSPL.Components	100,00 %	54	0,00 %	0
ReferencedAssembliesFilter	100,00 %	51	0,00 %	0
GetAllReferencedAssembliesFromDll(string)	100,00 %	36	0,00 %	0
ReferencedAssembliesFilter()	100,00 %	15	0,00 %	0
ReferencedAssembliesFilter.<>c_DisplayClass2_0	100,00 %	3	0,00 %	0
<GetAllReferencedAssembliesFromDll>b__0(string)	100,00 %	3	0,00 %	0

Figure 106: Code coverage af ReferencedAssembliesFilter.

9.4.3 CSPL.Common (NuGet-pakke)

9.4.3.1 TypeResolver

TypeResolver er et delt komponent mellem Master- og Workerprogrammerne. Komponentet har bla. den funktion at bestemme/finde typer på retur værdier og argumenter for en klasse, som har implementeret IClusterParallelizer, ud fra en given assembly. Derudover har den nogle metoder til lave instancer af generiske typer. Hver metode i klassen, samt konstruktoren, er testet med unit tests, som tager udgangspunkt i en tilfældig implementering af IClusterParallelizer med diverse brugerdefinerede og build-in typer. Code coverage af TypeResolver kan ses på figur 107.

Hierarchy ▲	Covered (% Blocks)	Covered (Blocks)	Not Covered (% Blocks)	Not Covered (Blocks)
• {} CSPL.Common.TypeResolver	100,00 %	51	0,00 %	0
• ↗ TypeResolver	100,00 %	51	0,00 %	0
∅ CreateListOfGenericType(System.Type)	100,00 %	5	0,00 %	0
∅ CreateInstanceOfType(System.Type)	100,00 %	3	0,00 %	0
∅ GetTypeFromJson(string, System.Type)	100,00 %	3	0,00 %	0
∅ TypeResolver(System.Reflection.Assembly, string)	100,00 %	30	0,00 %	0
∅ get_ClientClusterParallelizerType()	100,00 %	1	0,00 %	0
∅ get_CollectArgsType()	100,00 %	1	0,00 %	0
∅ get_CollectMethodInfo()	100,00 %	1	0,00 %	0
∅ get_CollectReturnType()	100,00 %	1	0,00 %	0
∅ get_ParalellMethodInfo()	100,00 %	1	0,00 %	0
∅ get_ParallelArgsType()	100,00 %	1	0,00 %	0
∅ get_ParallelReturnType()	100,00 %	1	0,00 %	0
∅ get_SplitArgsType()	100,00 %	1	0,00 %	0
∅ get_SplitMethodInfo()	100,00 %	1	0,00 %	0
∅ get_SplitReturnType()	100,00 %	1	0,00 %	0

Figure 107: Code coverage af TypeResolver.

9.5 Integrationtest

For at løbende sikre at CSPL systemet kan parallelisere forskellige algoritmer, er der udviklet fuld automatisk system integrations tests. Når nyt funktionalitet er blevet tilføjet til CSPL systemet, vil bygge serveren automatisk starte en fuld integrations test på CSPL systemet, ved at starte en job på clusteret, sende fejl eller succes tilbage. På denne måde sikres at nye features ikke bryder eksisterende funktionalitet. På figur 108 ses hvorledes TeamCity buildServer poller github om nye features som bliver mergeret til master branchen på github. Når en ny feature bliver lagt på master vil TeamCity starte et job på clusteret, samt give fejl and succes tilbage. Da CSPL system kun har haft 20 raspberry Pi's til rådighed er disse system tests blevet køre på productions clustet. I en større og fremtide udviklings proces, ville det have været bedre at have en sekundært cluster hvor disse test kunne blive kørt, for at unga unødvendig load og fejl på produotions clusteret.

Oversigt er de forskellige IntegrationTests i CSPL systemet:

- **NugetPackage:**
 - BruteForceTest
 - MonteCarloTest
 - ExecutionHandlingTest

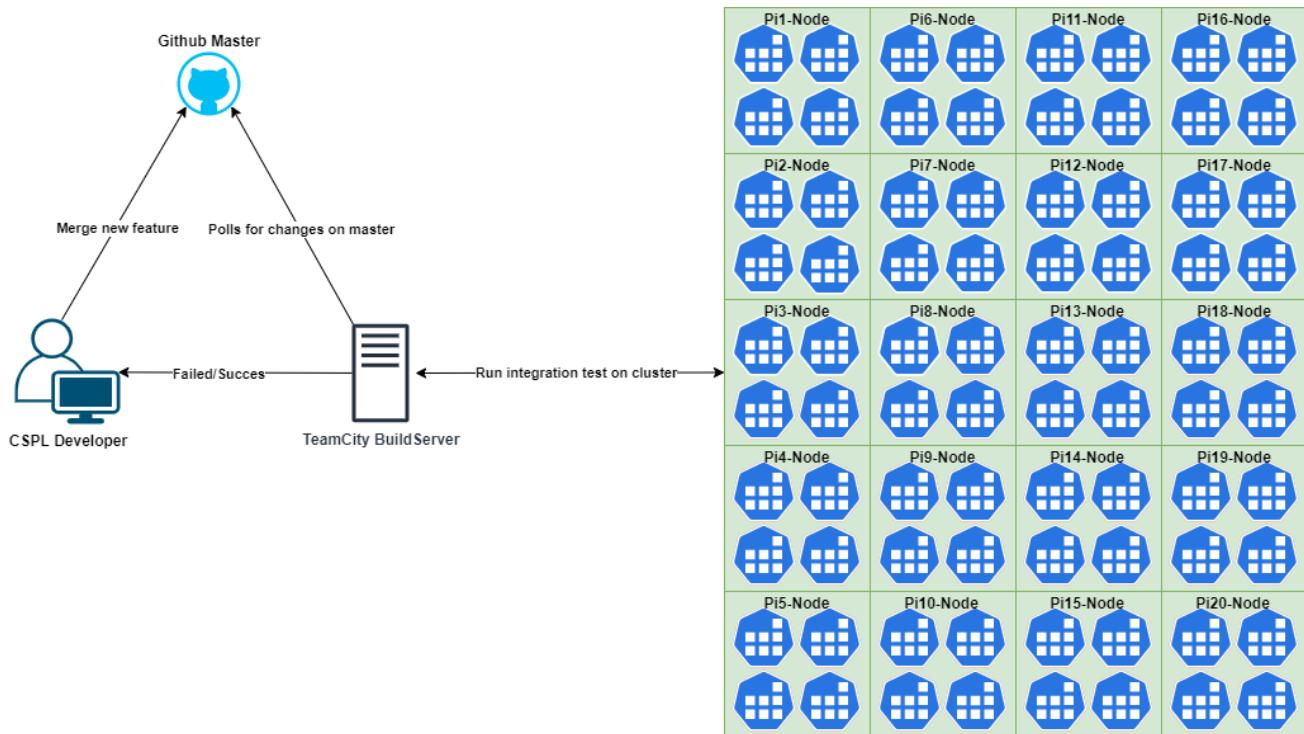


Figure 108: Fuld system integrations test

9.5.1 MonteCarlo

Hver gang et merge bliver lavet til til master branchen vil forskellige mortecarlo tests blive kørt på clusteret. Set på figur 109, ses de forskellige

CSPL.Test.MonteCarloTest (5)	1 min
MonteCarloTest (5)	1 min
CanMonteCarloShowCorrectException	1 sec
MonteCarlo10PodsTest	8 sec
MonteCarlo8BilTest	49 sec
MonteCarloDefaultValues	16 sec
MonteCarloThisPCTest	

Figure 109: Montecarlo integration test

9.5.2 Brute force hash

Hver gang et merge bliver lavet til til master branchen vil brute-force algoritmen blive kørt på clusteret. Set på figur 120, ses testen for brute-force algoritmen.

Test Name:	BruteForceFromASCII48To122TestCLUSTER
Test Outcome:	Passed
Standard Output	
Elapsed time:	362s
Result:	psswd => 101ec841001c02973350d0c2fee6a3a57ad855b8dc0a534522b7e20a1541ddbc

Figure 110: Brute force hash på 74 pods i clusteret.

9.5.3 Fejlhåndtering af klient-exceptions

Som beskrevet i afsnit 7.13.2 er der implementeret fejlhåndtering af exceptions fra klientens implementerede kode, som eksekveres på clusteret. I den forbindelse er der skrevet integrations test af dette, som på det fulde system/cluster tester de scenarier hvor en exception bliver kastet i klientens kode. En sådan test kan se såldes ud, se figur 111, hvor der bliver kastet en System.IndexOutOfRangeException exception i IClusterParallelizer.Parallel.

```

FejlhaandteringAfKlientExceptions.cs

1  [Test]
2  public void ExceptionFromParallelTest()
3  {
4      var clusterRunner = new ClusterRunner<List<float>, List<float>, float, float>(new ClusterParellizerThrowsInParallel());
5
6      Exception exception = Assert.Throws<CSPL.Common.ExceptionMapper.FailedToExecuteWork>(() => clusterRunner.Run(new List<float>()));
7      Assert.That(exception.InnerException, Is.TypeOf<System.IndexOutOfRangeException>());
8  }
9
10 class ClusterParellizerThrowsInParallel : IClusterParallelizer<List<float>, List<float>, float, float>
11 {
12     ...
13
14     public float Parallel(List<float> subDataSource)
15     {
16         throw new System.IndexOutOfRangeException("IndexOutOfRangeExceptionInParallel");
17     }
18
19     ...
20 }

```

Figure 111: Test af exception på clusteret kastet i klientens implementering af IClusterParallelizer.Parallel

på figur 112 ses de forskellige exceptions integrations tests, som bliver kørt på clusteret hver gang en ny feature bliver mergeret til masten branchen.

◀ ✓ CSPL.Test.ExceptionHandlingTest (6)	16 sec
◀ ✓ ExceptionHandlingTest (6)	16 sec
✓ CustomInnnerExceptionTest	1 sec
✓ ExceptionFromCollectTest	6 sec
✓ ExceptionFromParallelTest	6 sec
✓ ExceptionFromSplitTest	1 sec
✓ InvalidAPIKeyExceptionTest	422 ms
✓ SystemInnnerExceptionTest	1 sec

Figure 112: Exception integrations test

9.6 Performancetest

I et system som har til opgave at forbedre performance, giver det mening af teste hvor godt det performer i praksis. Testene vil måle systemets ydeevne i nogle forskellige scenarier, med forskellige implementationer af IClusterParallelizer interfacet. Forskellige faktorer som antallet af noder, HTTP/HTTPS, belastning af clusteret, typen af arbejde og forskellige datasæt. Derudover vil der blive diskuteret om der findes et overhead, ved at bruge et system som dette og i hvilke sammenhænge det potentielt kunne opstå.

Følgende performancetests vil sammenligne CSPL systemet med en laptop Lenovo T480s og en mere kraftig desktop. Figur 113 viser en oversigt over CSPL clusterets hardware og det eksterne hardware brugt til benchmarkstests.

Følgende tests er også beskrevet og vist på hjemmesiden under 'Proof of Work'
<https://cspl.dk/proof-of-work>.

Primary cluster HW:

Platform (device)	Role	Pcs.	Cost pr. pcs.	OS	CPU	Cores	MHz
Raspberry Pi 3B	worker	15	280kr.	Raspbian Buster Lite	Broadcom BCM2837	4	1.200 Mhz
Raspberry Pi 3B+	worker	5	320kr.	Raspbian Buster Lite	Broadcom BCM2837B0	4	1.400 Mhz
J5005 mini-ITX	master	1	1.048,00 kr.	Ubuntu Desktop	Intel Quad Core Pentium Silver J5005	4	1.500 Mhz

Supportive cluster HW:

Platform (device)	Role	Pcs.	Cost pr. pcs.	Speed
Cisco SG112-24	switch	1	477,00 kr.	10/100/1000 MBps
Cisco Linksys WRT320N	router	1	217,00 kr.	10/100/1000 MBps
Kingston SSDNow A400 SSD	master SSD	1	242,00 kr.	500 MBps(read) / 320 MBps (write)
CAT.5e	LAN cables	24	~ 22,00 kr.	1000 MBps

External HW. for benchmarking:

Platform (device)	Role	Pcs.	Cost pr. pcs.	OS	CPU	Cores	MHz
Desktop	Tester	1	11.000 kr.	Windows 10	64-bit AMD Ryzen 7 2700X	8	3.800 Mhz
Laptop Lenovo T480s	Tester	1	10.000kr.	Windows 10	64-bit Intel i7-8550U	4	1.990 Mhz

Figure 113: Test Specs

Prisforsk

- **Laptop Lenovo T480s** 10.000kr.
- **Desktop** 11.000kr.
- **CSPL System - 20 nodes** 7.000kr.

9.6.1 Raspberry Pi 3B vs. Raspberry Pi 3B+

Da CSPL clusteret både har to forskellige Raspberry Pi versioner som worker nodes, kan klienten komme ud for, at hans program bliver distribueret ud på både Raspberry Pi 3B og Raspberry Pi 3B+. Klienten kan også komme ud for, at han kun får distribueret arbejde ud på workers af typen Raspberry Pi 3B eller Raspberry Pi 3B+. Hvis klienten får en blanding af både Raspberry Pi 3B+ og 3B vil Raspberry Pi 3B+ workers blive hurtigere færdig med deres arbejde, end de workers som kører på Raspberry Pi 3B. Klientens arbejde bliver først endeligt færdigt når alle workers har udregnet hver deres sub-resultat af arbejdet, såldes at det endelige resultat kan samles. Generelt betyder det, at klientens job i få tilfælde kan være hurtigere, hvis klienten er heldig nok kun at få tildelt workers som kører på Rasberry Pi 3B+.

Når klienten anmelder CSPL systemet med et job er der følgende muligheder for eksekveringer:

- **Raspberry Pi 3B workers:** Normal eksekveringstid
- **Raspberry Pi 3B+ workers:** Hurtigere eksekveringstid
- **Mix af 3B og 3B+ workers:** Normal eksekveringstid

For at sammenligne performance for de scenerier, hvor klienten kun får tildelt Raspberry Pi 3B+ workers, er der eksekveret et identisk job på henholdsvis 3B+ og 3B, i dette tilfælde er det Monte Carlo Pi estimering [1]. Tidforskellen for det eksekverede job ses på figur 114. Programmet er både testet med 1, 2 og 4 tråde, hosted udenfor Kubernetes, og derefter testet i CSPL systemet med 20 workers for at se tidforskellene.

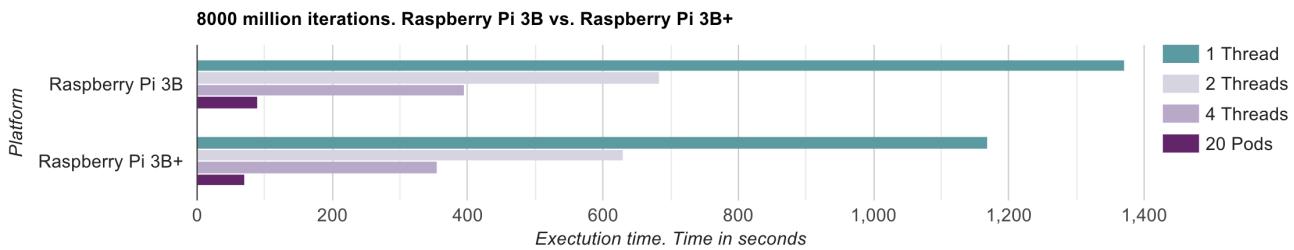


Figure 114: Raspberry Pi 3B vs. Raspberry Pi 3B+ performance.

Performance forskellen beregnes i procent med følgende formel:

$$\left(\frac{\text{Time}_{3B}}{\text{Time}_{3B+}} - 1 \right) \cdot 100 = \% \text{Performance Increase} \quad (15)$$

1 Thread 3B vs. 3B+

$$\left(\frac{1370,69s}{1169,53s} - 1 \right) \cdot 100 = 17,20\% \quad (16)$$

2 Thread 3B vs. 3B+

$$\left(\frac{683,96s}{630,56s} - 1 \right) \cdot 100 = 8,47\% \quad (17)$$

4 Thread 3B vs. 3B+

$$\left(\frac{396,28s}{354,82s} - 1 \right) \cdot 100 = 11,68\% \quad (18)$$

20 workers 3B vs. 3B+

$$\left(\frac{89,92s}{70,23s} - 1 \right) \cdot 100 = 28,06\% \quad (19)$$

Gennemsnitlig performance forbedring for Raspberry 3B+

Ud fra følgende test ses det at Raspberry Pi 3B+ er hurtige end Raspberry Pi 3B grundet deres forskellige hardware specifikationer. På beregning 20 ses den gennemsnitlig forøgelse af performance mellem Raspberry Pi 3B+ og Rasberry Pi 3B. Med kun fire antal samplings, vil gennemsnits procenten være upræcis, men Raspberry Pi 3B+ vil cirka have en performance forøgelse på 16,35% ift. Raspberry Pi 3B. Dette betyder at en klient kan være heldig at få en 16,35% hurtigere eksekverings tid, hvis klienten kun får tildelt workers, som kører på Rasberry Pi 3B+.

$$\text{average} = \left(\frac{17,20\% + 8,47\% + 11,68\% + 28,06\%}{4} \right) = 16,35\% \quad (20)$$

9.6.2 Docker overhead vs. normal proces

For er teste hvad det koster at hoste de CSPL systemet i Docker containers, i forhold til at køre programmet som en normal proces, er der blevet kørt et identisk job, i dette tilfælde er det Monte Carlo Pi estimering [1], i hhv. en Docker container og som en normal process. Følgende test foregår på både Raspberry Pi 3B og 3B+. På figur 115 ses forskellene i eksekveringstiden mellem en hostet proces i en Docker container og en normal proces.

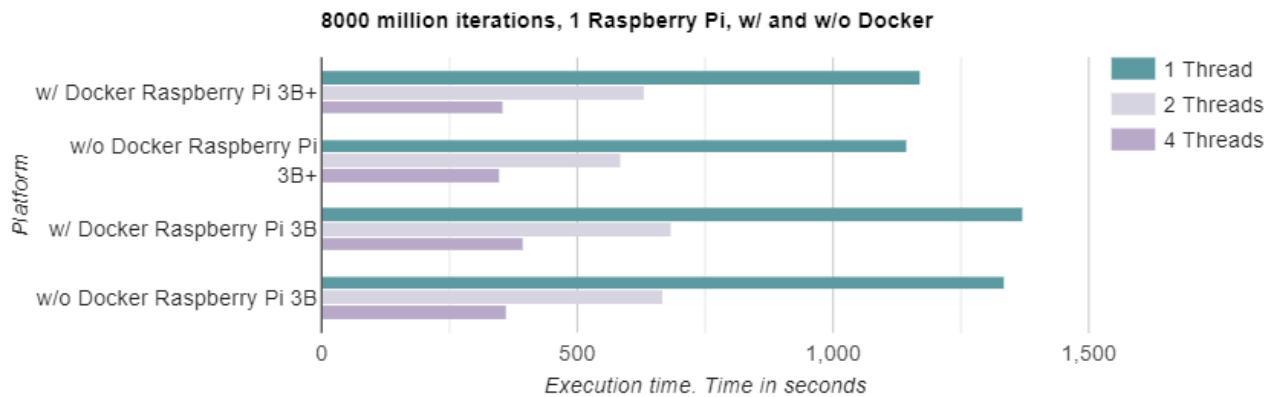


Figure 115: Docker container vs. normal proces.

Overhead beregnes i procent med følgende formel:

$$\left(\frac{Time_{container}}{Time_{proces}} - 1 \right) \cdot 100 = \%Overhead \quad (21)$$

1 Thread 3B

$$\left(\frac{1370,7s}{1334,64s} - 1 \right) \cdot 100 = 2,70\% \quad (22)$$

2 Thread 3B

$$\left(\frac{683,97s}{668,01s} - 1 \right) \cdot 100 = 2,39\% \quad (23)$$

4 Thread 3B

$$\left(\frac{396,28s}{362,04s} - 1 \right) \cdot 100 = 9,46\% \quad (24)$$

1 Thread 3B+

$$\left(\frac{1169,53s}{1145,47s} - 1 \right) \cdot 100 = 2,10\% \quad (25)$$

2 Thread 3B+

$$\left(\frac{630,65s}{585,76s} - 1 \right) \cdot 100 = 7,66\% \quad (26)$$

4 Thread 3B+

$$\left(\frac{354,81s}{347,85s} - 1 \right) \cdot 100 = 2,00\% \quad (27)$$

Gennemsnitlig overhead

Ud fra ovenstående test ses det, at der er et lille overhead ved at hoste de forskellige processer i en Docker container. I beregning 28 ses et gennemsnitligt overhead på cirka 4,39% når processer skal hostes i Docker containerne. Procenten er kun beregnet ud fra seks samplings, og vil derfor være en smule upræcis, dog ses der stadig et lille overhead.

$$average = \left(\frac{2,70\% + 2,39\% + 9,46\% + 2,10\% + 7,66\% + 2,00\%}{6} \right) = 4,39\% \quad (28)$$

9.6.3 CSPL with Kubernetes vs. Docker container overhead

For at teste hvad det koster at lade Kubernetes hoste de forskellige processer i pods, samt det overhead der ligger i at finde og load assemblies på runtime, for derefter at eksekvere dele af disse assemblies via reflection, er MonteCarlo-algoritmen blevet testet i CSPL systemet med Kubernetes vs. hostet i en Docker container. På figur 116 ses forskellene i eksekveringstiderne mellem Kubernetes med CSPL og Docker.

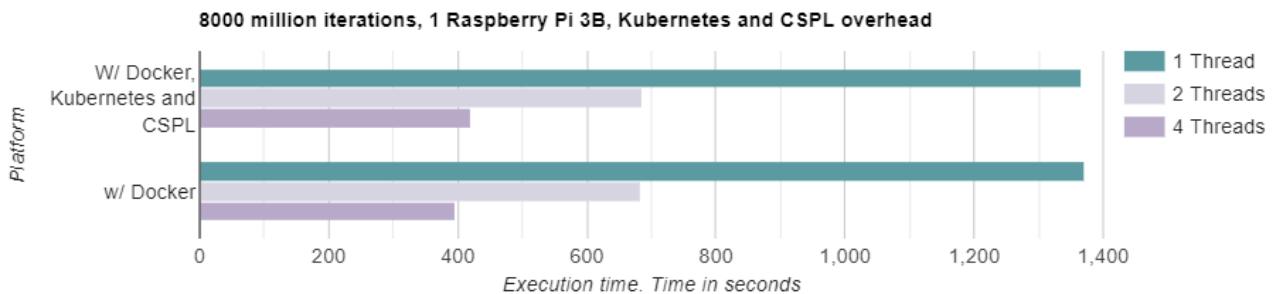


Figure 116: CSPL med Kubernetes overhead.

Overhead beregnes i procent med følgende formel:

$$\left(\frac{Time_{CSPLWithKubernetes}}{Time_{docker}} - 1 \right) \cdot 100 = \%Overhead \quad (29)$$

1 Thread 3B

$$\left(\frac{1365,89s}{1370,7s} - 1 \right) \cdot 100 = -0,35\% \quad (30)$$

2 Thread 3B

$$\left(\frac{684,8s}{683,97s} - 1 \right) \cdot 100 = 0,12\% \quad (31)$$

4 Thread 3B

$$\left(\frac{419,85s}{396,28s} - 1 \right) \cdot 100 = 5,95\% \quad (32)$$

Gennemsnitlig overhead

Ud fra ovenstående test ses et minimalt overhead ved at hoste de forskellige processer i Kubernetes med CSPL. I beregning 33 ses et gennemsnitligt overhead på cirka 1,90% når programmet kører i Kubernetes med CSPL. Procenten er kun beregnet ud fra tre samplings, og vil derfor være en smule upræcis, dog ses der stadig et minimalt overhead.

$$average = \left(\frac{-0,35\% + 0,12\% + 5,95\%}{3} \right) = 1.90\% \quad (33)$$

9.6.4 Opsummering - hosting overhead i CSPL systemtet

Ud fra de ovenstående test blev det fundet et overhead ved at hoste programmer i Docker container på cirka 4,39%. Derudover blev der fundet et ekstra overhead ved yderligere at ligge programmet i Kubernetes med CSPL på cirka 1,90%. Med disse få antal samplings vil der være en lille usikkerhed hvis forsøget skulle gentages, flere målinger ville være bedre. Ved flere målinger vil de forskellige målinger få en normalfordeling [51]. Således kunne der matematisk vises at de værdier for performance og overhead som er blevet målt, er præcise indenfor en afvigelse af $\pm 5\%$ vha. hypotese-testing [52].

9.6.5 Desktop vs. Laptop vs. CSPL

For at finde optimeringer, fejl og mangler mm. til systemet, er det nødvendigt at have nogle konkrete implementeringer af IClusterParallelizer, som tilnærmelsesvis ligner noget som en klient kunne implementere. For at teste om systemet giver bedre performance og i hvilke scenarier, er det nødvendigt at kunne teste med forskellige algoritmer og datasæt. Parametrene i de tests er henholdsvis 'stor algoritme', 'lille algoritme', 'stort datasæt' og 'lille datasæt'. De forskellige tests er implementeringer af algoritmer som er fundet på internettet og tilpasset til IClusterParallelizer. Et fælles kriterie for de konkrete eksempler er, at de kan paralleliseres. Følgende algoritmer vil blive testet på både Desktop, Labtop og CSPL systemet, for at undersøge om CSPL systemet kan udkonkurrerer algoritmerne på Desktop og Laptop, eller hvornår CSPL systemet evt. udkonkurrerer algoritmerne på Desktop og Laptop.

9.6.5.1 MonteCarlo Pi approksimation

Monte Carlo Pi approksimation er en algoritme der ved hjælp af x-antal iterationer, kommer tilnærmelsesvis tættere på den korrekte værdien af pi, ved hver iteration [1]. Algoritmen bliver så lang som man ønsker, ved blot at konfigurere hvor mange iterationer der skal gennemføres. Algoritmen udskiller sig fra billedekomprimering i det, at der ikke er noget datasæt. Her er det parametrene 'lille datasæt' og 'stor algoritme' som der tages udgangspunkt i. Til selve implementeringen i C# er der taget inspiration fra internettet [3]. I denne implementering benyttes Task Parallel Library, men til test af CSPL er algoritmen blevet implementeret til at benyttes CSPL-interfacet.

MonteCarlo - Basis

For at sammenligne de forskellige hardwaremoduler der benyttes, er MonteCarlo-algoritmen blevet kørt med 8000 millioner iterationer på henholdsvis 1, 2, 4, 8 & 16 tråde uden brug af CSPL systemet, for at få et basis for hvor meget kraftigere laptop'en og desktop'en er ift. Raspberry Pi 3B. På figur 117 ses de forskellige eksekveringstider for de forskellige hardwaremoduler, samt med forskellige antal tråde benyttet. Det ses her hvor kraftig både desktop'en og laptop'en er ift. Rasberry Pi 3B. Det ses at Raspberry Pi'en ikke får bedre eksekveringstider efter fire tråde da den kun har fire kerner, hvorimod labtop'en kan gå op til otte tråde pga. at den har fire kerner og fire hyperthreads. Det ses også at desktop'en kan gå op til 16 tråde da den har otte kerner og otte hyperthreads.

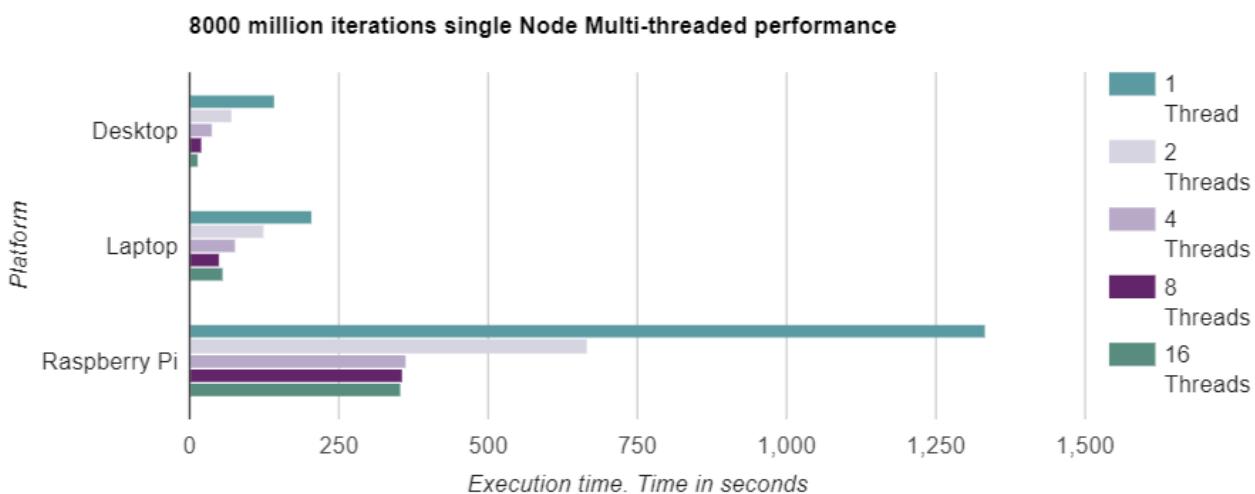


Figure 117: MonteCarlo 1-16 tråde - Desktop vs. Laptop vs. Raspberry Pi 3B.

For at ligge et udgangspunkt for hvor kraftig desktop'en og laptop'en er ift. én Raspberry Pi 3B, beregnes hvor stor en performance faktor forskel der er imellem de forskellige PC'er, dette gøres med udgangspunkt i data fra figur 117, med eksekveringstiden for én tråd. Som det ses i beregning 34 og 35 er desktop'en 9,30 gange hurtigere end Rasberry Pi 3B og labtop'en er 6,46 gange hurtigere end Raspberry Pi 3B.

Desktop vs. Raspberry Pi faktor:

$$\frac{RaspberryPi_{1Thread}}{Desktop_{1Thread}} = \frac{1334,67s}{143,58s} = 9,30 \quad (34)$$

Laptop vs. Raspberry Pi faktor:

$$\frac{RaspberryPi_{1Thread}}{Laptop_{1Thread}} = \frac{1334,67s}{206,73s} = 6,46 \quad (35)$$

MonteCarlo - Desktop vs. Laptop vs. CSPL

For at teste om CSPL systemet udkonkurrerer eller hvornår CSPL systemet udkonkurrerer de andre hardware komponenter, er systemet blevet testet med MonteCarlo algoritmen med 8000 millioner iterationer. Testen presser CSPL systemet og de andre hardware komponenter til det yderste. Som det ses på figur 118 kan CSPL systemet ikke udkonkurrerer den kraftige desktop med 8 kerne + 8 hyperthreads. Selv med 80 workers i CSPL systemet er den kraftige Desktop stadig cirka tre gange hurtigere. Dermed ses det også at CSPL systemet allerede ud konkurrerer laptop'en efter brug af 32 workers.

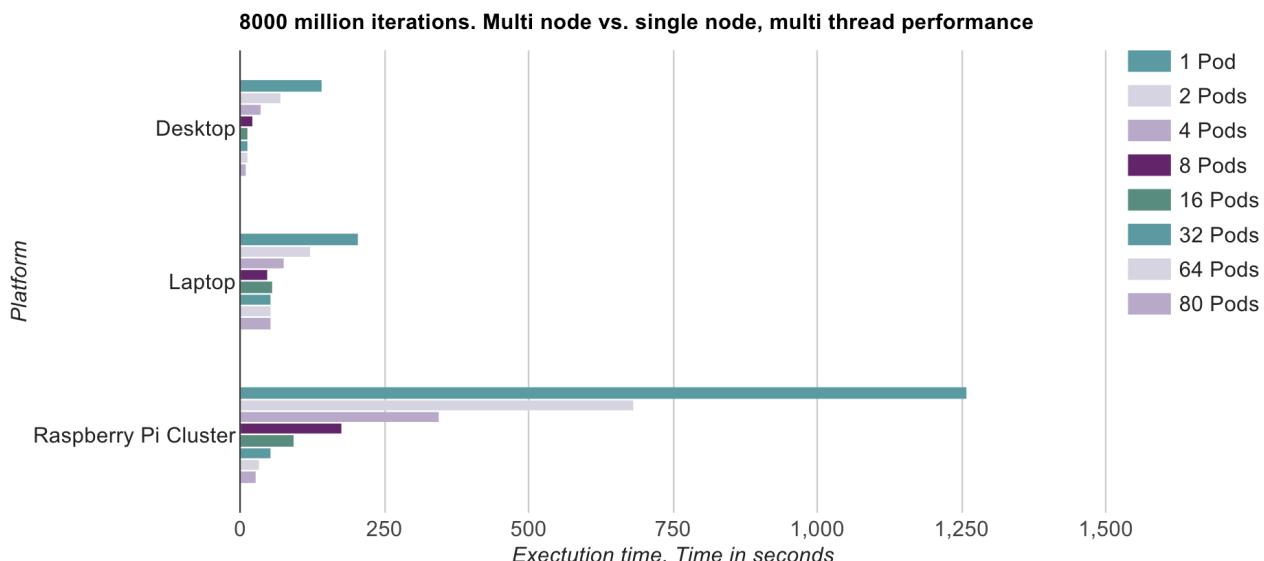


Figure 118: MonteCarlo CSPL 1-80 workers vs. Desktop vs. Laptop

MonteCarlo - lineær gain scaling

Som det ses på figur 119 er MonteCarlo algoritmen blevet kørt med 8 milliarder iterationer vs 100 milliarder iterationer. Derudover ses den teoretiske og ønskede lineær gain-faktor. Den teoretiske bedste skalering er hvor double så mange workers resultere i den havle eksekverings tid.

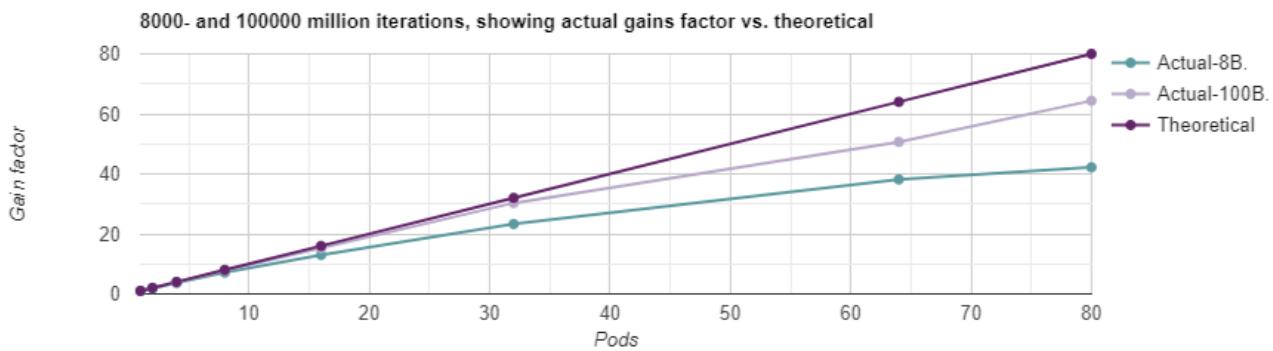


Figure 119: MonteCarlo lineær gain skalering.

MonteCarlo 8 milliader - lineær scalering

For MonteCarlo kørt med 8 milliader iterationer ses det er at det lineær gain følger den ønske teoretiske gain-faktor i starten, men senere afvigeres der mere og mere. Dette skyldes at jobbet for de enkelte workers bliver mindre og mindre, jo flere workers der bruges i algoritmen. Ved omkring 60-80 workers bliver arbejdet så småt at eksekveringen bliver kort, således bliver distribuering/samling af de mange små resultater mere og mere overhead og derfor giver 80 workers kun et gain på en faktor 42 hvor den skulle have været 80 i forhold til den ønskede teoretiske gain-faktor.

MonteCarlo 100 milliader - linear scaling

For at undgå at de forskellige arbejdsopgaver ikke bliver så små, at distribueringen/samling af sub-resultaterne giver overhead ved 80 workers, er 100 milliader iterationer også blivet testet. Her ses det at ved 80 workers er gain-faktoren 64,4 hvor den dertilhørende ønskede teoretiske gain-faktor burde være 80. Det ses i beregning 36 at den faktiske gain-faktor ved 80 workers afviger med 24,22 % procent fra den teoretiske.

$$\left(\frac{\text{PerfectGain}}{\text{ActualGain}} - 1 \right) \cdot 100 = \left(\frac{80}{64,4} - 1 \right) \cdot 100 = 24,22\% \quad (36)$$

9.6.5.2 Brute force SHA256 hash

I forbindelse med performance test af CSPL systemet er der blevet implementeret en SHA256 brute force algoritme, som er inspireret fra Parallel Brute Force, Rosetta Code [5]. Algoritmen kan brute force et hvilket som helst fem-tegns password der indeholder ASCII-værdier fra 48 til 122 ved at fodre algoritmen med den dertilhørende SHA256 hash-værdi. Hver pod eller task får dens eget faste start ASCII-tegn og derefter beregnes hash-værdien hvor hvert fem-tegns kombination med dette unikke start ASCII-tegn. Når algoritmen og alle pods/tasks har været alle kombinationer igennem returneres det fem-tegns password til den givne hash-værdi. Grundet algoritmens natur vil en hvilken som helst fem-tegns kombination give samme eksekveringstid, da algoritmen først returnerer når alle pods/tasks er færdige.

I og med at der tjekkes for ASCII-værdier fra 48 til 122 kan algoritmen paralleliseres på 74 pods eller tasks. Der vil i dette tilfælde være 29.986.576 iterationer pr. pod/task og totalt set 2.219.006.624 iterationer. På figur 120 ses eksekveringstiderne for hhv. laptop, Raspberry Pi cluster og desktop. CSPL systemet udkonkurrerer laptop'en, men det ses tydeligt at desktop'en er langt hurtigere og altså mere effektiv til at beregne SHA256 hash-værdier.

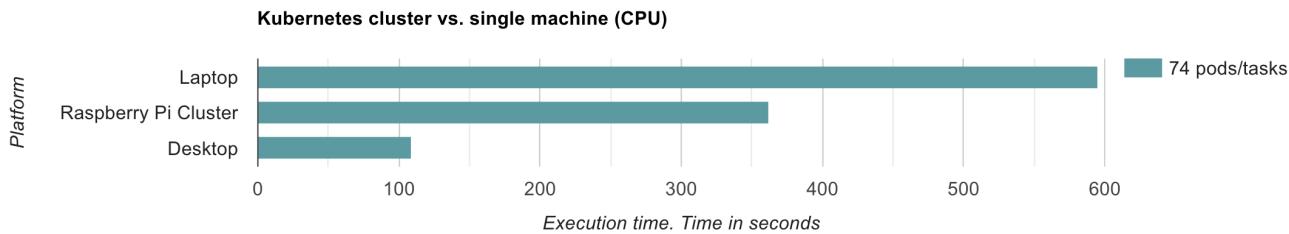


Figure 120: Parallel brute force algoritme der beregner SHA256 hash-værdier for alle kombinationer med 74 forskellige ASCII-værdier (fra 48 til 122). Algoritmen er kørt på hhv. laptop, Raspberry Pi cluster og desktop.

9.6.6 Data sending overhead

Når en udvikler bruger CSPL til at optimere tunge dele af hans program, er det vigtig at tage højde for den data-mængden der sendes, samt hvor lang tid arbejdet tager. Der kan være et stort overhead, hvis man f.eks. sender et stort datasæt og arbejdet ikke tager særlig lang tid. Udviklerens arbejde skal nemlig først sendes til en remote maskine, før det bliver eksekveret, her vil der opstå et naturligt delay. For at få mest ud af det afsatte arbejde, og undgå et for stort overhead, er det vigtigt at udviklerens arbejde tager noget tid, således at delay-tiden bliver mindre markant mindre end eksekveringstiden (CPU-tiden).

Hvor er der delay?

- Initial package request fra klient til master
- Klient result-package fra master til klient
- Sub-packages fra master til x-antal workers
- Sub-results fra x-antal workers til master

Som nævnt i overstående, er der fire led hvori der opstår delay. Som vist på figur 121 ses det visuelt hvor delays opstår. Først og fremmest starter udvikleren hans arbejde, hvilket sender alle hans DLL-dependencies, samt datasæt til master. Dernæst deles pakken op i mindre pakker, som hver især skal sendes til en worker. Hver worker producerer et sub-result, som sendes tilbage til master. Til sidst sender master det endelige resultat til klienten.

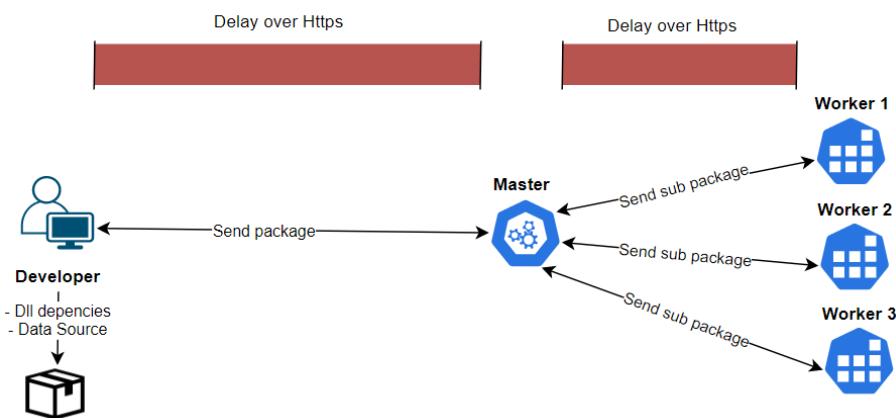


Figure 121: Delay over https

Hvornår er der overhead?

Som det ses på figur 122 er det vigtigt at CPU-tiden er den primære tid i eksekveringen. Hvis udvikleren sender et lille datasæt som har en stor eksekveringstid, vil CPU-tiden fyldes mest i tidsfordeling. Hvis udvikleren derimod sender en stor datapakke som f.eks. 10 millioner ord, og gerne vil finde alle de steder ordet "bord" forekommer, hvilket er en $O(n)$ eksekvering, vil udvikleren få en tidsfordeling hvor HTTPS-delayet fylder meste.



Figure 122: Https delay vs CPU Time

MonteCarlo vs. billedekomprimering

På figur 123 ses to tests som er blevet lavet på CSPL systemet for at finde ud af tidsfordelingen mellem CPU vs. HTTPS-delay. Det ses at når MonteCarlo bliver eksekveret med 8 milliarder iterationer er CPU-tiden høj mens HTTPS-delayet er lavt. Med et stor datasæt er billedekomprimering blevet testet med 35 billeder, her ses det at CPU-tiden er meget lav, imens at HTTPS-delayet er højt. Det er her vigtigt at se overhædet på ImageProcessing arbejdet, det kan godt være at CSPL systemet på x-antal workers kan give 2x gain, men det nyttet ikke noget hvis HTTPS-delayet er 3x gange så stort som CPU tiden.



Figure 123: HTTPS-delay vs CPU-tid - MonteCarlo vs. billedekomprimering. Testet med en klient internet-forbindelse på 30/30 Mbps.

Hvornår kan der være et gain?

Overordnet skal man altid gå efter høj CPU-tid, således HTTPS-delayet ikke bliver et overhead. Hvis udvikleren har et stort dataset skal udviklerens arbejde også gøre op for HTTPS-delayet med noget arbejde der tager en del CPU-tid.

9.6.7 HTTP vs. HTTPS

I systemet er alle forbindelser imellem komponenter konfigureret med en HTTPS, forbindelse således at data er krypteret. Det gælder også for master og worker som egentlig er interne komponenter i clusteret og som snakker over et virtuelt netværk. Som beskrevet i vaglet for HTTPS i afsnit 7.8.1, kan HTTPS nogle gange være hurtige og nogle gange langsommere, grundet forskellige faktorer. Nedenstående testresultater, er lavet på en desktop med en 30/30 forbindelse, og viser næsten ingen forskel. Testen sender 104 MB billeder til master servicen med HTTPS og HTTP. Testen er blevet lavet med 15 forskellige samplings og gennemsnittet er beregnet derudfra.

- **104MB med HTTP:** Gennemsnitligt delay på 13,20s.
- **104MB med HTTPS:** Gennemsnitligt delay på 13,17s.

9.6.8 Billedekomprimering

Der er kørt tests af CSPL sytemet med en billedekomprimerings-algoritme, som modtager billeder og komprimerer dem, ved at mindste kvaliteten og dermed filstørrelsen. Testene har vist at systemet har nogle udfordringer. Billedekomprimerings-algoritmen benyttet store mængder RAM jo flere billeder den skal komprimere

på hver worker-pod, hvilket resulterer i RAM overflow i worker-pods. Dette skete fordi hver Raspberry Pi kun har 1GB RAM hvilket bliver delt mellem fire worker pods, OS, virtuel netværk mm. og af den grund har hver worker-pod et maks RAM cap på 150MB. Hver worker-pod kunne maksimalt processere ét billede hve, hvilket slet ikke var effektivt pga. det overhead der ligger i at benytte CSPL systemet.

9.6.9 MapReduce

CSPL systemet blev testet med en MapReduce-algoritme som skulle finde alle antal af hvert ord i Biblen. Biblen blev opdelt i x-antal delte som blev paralleliseret i clusteret. Her var HTTPS-delayet et overhead, som gjorde at algoritmen ikke havde god performance i systemet. Det tog altså langt længere tid at sende datasættene ift. den tid det tog at processere dataen.

9.6.10 Performance diskussion

Med udgangspunkt i de performancetests som er blevet kørt i CSPL systemet, se afsnit 9.6, har det vist sig at CSPL systemet ved eksekvering af nogle algoritmer er hurtigere end en gennemsnitlig laptop, heriblandt brute force SHA256 hash og MonteCarlo pi estimering.

For nogle algoritmer er der ved at skalere antallet af worker-pods, tilnærmelsesvis en lineær gain-faktor. Dette betyder at clusteret kan skaleres op med flere Raspberry Pi's og workers-pods, og vil dermed kunne udkonkurrere en gennemsnitlig desktop. Dog skal arbejdet på hver worker også øges i takt med at antallet af workers øges, således at der bliver taget højde for det overhead der ligger i at benytte CSPL systemet.

Performancetests, se afsnit 9.6, har også vist at der er algoritmer som ikke performer godt i CSPL systemet. Heriblandt en billedekomprimerings-algoritme, som havde et for stort overhead ved at sende datasæt ift. processeringstiden. Dette var grundet at hver worker-pod maksimalt kunne processere ét billede hver, da flere billeder pr. worker ville forårsage RAM overflow i worker-pod'en, grundet for svag hardware i clusteret og dermed RAM-restriktioner for pods. En anden implementation som ikke performede godt var en MapReduce-algoritme, hvor der var et for stort overhead i at sende datasættet til clusteret ift. den tid det tog at processere dataen.

9.7 Accepttestspezifikation

Følgende afsnit dækker over accepttestspezifikationen. Afsnittet består af en række accepttest som afspejler de user stories som er opstillet i kravsspezifikationen. Måden de forskellige tests er opstillet, er at der er en person ved navn Peter som fungerer som bruger af system, og som har til ansvar at teste de opstillede user stories. Acceptestene er opstillet med Gherkin sproget [7], som er en bestemt måde at bruge sprog, hvor man bruger nogle "keywords" til at give struktur og mening til specifikationer som kan eksekveres. Man bruger ordene fra engelsk "Given" til at definere prækonditioner for en user story, "When" til at definere de specifikke handlinger der skal til for at gennemføre user storien. Til sidst bruges "Then" til at definere de forventede resultater for en given user story. Til sidst i afsnittet vil der være en tabel over resultaterne af de udførte accepttest.

9.7.1 Download NuGet-pakke

Scenarie: Peter kan downloade NuGet-pakken

Given	Peter ikke har downloadet NuGet-pakken
And	Peter har Visual Studio installeret
And	Peter har åbnet NuGet-pakke manageren i "browse" tabben
When	Peter søger på CSPL
And	Klikker på ClusterSupportedParallelLibrary
And	Klikker på install
Then	Peter kan se at NuGet-pakken er installeret

9.7.2 Implementere NuGet-pakkens interface

Scenarie: Peter kan implementere interfacet

Given Peter har downloadet NuGet-pakken til hans projekt
And Peter har en bruger på cspl.dk
And Peter har anskaffet sig en API-nøgle
When Peter får hans klasse til at nedarve fra IClusterParallelizer
Then Peter kan implementere interfacet

9.7.3 Start et job fra Nuget-pakket

Scenarie: Peter kan starte et job via Nuget-pakken

Given Peter har en bruger på cspl.dk
And Peter har en API-nøgle
And Peter har implementeret interfacet
When Peter kalder run metoden på ClusterRunner
And Peter logger ind på cspl.dk
And Peter går ind på "API-key history"-tabben
Then Peter kan se at hans job er igang

9.7.4 Cluster status

Scenarie: Peter kan se status og info omkring antallet af pods i clusteret

Given Peter er nавигeret til cspl.dk
When Peter trykker på "Status of cluster"
Then Peter kan se at cluster status er online og at der kører 80 pods i clusteret

9.7.5 Opret en bruger på hjemmesiden

Scenarie: Peter kan oprette en bruger på cspl.dk

Given Peter er nавигерet til cspl.dk
And Peters email er ikke allerede oprettet i systemet
And Peters password matcher kriterierne for et password
When Peter trykker på "Create account"-tabben
And Peter indtaster peter@gmail.com i email-adresse feltet
And Peter indtaster peter123 i "password" feltet
And Peter indtaster peter123 i "repeat password" feltet
And Peter indtaster Peter i "Display name" feltet
And Peter trykker på "Create account" knappen
Then En grøn besked kommer frem hvor der står at Peters account er oprettet
And Peter bliver nавигeret til "Documentation"-tabben

9.7.6 Login på hjemmesiden

Scenarie: Peter kan logge ind på cspl.dk

Given Peter har allerede oprettet en bruger på cspl.dk
And Peters er logget ind på cspl.dk
When Peter trykker på "Login"
And Peter indtaster sin email i "email" feltet
And Peter indtaster sit password i "password" feltet
Then En grøn besked kommer frem med teksten "You logged in!"
And Peter bliver nавигeret landing page

9.7.7 Log ud af hjemmesiden

Scenarie: Peter kan logge ud af cspl.dk

Given Peter har allerede oprettet en bruger på cspl.dk

And Peters er logget ind på cspl.dk

When Peter trykker på "Logout"

Then Peter bliver navigeret til landing page

And Peters navn forsvinder oppe fra højre hjørne

9.7.8 Ændrer password

Scenarie: Peter kan ændre password på cspl.dk

Given Peter har allerede oprettet en bruger på cspl.dk

And Peters er logget ind på cspl.dk

When Peter trykker på "Manage account"

And Peter indtaster hans nuværende password i "Current password" feltet

And Peter indtaster peter321 i "New password" feltet

And Peter indtaster peter321 i "Retype new password" feltet

And Peter trykker "Change password"

Then En grøn boks kommer op med teksten "You password has changed"

9.7.9 Tilsending af nyt password

Scenarie: Peter kan få tilsendt sit password på email

Given Peter har allerede oprettet en bruger på cspl.dk

And Peters er logget ind på cspl.dk

And Peter har trykket på "Login" knappen

And Peter har indtastet sin email peter@gmail.com i "Email Adress" feltet

When Peter trykker på "Forgot password"

Then En blåboks kommer op med teksten "An email with reset password link was sent to peter@gmail.com"

And Peter har modtaget en email på peter@gmail.com

9.7.10 Slet account på hjemmesiden

Scenarie: Peter kan slette sin bruger på cspl.dk

Given Peter har allerede oprettet en bruger på cspl.dk

And Peters er logget ind på cspl.dk

And Peter er navigeret til "Manage Account"

When Peter trykker på "Delete Account"

And Peter trykker "Agree" i dialog boksen

Then En grøn besked kommer frem med teksten "Your account has been deleted"

And Peter bliver navigeret til landing page

9.7.11 Se dokumentationen

Scenarie: Peter kan se dokumentation for brug af Nuget-pakke

Given Peter er navigeret til cspl.dk

When Peter trykker på "Documentation"-tabben

And Peter trykker på "Implementation"-tabben i venstre side

Then Peter kan se følgende tabs i venstre side

Introduction, Installation, Implementation, How it works, examples

And Peter kan se implementation siden hvor der står information om interfacet

9.7.12 Se krav for brug af NuGet-pakken

Scenarie: Peter kan se kravene for brug af NuGet-pakken

Given Peter er nавигeret til cspl.dk under "Documentation"-tabben

When Peter trykker på "Installation"

Then Peter kan se kravene under "Prerequisites"

And Peter kan trykke på de to links som fører til "Download NuGet-pakke" og "Get API key"

9.7.13 Se eksempler for brug af NuGet-pakken

Scenarie: Peter kan se eksempler for brug af NuGet-pakken

Given Peter er nавигерет til cspl.dk under "Documentation"-tabben

When Peter trykker på "Examples"

Then Peter kan se eksempler for implementeringer af interfacet

9.7.14 Se proof of work

Scenarie: Peter kan proof of work som viser grafer for performance ved brug af systemet

Given Peter er nавигерет til cspl.dk

When Peter trykker på "Proof of work" tabben

Then Peter kan se forskellige tabs som viser information omkring performance i clusteret

9.7.15 Få genereret en API-nøgle

<https://www.overleaf.com/project/5e301dd454e0d8000155d330>

Scenarie: Peter kan få genereret en API-nøgle

Given Peter er logget ind på cspl.dk

And Peter er nавигерет til "Manage Account"

When Peter trykker på "Buy API key"

And Peter trykker "Purchase"

Then En grøn boks kommer op med teksten "You API Key has been purchased."

And Peter kan se hans antal af resterende kald i "Calls left" kassen

9.7.16 Se hvor mange kald der er tilbage på API-nøglen

Scenarie: Peter kan se hvor mange kald der er tilbage på hans API-nøglen

Given Peter er logget ind på cspl.dk

And Peter har fået genereret en API-nøgle

When Peter trykker på "Manage Account"

Then Peter bliver nавигeret til "Account overview"

And Peter kan se hans antal af resterende kald i "Calls left" kassen

9.7.17 Se tabel over jobs som er associeret med specifik API-nøgle

Følgende accepttest dækker over alle "*Information omkring jobs*" user-stories.

Scenarie: Peter kan se hvor hurtigt hans jobs bliver eksekveret på clusteret

Given Peter er logget ind på cspl.dk
 And Peter har en API-nøgle
 And Peter har gennemført sekvensen : "Start et job fra Nuget-pakket"
 When Peter nавигerer til "API key history"
 Then Peter kan se en tabel over alle de kald, som han har lavet med hans nuværende API-nøgle
 And Peter kan se eksekveringstider på hans jobs
 And Peter kan se hans jobs start- og slutdato
 And Peter kan se status på hans jobs
 And Peter kan se labels for hans jobs
 And Peter kan se hvor mange pods/workers hans job bruger på clusteret

9.8 Accepttest

9.8.1 Funktionelle krav

Accepttest for de funktionelle krav til systemet. Opsat i en tabel med scenarie, godkendt/ikke-godkendt samt resultatet hvis resultatet stiger udover det man kunne forvente.

Scenarie	Godkendt	Ikke-godkendt	Kommentar
Download NuGet-pakke	x		Ingen kommentar
implementere NuGet-pakkens interface	x		Ingen kommentar
Start et job med NuGet-pakken	x		Ingen kommentar
Cluster status	x		Ingen kommentar
Opret bruger	x		Ingen kommentar
Log ind	x		Ingen kommentar
Log ud	x		Ingen kommentar
Ændre password		x	Ikke implementeret
Tilsending af nyt password på email	x		Ingen kommentar
Slet account	x		Ingen kommentar
Se dokumentation	x		Ingen kommentar
Se krav for brug af NuGet-pakken	x		Ingen kommentar
Se eksempler for brug af NuGet-pakken	x		Ingen kommentar
Se proof of work	x		Ingen kommentar
Få genereret en API nøgle	x		Ingen kommentar
Se antal kald tilbage på API-nøgle	x		Ingen kommentar
Se tabel over jobs associeret med API-nøgle	x		Ingen kommentar
Data omkring jobs skal kunne ses i realtid på hjemmesiden	x		Ingen kommentar

9.8.2 Ikke-funktionelle krav

Krav	Udførelse af test	Godkendt	Ikke-godkendt	Resultat
Usability				
LED-indikation på Raspberry PI	Manuel testing ved at se på Raspberry Pi Clusteret	x		Ikke implementeret grundet corona. Der var elementer som forvirrede test-personen se afsnit 9.3.
Hjemmesiden skal være intuitiv at bruge	Testet ved hjælp af feedback fra test-person	x		Disse er blevet rettet, men på døvarende tidspunkt fejlede dette punkt.
Biblioteket (NuGet-pakken) skal være intuitiv at bruge ved hjælp fra dokumentationen på hjemmesiden	Testet ved hjælp af feedback fra test-person (Software udvikler). Test-personen brugte dokumentation på hjemmesiden til at implementere NuGet-pakkens interface.	x		Små forvirringer i starten, men da test-personen forstod systemet, fik han implementeret interfacet ved hjælp af dokumentationen..
Reliability				
Hvis en pod crasher (unhandled exception) skal den være tilgængelig i clusteret igen inden for 30 sekunder.	Testet ved at lægge en planlagt fejl ind i et program som sendes til clusteret samtidig med at Kubernetes dashboard blev manuelt monitoreret.	x		Pod'en kom op efter 16 sekunder
Hvis en node dor midt i at procesere noget arbejde, skal de andre noder kunne kompensere		x		Ikke implementeret og dermed ikke testet
Clusteret skal minimum have en uptime på 95% over en uge.		x		Ikke testet
Under opdatering af pods skal der være 100% uptime.	Testes ved at ændre versionsnummer i .yaml fil og apply på henholdsvis master og worker	x		Der er 100% uptime, men ved opdatering af workers er der 60/80 pods som er aktive hele tiden.
Performance				
Ved brug af biblioteket skal brugere kunne se et performance gain ved parallelisering	Dette refereres til performance-test afsnittet 9.6	x	x	Delvist accepteret da nogle algoritmer var hurtigere på clusteret og nogle ikke var.
Clusteret skal bestå af minimum 5 noder.	Kubernetes dashboard tjekkes. Her ses det at der er 20 noder	x		Accepteret
Det skal være muligt at skalere systemet med flere noder	Antal af replica sæt ændres i master og worker .yaml filer.	x	x	Delvist godkendt, da master noden ikke kan men worker nodes godt kan
Supportability				
Hvis en service på en node crasher allokeres fejlen i en ekstern fil		x		Ikke implementeret
Det bør være muligt at kontakte administratorer igennem hjemmesiden		x		Fjernet fra krav pga. ændring i forretningsmodel

10 Resultater

Følgende afsnit giver en fyldestgørende beskrivelse af de resultater som er opnået i dette projekt.

10.1 Viden om cloud computing

I projektets gang er har gruppen tilegnet sig viden omkring cloud computing og parallelisering og brugt det til realiseringen af systemet. Der er udviklet et system hvor en bruger kan eksekvere kode parallelt på multible Raspberry Pi's.

10.2 Raspberry pi Cluster

Der er bygget et fysisk Raspberry Pi cluster som ses på figur 124. Clusteret består på nuværende tidspunkt af 20 Raspberry Pi's som alle er kablet sammen igennem en switch.

Clusteret bliver orkestreret via Kubernetes og på hver Raspberry Pi ligger der 4 pods. Der er arbejdet på op-sætningen af et Raspberry Pi cluster på Kubernetes.

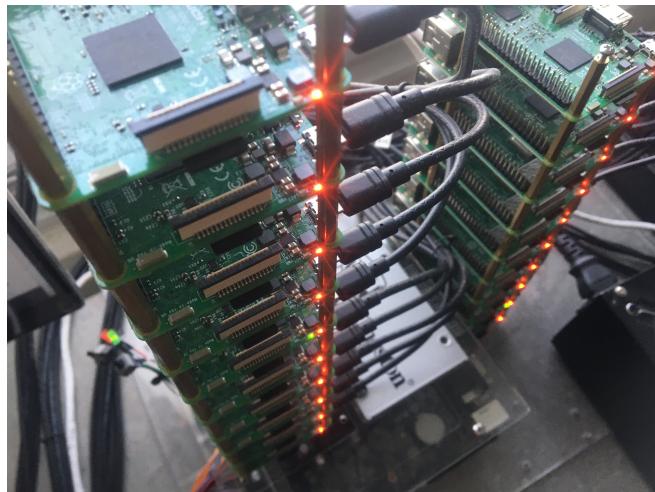


Figure 124: Cluster

10.3 Scrum

Scrum er blevet anvendt som projektstyringsværktøj til udarbejdelse af produktet. I forbindelse med dét, er der anvednt forskellige metoder fra scrum. Heriblandt er daily stand-up, prioritering af opgaver, iterationer i form af sprints mm. Se afsnit 3.5 for en uddybelse af brugen af scrum.

10.4 NuGet-pakke

Der er udviklet en NuGet-pakke som brugere kan hente fra Nuget.org og anvende i sit program. NuGet-pakke inkluderer et interface som brugeren skal implementere for at gøre brug af systemet.

10.5 Kommercialisering

Med henblik på kommercialisering af produktet er der implementeret logik til håndtering af API-Nøgler. Der er implementeret logik til at få genereret en API-nøgle ved køb, håndtering af hvor mange kald der er lavet på clusteret samt mulighed for brugeren for at se brugen af sin API-nøgle.

10.6 Performance

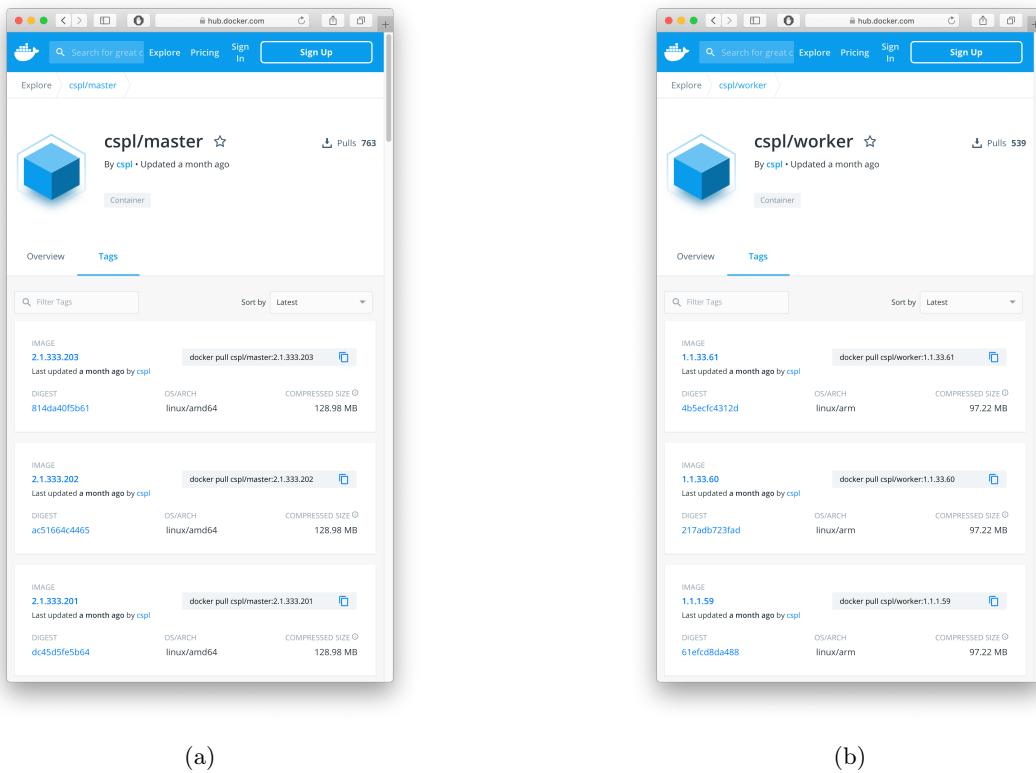
Paralellisering af kode på clusteret har vist sig at give et performance boost visse tilfælde. Der henvises til afsnittet omkring performance i afsnit 9.6.

10.7 Acceptest

Med få afvigelser er accepttesten opfyldt. Acceptesten afspejler de udarbejdede user stories og ud af dem, er det kun "Ændrer password" som ikke er implementeret.

10.8 DockerHub

Der er lavet automatiske byg af docker images og deployment til docker hub, se figur 125.



(a)

(b)

Figure 125: Billeder af Docker images på Docker Hub.

11 Diskussion

11.1 Fejl og fremtidige forbedringer

11.1.1 CPU overlapp

Det kan give problemer at der ikke er nogen restriktioner for, om en klient kan oprette tråde/tasks i IClusterParallelizer.Parrallel, samt at der i pod-konfigurationen på Kubernetes ikke er nogen begrænsninger for, hvilke fysiske kerner der på noden bliver benyttet til at processere tasks. Én pod kan altså benytte alle kerner på en node. En klient kan dermed overlappes/stjæle' CPU-kraft fra andre klienter, hvis de deler en node. Det kan ske ved at klienten opretter mange tråde/tasks i den pod'en der er tildelt og det vil belaste hele noden og dermed de andre pods, som ligger på noden og potentielt set en eller flere pods der benyttes af en anden klient.

11.1.2 Manuel crashing af master

En klient vil kunne crashne master pod'en ved f.eks. at require mere memory end der fysisk er til stede eller som pod'en kan få tildelt. Dette kan f.eks. gøres ved at require memory i et while-loop i IClusterParallelizer.Split.

11.1.3 Manuel slowdown af master

Hvis en klient starter en masse krævende tråde i IClusterParallelizer.Split som aldrig terminerer, vil master-noden blive hængt rigtig meget ud og performance vil blive svækket markant. Dette vil vare ved i seks timer, hvorefter at klienten vil få en timeout exception på HTTPS-kaldet til clusteret.

11.1.4 Ulovigheder/sikkerhed

Generelt kan systemet udnyttes til ulovlige aktiviteter, da der ingen kontrol er af hvad en klient eksekvere, samt tilgår på internettet via clusteret. Der er ingen samtykkeerklæring mellem klienten og CSPL, heller ingen logging af hvad klienten foretager sig. Dette betyder at CSPL (host) hæfter og står til ansvar for al internet-trafik i clusteret.

11.1.5 Datasæt sendes som JSON

Det datasæt som klienter sender til clusteret bliver serialiseres til JSON-tekst. Dette er ikke optimalt da datasættene kan ende med at fyde op til tre gange så meget. Det ville være mere oplagt at sende datesæt som rå byte-arrays og evt. benytte gzip til at komprimere datasæt yderligere [61].

11.1.6 Streaming af klient-data

Alt data der bliver sendt mellem klient og CSPL systemet, samt internt i clusteret, bliver sendt som objekter. I og med at der i .NET er en grænse for hvor meget objekter må fyde, er der altså en teknisk grænse for, hvor meget data der kan sendes på 32-bit systemer på 2GB [34]. Der er flere måder at løse dette på, bla. ved udelukkende at køre 64-bit systemer i clusteret eller ved at sende data frem og tilbage i streams [42]. Ved brug udelukkende af streams på 32-bit systemer, skal grænsen for et objekts størrelse på 2GB stadig håndteres, ved evt. at skrive til filer eller lignende.

11.1.7 DLLPackage på completime

I stedet for at klient-programmet skal pakke DLLPackage (DTO) på runtime, hvilket inkludere rekursivt at finde og derefter load alle klient-programmets DLL'er og dependencies, så kunne dette optimeres ved at gøre det på completime vha. scripts.

11.1.8 DLL Cache

For yderligere at optimere performance kunne der implementeret funktionalitet, som cacher klienters DLL'er og deres dependencies i clusteret/database. Ved at beregne en hash-værdi af de DLL'er en klient sender og map disse, kan en klient slippe for at sende de DLL'er til clusteret, hvis de allerede ligger i cachen.

11.1.9 Stateful master

Arkitektonisk set er det en fejl at master-pod'en holder state, det gør den pga. kø og load balancing. Det gør at det ikke er muligt at starte flere replicas af master-pod's bag master-servicen, hvilket ville være mere optimalt i forhold til rullende opdateringer og skalering. Istedet bør dette data holdes i en database, så multible master API'er kan dele denne state. Hvilket kan gøre alle replicas af master API'et stateless.

Derudover kunne master API'et og de komponenter der ligger deri opbygges omkring en microservice-struktur. Således at de funktionaliteter/komponenter som f.eks. håndtere klientkald, kø og load balancing kunne refakureres ud til at være selvstændige services.

12 Logbog

Dato: 28/01/2020

- Opsætning af scrum værktøjet Jira
- Brainstorming af tasks som lægges i backloggen på Jira
- Installeret af teamcity på Peters server
- Lavet latex dokumenter (rapport & dokumentation)
- Snak med Rasmus omkring akryl plader. Der skal snakkes med Jesper for at få accept.

Dato: 29/01/2020

- Udarbejdelse af samarbejdskontrakt
- Udarbejdelse af foreløbig tidsplan
- Risikoanalyse
- Gennemgik kravsspecifikationen for funktionelle og ikke funktionelle krav (for at nå til enighed)
- Mere brainstorming af tasks til backlog
- udvælgelse af scrum master og beslutningstager

Dato: 30/01/2020

- Bearbejdning af risiko matrix, på baggrund af Jira epics og tasks
- Sprint planlægning
- Tids estimation af sprint og tasks
- Sprint bliver startet og kørers over 2 uger
- Første tasks bliver sat i 'progress' state
- Første tasks bliver reviewet af et gruppe medlem og sat i 'Done' state

Dato: 31/01/2020

- 3x Raspberry Pi bliver hentet i embedded stock

Dato: 03/02/2020

- Brainstorm
- Analyse om assemblies og dependencies

Dato: 04/02/2020

- Vejledermøde
- Analyse om assemblies og dependencies

Dato: 05/02/2020

- Video om Docker og Kubernetes

Dato: 06/02/2020

- Eksekvering af .dll på Docker-image
- Arbejde med Kubernetes POC fortsættes
- Installeret af Linux på master

- Netværk bridge fra wifi til switch lykkes
- Kubeadm installeres og

Dato: 07/02/2020

- Arbejde med Kubernetes POC fortsættes
- persisting af environment variable
- ssh opsætning fra windows
- problemer med ssh på forskellige pc'er på eduroam (på mobil net virker alt)
- reinstall af kubernetes cluster skabte problemer (mangler løsninger)
- Install guide updatet.

Dato: 10/02/2020

- Eksekvering af dll fil sendt fra et program til et andet over http.
- Forsøg på at finde ud af hvilke dependencies en clients program har. Vi fandt en liste ved navn frameworkList_core_v4.5 som vi håbede på var alle de dll'er som ligger i dotnet og prøvede at holde dem op i mod de dependencies der var for clientes assembly. men det mislykkedes da listen vi fandt mangler mange af de standard dll'er der er i dotnet. I morgen prøver vi at lave et post build step som skal

Dato: 11/02/2020

- Vejleder møde. Referat findes i Docs.
- Liste af alle dependencies lavet og virker. ReferencedAssembliesFilter er dermed implementeret.
- Sending af .dll over http, samt load af disse assemblies.

Dato: 12/02/2020

- Kald af metoder fra .dll-filer med parameter af generisk type.
- Refakturering af cluster parallelizer NuGet-pakke.

Dato: 13/02/2020

- Docker build analyse
- Docker images/architecture analyse
- Dockerfile analyse
- Raspbian arm32 vs Ubuntu 18.04 arm64 på PI.

Dato: 14/02/2020

- Opsætning af kubernetes cluster på to Pi's med Ubuntu 18.04.

Dato: 17/02/2020

- Opsumering/gennemgang af POC
- Sprint planlægning
- Sprint goals vurderes således:
 - 1. Rest analyse skrives ned
 - 2. Business model og konkret case
 - 3. User stories
 - 4. Kubernetes
 - 5. Kigge på master

- Nedskrivning af analyse i dokumentations rapporten

Dato: 18/02/2020

- Hele dagen er gået med at skrive ting ned som vi har fundet ud af den sidste uge, dette omhandler: - reflection - dll pakker - docker - kubernetes
- Derudover er der også skrevet noget process, med bla. gruppens brug af scrum og møder.

Dato: 19/02/2020

- Forretningsmodel udarbejdes.
- User stories udarbejdes.

Dato: 20/02/2020

- Opsætning af Kubernetes cluster fortsættes
- Nye opdagelser til tidligere problemer findes i Ubuntu firewall og SELinux (Security Enhanced Linux)
- Overvejelser til hvilket cluster network addon skulle installeres, og hvad Weave works tilbyder kontra andre.

Dato: 21/02/2020

- Opsætning af Kubernetes cluster fortsættes
- Nye problemer findes i dynamiske IP'er og Kubernetes Master node API'et ikke kan tilgåes via den forrige.
- Undersøgelse i DNS-server og om det ville løse IP problemer.

Dato: 24/02/2020

- Der arbejdes på en POC for master programmet, som skal kunne sende dll'er med datasæt remote og så algoritmen esekveret. Vi får lavet et program der kan ovenstående. Næste skridt er at få programmet lagt ud på 2 raspberry pi's og implementere metoden split, som splitter datasættet i x antal chunks og sender det ud til x antal pi's.
- Ny guide følges i forsøget på at få opsat et Kubernetes cluster. Denne guide ender med at give pote, idet worker nodes og master uden problemer formår at joine hindanden. Til dette var én af løsningerne bla. af bruge Flannel i stedet for Weave som det interne router network.

Dato: 25/02/2020

- I forhold til master programmet får vi vi lavet det sådan at : vi implementere IClusterParallizer som client, sender programmet til master som uddeligerer programmet til to pi's med hver deres del af et datasæt, de to pi's kører algoritmen og sender resultatet tilbage til master som samler det.
- Vi stødte ind i et program som giver en fatal error, vi mistænker memory leak. Det kigger vi på imorgen.
- Problemer med at få accesset den deployet Kubernetes pod. Poden som servicere en .NET applikation virkede til at køre, men var ikke mulig at tilgå

Dato: 26/02/2020

- Opsætning af statisk IP på master Ubuntu, resultatet var at internetforbindelsen fra AU-internettet gik tabt.
- Undersøgelse af kubernetesopsætning med DNS.

Dato: 27/02/2020

- Møde med vejleder om statisk routing, der skal skaffes en router og der skal internetforbindelse i LAN-stikket i vores lokale.

Dato: 28/02/2020

- opsætning af build server
- Få pipeline til at bygge en nuget pakke og publishe pakken til private nuget server.

Dato: 02/03/2020

- Set video omkring C4 arkitektur. Det ser godt ud, og vi prøver at køre med det.
- Klargøring af plugin PlnatUml til visual studio code til arkitektur.
- Statiske IP'er på plads vha. router.
- hul igennem til pods/services på kubernetes.

Dato: 03/03/2020

- Start af C4 arkitektur med kontekst diagram og container diagram på plads.
- Indskrivning af arkitektur i dokumentation.
- Hul igennem til pods via master IP.

Dato: 04/03/2020

- Kubernetes installation and setup guide.
- Cluster konfigureret til at være mere responsivt på f.eks. døde nodes.
- Fortsættelse af TeamCity pipeline setup.

Dato: 09/03/2020

- Ordnet pipeline til teamcity
- Dokumentation skrivning om buildpipeline, docker

Dato: 10/03/2020

- worker program lagt ud på pods i clusteret og testet med lokalt kørende master og client program som implementere "pi-approximation". Det virker!!
- dokumentation skrivning om build server.
- Vejleder møde. han var imponeret
- Vi implementerede nuget pakke interfacet og fandt ud af at det var lidt underligt at implementere en algoritme uden datasæt.

Dato: 11/03/2020

- Ændret typeFullName i Nuget pakken til selv at finde ud af typen.
- Kører dotnet core test på build server virker nu. Problem med arkitekturen i en af de programmer som kørte.
- Problem med http client delay. Vi skal lave 1 instans af client istedet for flere. Det går sløvt første gang der laves et request.
- Peter implementere IClusterParallelizer interfacet til at kunne compress images.
- Der rydder op i de forskellige programmer og refactoreres en smule.

Dato: 12/03/2020

- corona dag..

- Vi mødte op på skolen og hentede vores ting. Samlede clusteret hjemme ved peter. Vi arbejde hjemmefra fra nu af.
- Refactorering af master.

Dato: 13/03/2020

- Begynde at kigge på hjemmeside og firebase
- CSPL.common som er en dele-nugetpakke for vores projekter blev lavet.
- API-key validation på master blev lavet.
- Retrospektiv:
Vi bruger story points nu. Sprintet var 50 story points hvilket var for lidt. Næste sprint laver vi 75 story points.

Dato: 16/03/2020

- Refaktorering af worker.
- Unit test af TypeResolver i CSPL.Common
- Hjemmeside påbegyndt
- login, create account er blevet lavet med firebase
- Google credentials drillede med rå json fil. Default med env var blev den endelige løsning.

Dato: 17/03/2020

- arbejde på work delegation via labels
- få worker til at sende request til master API via PATCH dokument.
- Unit test af WorkDelegator påbegyndt

Dato: 18/03/2020

- Vi fandt ud af at workdelegation via labels var en dårlig ide
- Ny ide med custom LoadBalancer påbegyndt
- Unit test af WorkDelegator færdig

Dato: 19/03/2020

- CodeCoverage build server.
- Internal klasser i nugetpakkerne.
- Unit test afsnit i dokumentation startet.
- Unit test af LoadBalancer færdig

Dato: 23/03/2020

- Rapport skriving LoadBalancer
- LoadBalancer skal være thread safe da multiple clients skal bruge den samme cache. Derfor har vi skriftet fra dictionary til concurrent dictionary.

Dato: 24/03/2020

- Rapport skriving LoadBalancer, færdig gjort

Dato: 25/03/2020

- Error handling af exceptions fra clientens kode.

Dato: 26/03/2020

Dato: 01/04/2020

- Påbegynder opgaven om at lave api-key history.
- Får model til backenden og funktionalitet til at ligge api key data op på firebase.
- Refactorering af firebase klasse.

Dato: 02/04/2020

- Lavet tabel til api-key data på hjemmeside.
- Bruger react table som skelet og får lavet den til at håndtere vores data.
- lavet funktionalitet til at subscribe på sin api-key. Går fint når man adder calls men problemer med updates.

Dato: 03/04/2020

- Laver funktionalitet til at slette fra tabellen.
- Forsøg på at fixe en render bug i tabellen. Tabellen skal re-render hver gang nyt data bliver læst fra firebase, men dette driller.

Dato: 06/04/2020

- Kø funktionalitet bliver udviklet.
- Testing af kø samt optimering

Dato: 07/04/2020

- arbejde med headeren på hjemmesiden
- Brug af cluster ssh til opsætning af pi's. Problemet med cluster ssh er når man kommer til at lave rav i den. Så tager det lang tid at rette op på.
- Problemer med opsætning af pi's. Fordi masteren og de første pi's har været installeret med en ældre version af kubernetes kunne de resterende 16 ikke forbindes med de første pi's. Det tog lidt tid at finde ud af men nu er der 20 pi's.

Dato: 08/04/2020

- Vi lavede en stor todo liste med bugs for hjemmesiden som vi tager os af idag og imorgen.
- Refactorering af kø funktionalitet. Ligger nu i en QueueUpdater klasse.

Dato: 09/04/2020

- Arbejde på bugs for hjemmesiden
- Vi finder et problem med en process på master noden som hedder kswapd0. Det er en process som swapper memory ud af RAM og ind i noget der hedder swap-space som er lidt hurtigere en harddrive og langt sommere end RAM. Anyways så æder kswapd0 100% af to ud af fire kerner på master noden. Dette sker af en uvidst årsag og vi prøvede at disable swappen og søgte på nettet, men intet virkede. Til sidst blev vi enige om at draebe processen med `kill -9 "processId"`. Det fjerner processen og frigører vores 2 cpu kerner. problemet er nu at kswapd0 bliver sat igang ved hvert restart af masteren. Så har vi lavet et startup script som fjerner processen.
- ovenstående giver et kæmpe performance boost for hele systemet, da masteren som laver mange ting, nu har meget mere CPU at tage af.
- Vi oplever et andet problem som er, at jobs kan blive sat i running-state efter de er completed. Vi kigger på problemet i morgen.

Dato: 10/04/2020

Dato: 13/04/2020

- Sidste sprint planlægning tog det meste af tiden vi havde om mandagen.

Dato: 14/04/2020

- Bugfixing på hjemmeside og backend (se sprint goals)
- Oprydning af kode på hjemmeside

Dato: 15/04/2020

- Hjemmesiden er nu hostet på firebase under navnet cspl.dk
- Klargøring af tabben "proof of work"

Dato: 16/04/2020

- SSL Certifikater implementeret på både hjemmeside og master-worker.
- domæne er skaffet

Dato: 17/04/2020

- Vi finder ud af at nuget pakken overhovedet ikke er async... det bliver fixet

Dato: 21/04/2020

- Bug fixing på hjemmeside. Problem med ikke at kunne se jobs på api-key-data siden når det er første gang man fornyer sin api-nøgle.
- Når pods dør ved vi det nu.
- fundet ny implementering som skal prøves. En ordfinder som bruger mapReduce

Dato: 22/04/2020

-

Dato: 23/04/2020

- Rapport

Dato: 24/04/2020

- Rapport

Dato: 27/04/2020

- Rapport

13 Litteraturliste

Hjemmesider

- [1] Academo.org. *Estimating Pi using the Monte Carlo Method*. URL: <https://academo.org/demos/estimating-pi-monte-carlo/>.
- [2] Anthum. *HTTP vs HTTPS Test*. URL: <http://www.httpvshttps.com/>.
- [3] Tore Aurstad. *Calculating PI in C# using Monte-Carlo simulation*. URL: <http://toreaurstad.blogspot.com/2015/07/calculating-pi-in-c-using-monte-carlo.html>.
- [4] Simon Brown. *Software arkitektur model C4*. URL: <https://c4model.com>.
- [5] Rosetta Code. *Parallel Brute Force*. URL: https://rosettacode.org/wiki/Parallel_Brute_Force.

- [6] CodeSnip. *Continuous Integration Essentials*. URL: <https://codeship.com/continuous-integration-essentials>.
- [7] Cucumber.io. *Gherkin Reference*. URL: <https://cucumber.io/docs/gherkin/reference/>.
- [8] Google developer. *Firebase Hosting*. URL: <https://firebase.google.com/docs/hosting>.
- [9] Google Developers. *Pricing plans*. URL: <https://firebase.google.com/pricing>.
- [10] Docker. *Docker Engine overview*. URL: <https://docs.docker.com/engine/>.
- [11] Microsoft Docker Hub. *.NET Core (Official images for .NET Core and ASP.NET Core)*. URL: https://hub.docker.com/_/microsoft-dotnet-core.
- [12] Firebase. *Add Firebase SDKs and initialize Firebase*. URL: <https://firebase.google.com/docs/web/setup#add-sdks-initialize>.
- [13] Firebase. *Cloud Firestore*. URL: <https://firebase.google.com/docs/firestore>.
- [14] Firebase. *View changes between snapshots*. URL: https://firebase.google.com/docs/firestore/query-data/listen#view_changes_between_snapshots.
- [15] Raspberry Pi Foundation. *Raspbian*. URL: <https://www.raspberrypi.org/downloads/raspbian/>.
- [16] SSL For Free. *SSL For Free*. URL: <https://www.sslforfree.com/>.
- [17] Hackernoon. *Kubernetes vs Docker Swarm — A Comprehensive Comparison*. URL: <https://hackernoon.com/kubernetes-vs-docker-swarm-a-comprehensive-comparison-73058543771e>.
- [18] <http://www.risikoanalyser.dk/>. *Risikoanalyse*. URL: <http://www.risikoanalyser.dk/>.
- [19] Kubernetes. *Cluster Networking*. URL: <https://kubernetes.io/docs/concepts/cluster-administration/networking/>.
- [20] Kubernetes. *Configure a Pod to Use a ConfigMap*. URL: <https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/>.
- [21] Kubernetes. *kubeadm init*. URL: <https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-init/>.
- [22] Kubernetes. *Kubernetes Features*. URL: <https://kubernetes.io/>.
- [23] Kubernetes. *Labels and Selectors*. URL: <https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>.
- [24] Kubernetes. *Overview of kubeadm*. URL: <https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/>.
- [25] Kubernetes. *Overview of kubectl*. URL: <https://kubernetes.io/docs/reference/kubectl/overview/>.
- [26] Kubernetes. *Restart policy*. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/?fbclid=IwAR116EK1ZpPhyun5qEhp-c79xd5832MfEY_-iTIBsKTqTiBNloJpZIuyLxQ.
- [27] Material-UI. *Sorting & Selecting*. URL: <https://material-ui.com/components/tables/#sorting-and-selecting>.
- [28] Medium. *Kubernetes vs Docker Swarm. Who's the bigger and better?* URL: <https://medium.com/faun/kubernetes-vs-docker-swarm-whos-the-bigger-and-better-53bbe76b9d11>.
- [29] Medium. *Long Polling vs WebSockets vs Server-Sent Events*. URL: <https://medium.com/system-design-blog/long-polling-vs-websockets-vs-server-sent-events-c43ba96df7c1>.
- [30] Microsoft. *AssemblyLoadContext Class*. URL: <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.loader.assemblyloadcontext?view=netcore-3.1>.
- [31] Microsoft. *AssemblyLoadContext Class*. URL: <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.loader.assemblyloadcontext?view=netcore-3.1>.
- [32] Microsoft. *AssemblyLoadContext.Unload Method*. URL: <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.loader.assemblyloadcontext.unload?view=netcore-3.1>.
- [33] Microsoft. *ConcurrentQueue<T> Class*. URL: <https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1?view=netcore-3.1>.
- [34] Microsoft. *<gcAllowVeryLargeObjects> Element*. URL: <https://docs.microsoft.com/en-us/dotnet/framework/configure-apps/file-schema/runtime/gcallowverylargeobjects-element>.

- [35] Microsoft. *How to: Add class diagrams to projects*. URL: <https://docs.microsoft.com/en-us/visualstudio/ide/class-designer/how-to-add-class-diagrams-to-projects?view=vs-2019>.
- [36] Microsoft. *How to use and debug assembly unloadability in .NET Core*. URL: <https://docs.microsoft.com/en-us/dotnet/standard/assembly/unloadability>.
- [37] Microsoft. *Map dependencies with code maps*. URL: <https://docs.microsoft.com/en-us/visualstudio/modeling/map-dependencies-across-your-solutions?view=vs-2019>.
- [38] Microsoft. *.NET Core application publishing overview*. URL: <https://docs.microsoft.com/en-us/dotnet/core/deploying/>.
- [39] Microsoft. *Overview of Azure Service Fabric*. URL: <https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-overview>.
- [40] Microsoft. *Reflection (C#)*. URL: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection>.
- [41] Microsoft. *Use code coverage to determine how much code is being tested*. URL: <https://docs.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested?view=vs-2019>.
- [42] Microsoft. *Using Streams on the Network*. URL: <https://docs.microsoft.com/en-us/dotnet/framework/network-programming/using-streams-on-the-network>.
- [43] Microsoft. *WeakReference Class*. URL: <https://docs.microsoft.com/en-us/dotnet/api/system.weakreference?view=netcore-3.1>.
- [44] Raelene Morey. *How HTTPS Affects Website Performance*. URL: <https://wp-rocket.me/blog/https-affects-website-performance/>.
- [45] npmjs. *node.bcrypt.js*. URL: <https://www.npmjs.com/package/bcrypt>.
- [46] Ordnet. *Proces*. URL: <https://ordnet.dk/ddo/ordbog?query=proces>.
- [47] Stack Overflow. *Understanding 32-bit vs 64-bit in .Net (Section: Why .Net assemblies are not affected)*. URL: <https://stackoverflow.com/questions/50619692/understanding-32-bit-vs-64-bit-in-net>.
- [48] Matthew Palmer. *Port, TargetPort, and NodePort in Kubernetes*. URL: <https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ports-targetport-nodeport-service.html>.
- [49] Paras Patidar. *Add SSL / TLS Certificate or .PEM file to Kubernetes' Pod's trusted root ca store*. URL: <https://medium.com/@paraspatidar/add-ssl-tls-certificate-or-pem-file-to-kubernetes-pod-s-trusted-root-ca-store-7bed5cd683d>.
- [50] Hossein Pishro-Nik. *1.2.2 Set Operations*. URL: https://www.probabilitycourse.com/chapter1/1_2_2_set_operations.php.
- [51] Hossein Pishro-Nik. *Central Limit Theorem*. URL: https://www.probabilitycourse.com/chapter7/7_1_2_central_limit_theorem.php.
- [52] Hossein Pishro-Nik. *Hypotese testing*. URL: https://www.probabilitycourse.com/chapter8/8_4_1_intro.php.
- [53] React. *Higher-Order Components*. URL: <https://reactjs.org/docs/higher-order-components.html>.
- [54] RedHat. *What is orchestration?* URL: <https://www.redhat.com/en/topics/automation/what-is-orchestration>.
- [55] Seralo. *RaspberryPI models comparison*. URL: https://socialcompare.com/en/comparison/raspberrypi-models-comparison?fbclid=IwAR3xE7XpbmcSijxhITtZ2JpAglq_whCQ4f510bGsN7Oo-sE4bQx3EfWkcf0.
- [56] Stackshare. *Docker vs rkt*. URL: <https://stackshare.io/stackups/docker-vs-rkt>.
- [57] Stackshare. *Jenkins vs. TeamCity*. URL: <https://stackshare.io/stackups/jenkins-vs-teamcity>.
- [58] Apurva Thorat. *What is the difference between concurrency and parallelism? (Section: Concurrent programming execution has 2 types)*. URL: <https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism>.
- [59] Christopher Tozzi. *Linux vs. Windows Containers: What's the Difference?* URL: <https://containerjournal.com/features/linux-vs-windows-containers-whats-difference/>.

- [60] Aarhus Universitet. *ITONK - Objektorienteret netværkskommunikation*. URL: <https://studerende.au.dk/studier/fagportaler/diplomingenioer/undervisningdiplom/valgfag-for-diplomingenioerer-katrinebjerg/valgfag-kun-foraar/itonk-objektorienteret-netvaerkskommunikation/>.
- [61] Wikipedia. *gzip*. URL: <https://en.wikipedia.org/wiki/Gzip>.
- [62] Wikipedia. *Hypervisor*. URL: <https://en.wikipedia.org/wiki/Hypervisor>.
- [63] Wikipedia. *Orchestration (computing)*. URL: [https://en.wikipedia.org/wiki/Orchestration_\(computing\)](https://en.wikipedia.org/wiki/Orchestration_(computing)).
- [64] Wikipedia. *PageRank*. URL: <https://en.wikipedia.org/wiki/PageRank>.
- [65] Wikipedia. *Security Enhanced Linux*. URL: https://en.wikipedia.org/wiki/Security-Enhanced_Linux.
- [66] Wikipedia. *Transport Layer Security*. URL: https://en.wikipedia.org/wiki/Transport_Layer_Security.
- [67] XQuartz. *Quick Download*. URL: <https://www.xquartz.org/>.
- [68] Awadesh Kumar Yadav. *Containerization and Container Orchestration*. URL: <https://medium.com/@awadesh22kumar/containerization-and-orchestration-13538e4b8993>.