

F20 I7-BAC-01 Bachelorprojekt, 2020

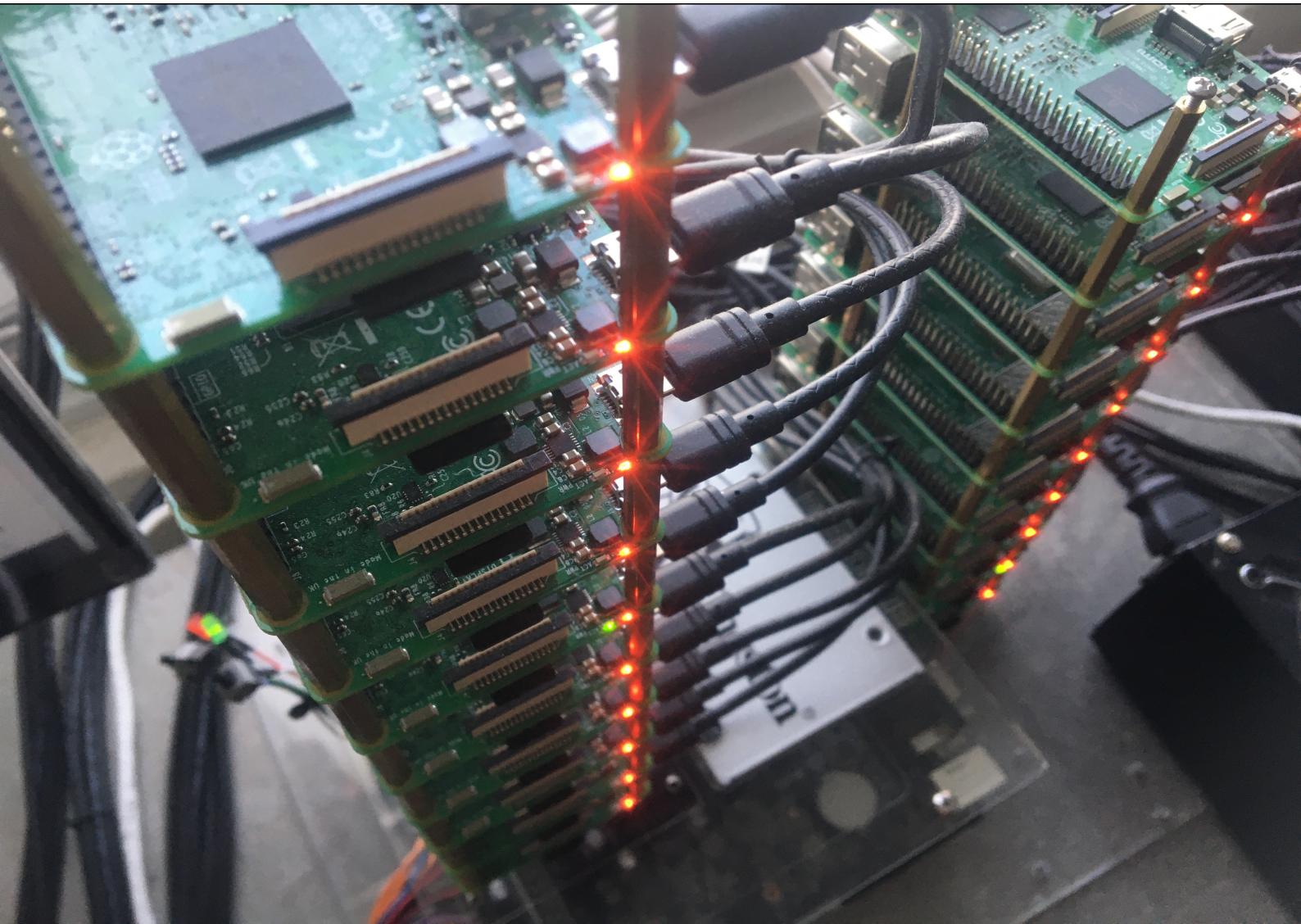
Gruppe 2020F49, Aarhus Universitet

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

Rapport for Cluster Supported Parallel Libraries (CSPL)

Hjemmeside: <https://cspl.dk>

Afleveringsdato: 27-05-2020
Vejleder: Jesper Rosholm Tørresø
Antal tegn: 71.491



Resumé

Formålet med denne rapport er at dokumentere produktet Cluster Supported Parallel Libraries som forkortes CSPL. CSPL er en service som består af en NuGet-pakke og en hjemmeside, bygget op omkring en backend og et 20-nodes Raspberry Pi cluster, orkestreret med Kubernetes. NuGet-pakken tillader klienter at få deres programkode paralleliseret i clusteret. Performance for systemet bliver beskrevet og analyseret i rapporten.

Udviklingsforløbet som gruppen har gennemgået, bliver belyst og dokumenteret i denne rapport. Det omfatter alt fra fastlæggelse af kravsspecifikation, arkitektur og design, til implementering og test af systemet. Der vil også være beskrevet hvordan selve arbejdsprocessen er blevet håndteret, heriblandt arbejdsfordeling, organisering af opgaver, continuous integration og samarbejde mellem gruppens medlemmer.

Abstrakt

The purpose of this report is to document the product Cluster SupportedParallel Libraries, shortend CSPL. CSPL is a service consisting of a NuGet package, a website, build around a backend and a 20-nodes Raspberry Pi cluster, orchestrated using Kubernetes. The NuGet package allows clients to have their program code parallelized on the cluster. Performance of the system is described and analyzed in the report.

The development process for the project is described and highlighted in this report. It includes everything from requirement specification, architecture, design, implementation and test of the system. There will also be a description of how the workflow has been conducted, including the division of labor, organization of tasks and collaboration within the group.

Indholdsfortegnelse

1 Forord	5
1.1 Ordliste	5
2 Indledning & problemformulering	6
2.1 Beskrivelse af projektet	6
2.2 Projektmål	7
2.3 Rigt billede	8
3 Kravspecifikation	8
3.1 Aktør-kontekstdiagram	8
3.2 FURPS	9
3.2.1 Funktionelle krav	9
3.2.2 Ikke-funktionelle krav	9
3.3 User stories	9
4 Afgrænsning	10
5 Metode og proces	10
5.1 Metode	10
5.2 Proces	11
5.2.1 Gruppedannelse	11
5.2.2 Samarbejds aftale	11
5.2.3 Peer programmering og inspektioner	11
5.2.4 Scrum	11
5.2.5 Arbejdsfordeling	11
5.2.6 Tidsestimering	11
5.2.7 Continuous integration	11
5.2.8 Coronavirus tilpasning	12
6 Kommercielt	12
6.1 Forretningsmodel	12
6.2 Funktionaliteter på baggrund af forretningsmodellen	12
7 Analyse	12
7.1 Clusterets hardware	12
7.2 Fjerneksekvering af kode	13
7.2.1 Valg af metode til fjerneksekvering	14
7.3 Orkestreringsværktøj	14
7.3.1 POD skalering	15
7.3.2 Én pod pr. Raspberry Pi	15
7.3.3 Fire pods pr. Rasberry Pi	15
7.4 Distribuering af arbejde til workers	16
7.5 Responsiv data fra clusteret til hjemmesiden	17
8 Arkitektur	17
8.1 C4 level 1 & 2 - Systemets opbygning	18
8.2 Hosting af applikationer	20
8.3 C4 level 3 - komponentdiagrammer	21
8.3.1 NuGet-pakken	21
8.3.2 Master API	22
8.3.3 Worker API	22
8.3.4 CSPL.Common	23
8.3.5 Hjemmesiden	24
8.4 Kubernetes arkitektur	24
8.5 Kommunikation mellem frontend og master	25

8.6 Hardware arkitektur	25
8.7 Port forwarding	26
9 Design	27
9.1 ClusterSupportedParallelLibrary NuGet-pakken	27
9.2 DLL-pakke	29
9.3 Load balancing	29
9.3.1 Custom load balancing	29
9.3.2 Kubernetes API	29
9.3.3 Håndtering af nye og gamle worker-pods	29
9.4 Køstruktur	29
9.4.1 FIFO-kø vs. FLEX-kø	30
9.4.2 FIFO-kø design	30
9.4.3 Klient-status	30
9.4.4 QueueUpdater	31
10 Implementering	31
10.1 ClusterSupportedParallelLibrary NuGet-pakken	31
10.1.1 DLL dependencies og filter	32
10.1.2 ClusterRunner	32
10.1.3 Sikker sending af klient-kode	32
10.2 Master API	33
10.3 Worker API	33
10.4 Fejlhåndtering	33
10.5 Opsætning af Kubernetes	34
10.5.1 Deployment og konfigurering	34
10.6 Forurening af pods i clusteret	34
10.7 Autorisation	34
10.8 API-Key history view	35
11 Test	35
11.1 Unit test	35
11.2 Integration	36
11.3 Manuel test	36
11.3.1 Test med ekstern softwareudvikler	36
11.4 Performancetests	37
11.4.1 Udnyttelse af CPU i systemet	37
11.4.2 Raspberry Pi 3B vs. Raspberry Pi 3B+	38
11.4.3 Docker container overhead vs. traditional deployment	38
11.4.4 CSPL with Kubernetes vs. Docker container overhead	38
11.4.5 Desktop vs. Laptop vs. CSPL	38
11.4.6 Data sending overhead	40
11.4.7 Billedekomprimering	41
11.4.8 MapReduce	41
11.5 Accepttest	41
12 Resultater	42
12.1 Produkt	42
12.2 Performance	44
12.3 Proces	44
13 Diskussion	46
13.1 Produkt	46
13.1.1 CPU overlap	46
13.1.2 Ulovigheder/sikkerhed	46
13.1.3 Stateful master	46
13.1.4 Master svagheder	46
13.2 Performance	47

13.2.1 Overordnet performance	47
13.2.2 Datasæt sendes som JSON	47
13.2.3 Streaming af klient-data	47
13.2.4 DLLPackage på compiletime	47
13.2.5 DLL Cache	48
14 Konklusion	48
14.1 Produkt	48
14.2 Proces	48
15 Fremtidigt arbejde	49
15.1 Forretningsmodel	49
15.2 Cloud-løsning	49
15.3 Systemet som simuleringsværkøj	49
16 Litteraturliste	50
Hjemmesider	50
17 Bilagsoversigt	52

1 Forord

Denne rapport dækker over bachelorprojektet Cluster Supported Parallel Libraries som er udarbejdet af diplomingenørstuderende 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 klientkald 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
- **DTO**: Data Transfer Object
- **ATX PWR**: Port for motherboard power
- **SATA PWR**: Port for powering sata devices
- **Kubectl**: Kubernetes commandline

2 Indledning & problemformulering

Tabel 1 er oversigt over arbejdsfordelingen og kendskab til forskellige dele af projektet. Et stort **X** er udtryk for, at personen har været hovedansvarlig og dermed har stort kendskab. Et lille x er udtryk for, at personen har haft en mindre del i emnet og dermed et mindre kendskab. Hvis der ikke noget kryds, har personen ikke haft del i emnet.

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 1: Arbejdsfordeling for projektet.

2.1 Beskrivelse af projektet

I dette projekt er der udviklet en prototype af et system, som giver softwareudviklere mulighed for, at eksekvere eget kode parallelt vha. cloud computing [28]. Systemet er udviklet med henblik på, at en softwareudvikler kan tilgå ressourcer online og dermed få sin egen kode/algoritme til at blive eksekveret hurtigere.

Cluster

Systemet består bla. af et cluster på 20 Raspberry Pi's (worker-nodes), som skal processere det kode, der kommer fra en bruger/softwareudvikler. Derudover indeholder clusteret en Intel-baseret PC (master-node) som håndterer kommunikationen mellem clusterets workere og brugeren over internettet. Clusteret orkestres vha. Kubernetes, som tilbyder automatisk håndtering og vedligehold af applikationer og ikke mindst skalering deraf [16].

NuGet-pakke

Der er udviklet en NuGet-pakke hvorigennem softwareudviklere, kan få adgang til clusteret, ved at implementere egen kode/algoritme via et generisk interface. Softwareudvikleren kan via NuGet-pakken eksekvere dette kode parallelt i clusteret og få resultat/fejl returneret direkte tilbage i hans/hendes program.

API'er

I systemet bliver der hostet et master API på master-noden, som er udviklet til at modtage det arbejde, som en softwareudvikler ønsker at processere parallelt i clusteret. Derefter uddellegere master API'et "stumper" af dette arbejde til multible instanser af et worker API, som bliver hostet på alle Raspberry Pi's i clusteret. Worker API'ernes funktion er at processere dette arbejde og derefter returnere resultater/fejl til master API, som derfra samles og videredelegeres tilbage til softwareudvikleren. Master API'et er designet til at håndtere flere klienter ad gangen.

.NET Core

Backenden er udviklet i C# vha. ASP.NET Core og .NET Core, da det har været essentielt at supportere cross-platform både ift. de systemer og miljøer softwareudviklere arbejder i, men også pga. muligheden for at hoste systemets API'er på Raspberry Pi's i Linux vha. Kubernetes og Docker.

Hjemmeside

Der er udviklet en hjemmeside, cspl.dk, hvorpå softwareudviklere kan finde information og dokumentation om hvad systemet tilbyder. Derudover er der eksempler på hvordan NuGet-pakken benyttes samt en side med 'Proof of work' som viser klienter, at der kan være en performance-mæssig gevinst ved benytte systemet. Man kan også købe en API-key, der giver brugeren adgang til systemet. Til sidst kan hjemmesiden benyttes til at monitorere status og information om softwareudviklerens egne jobs som kører/har kørt i clusteret.

Proces

Systemet er udviklet vha. Scrum som projektstyringsværktøj. Derudover er der benyttet continuous integration i form af en TeamCity build-server. Build-serverens opgaver har været at køre unittests, bygge Docker images og upload dem til Docker Hub, samt at bygge NuGet-pakker og upload dem til projektets egen NuGet-server.

2.2 Projektmål

- Parallelisere kode på multible Raspberry Pi's.
- Undersøge muligheden for fjerneksekvering af kode vha. cloud computing.
- Bygge et Raspberry Pi-cluster, som orkestreres vha. Kubernetes.
- Udvikle en brugervenlig løsning til cloud computing ift. parallelisering og hurtigere eksekvering af programmer.
- Udvikle en NuGet-pakke i C#, hvorfra softwareudviklere kan implementere eget kode og eksekvere dette i et Raspberry Pi-cluster via et generisk interface.
- Udvikle en grafisk brugergrænseflade i form af en hjemmeside, hvor softwareudviklere kan købe adgang til systemet, søge information om brugen af det, samt monitorere egne jobs som eksekveres i clusteret.
- Undersøge om hvorvidt der kan være en performance-mæssig gevinst, for en softwareudvikler, ved at parallelisere kode i et Raspberry Pi-cluster.
- Udvikle et skalerbart system således at det kan benyttes af multible brugere/softwareudviklere samtidigt, med henblik på kommercialisering af produktet.
- Anvende Scrum som projektstyringsværktøj til udarbejdelse af produktet.
- Anvende continuous integration i udviklingen og produktionen af produktet.

2.3 Rigt billede

På figur 1 er der vist et rigt billede, som giver et overblik over de tanker, der er gjort ift. konceptet.

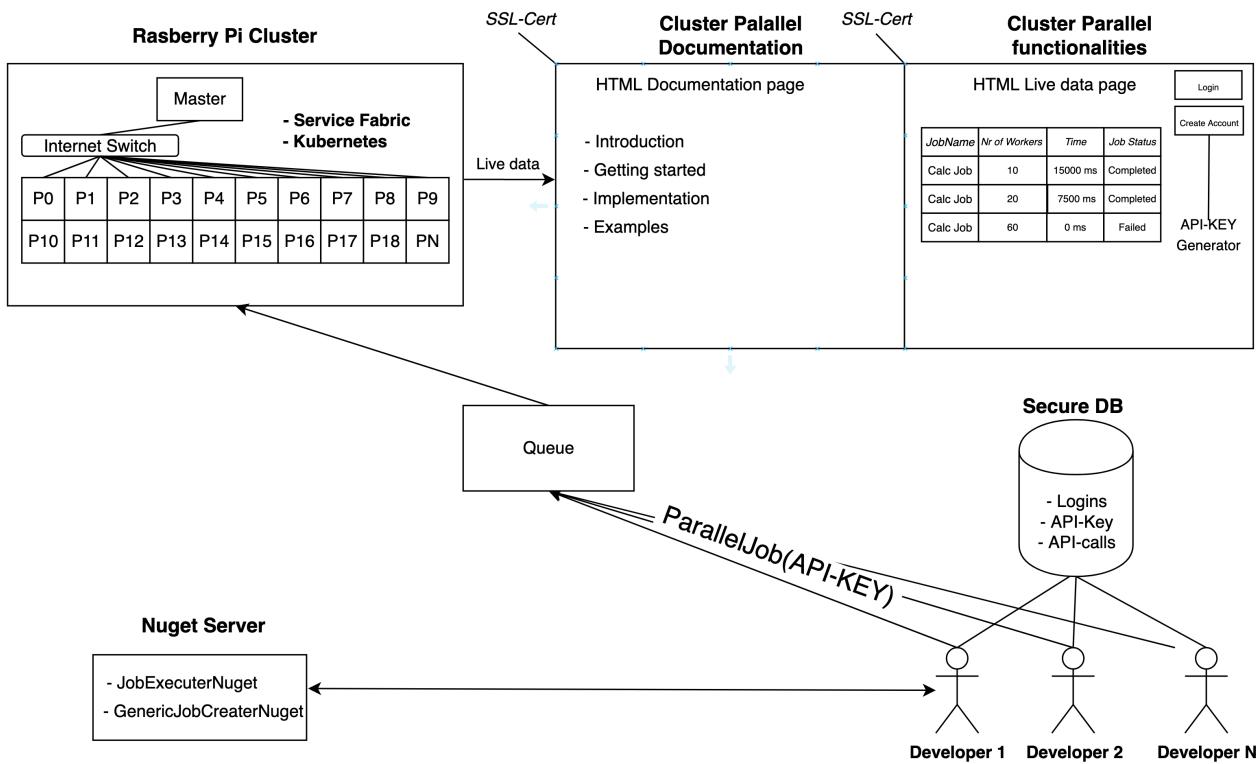


Figure 1: Rigt billede

3 Kravspecifikation

I følgende afsnit gives en opsamling af de opstillede krav til systemet. Dette er gjort ved hjælp af et aktør-kontekst diagram, user stories samt funktionelle og ikke-funktionelle krav.

3.1 Aktør-kontekstdiagram

Aktør-kontekstdiagrammet, som ses på figur 2, viser de primære og sekundære aktører i systemet. Den primære aktør er brugeren, altså softwareudvikleren, som anvender systemets bibliotek eller bruger hjemmesiden til monitorering. De sekundære aktører er henholdsvis hosting af hjemmeside, database og hosting af NuGet-pakken.

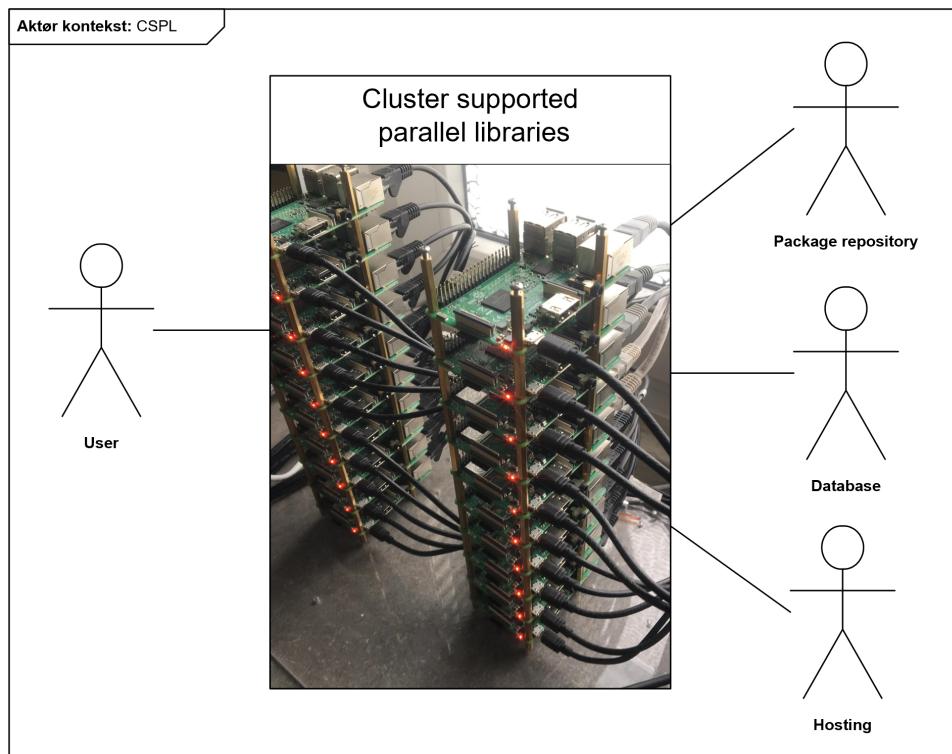


Figure 2: Aktør-kontekstdiagram.

3.2 FURPS

De funktionelle og ikke-funktionelle krav er udarbejdet ud fra FURPS [29]. Se bilag afsnit 2.2 for en uddybelse af FURPS og vægtningen af de forskellige klassificeringer.

3.2.1 Funktionelle krav

Der er ved hjælp af MoSCoW analyse udarbejdet funktionelle krav til systemet. En liste over projektets funktionelle krav kan ses i bilag 2.3.

3.2.2 Ikke-funktionelle krav

Systemets ikke-funktionelle krav dækker over eksempelvis uptime på cluster samt antal af noder og performance. For en yderligere beskrivelse af systemets ikke-funktionelle krav refereres til bilag afsnit 2.4

3.3 User stories

På user stories diagrammet på figur 3, ses de seks user stories som er opstillet. Her ses hvilke user stories de primære og sekundære aktører interagerer med.

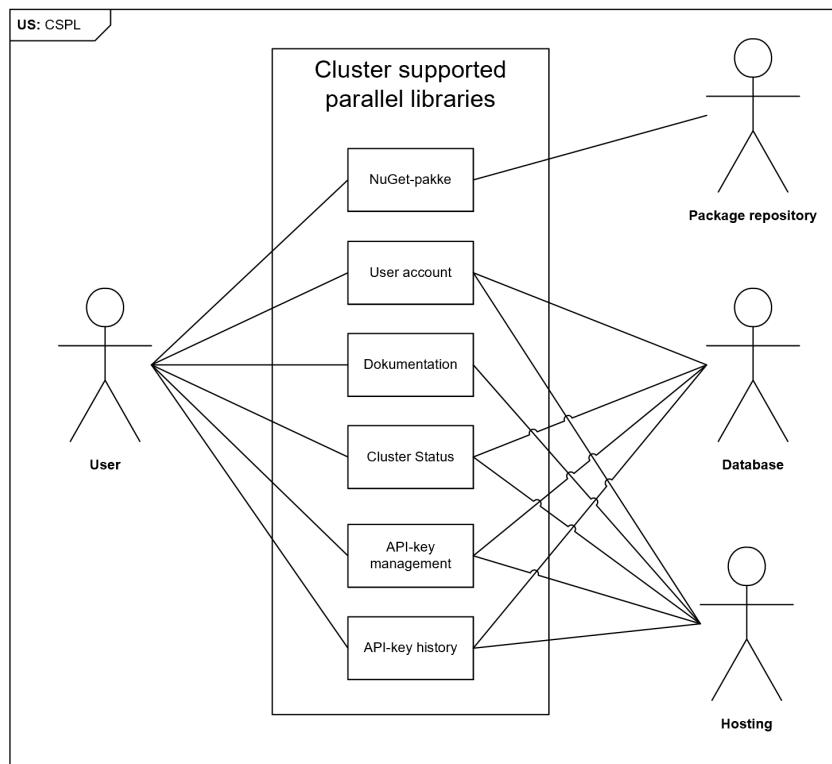


Figure 3: User stories diagram.

I bilag afsnit 2.5 findes en komplet liste af projektets user stories. Listen med user stories er skrevet med følgende syntaks "Som <en type af bruger> ønsker jeg <et ønske i systemet> så jeg <et bestemt formål med ønsket>" [25]. Listen er dannet ud fra de funktionelle krav, som er beskrevet i afsnit 3.2.1.

4 Afgrænsning

Projektet er afgrænset til hardwaremæssigt at bestå af Raspberry Pi's, da dette er en økonomisk overkommelig løsning. Derudover er systemets NuGet-pakke afgrænset til kun at supportere C# og .NET Core. Disse afgrænsninger er lavet med udgangspunkt i MoSCoW analysen i bilag afsnit 2.3.

5 Metode og proces

Følgende afsnit er et resume for procesbeskrivelsen, som er beskrevet i bilag afsnit 3. I afsnittet redegøres der for metode og proces i dette projekt. Derudover er værktøjer, frameworks og programmeringssprog beskrevet i bilag afsnit 3.12. En beskrivelse af gruppens udviklingsforløb kan ses i bilag afsnit 3.6 og alle mødeindkaldelser og referater kan ses i bilag afsnit 3.14.

5.1 Metode

Til prioritering af funktionelle og ikke-funktionelle krav har gruppen benyttet sig af MosCoW-metoden [31].

Gruppen har brugt risikovurdering til at adressere eventuelle risici og usikkerheder i projektet. Dette er gjort ved at udarbejde en risiko-matrix som kan ses i bilag afsnit 3.7. Matrixen danner et overblik over kritiske koncepter og opgaver i projektet. Matrixen har dannet baggrund for prioritering af opgaver, så de opgaver med størst risiko for at vælte projektet, har haft højest prioritet. For at imødegå disse risici har gruppen benyttet sig af POC's (proof of concept)[32]. Her har gruppen lavet små POC applikationer, for at teste de mest kritiske koncepter eller teknologier fra matrixen.

Til udformning og visualisering af projektets arkitektur har gruppen benyttet sig af Simon Browns C4 model. Modellen baserer sig på diagrammer omkring context, containers, components og code [2].

For at give fleksibilitet til projektet har gruppen fra start været enige om, at bruge den agile tilgang til emner som krav, analyse, design, arkitektur og implementering. I den forbindelse har projektet været styret af Scrum.

5.2 Proces

5.2.1 Gruppedannelse

Gruppen på fire mand er en softwareorienteret gruppe, som er dannet på baggrund af et fælles ønske om at samarbejde. Gruppen er vant til at arbejde sammen, og kender med erfaring fra tidligere projekter og opgaver, hinanden styrker og svagheder.

5.2.2 Samarbejds aftale

Der er udarbejdet en samarbejds aftale som kan ses i bilag afsnit 3.3. Den er lavet for at diskutere ting som arbejdstimer, ambitionsniveau og møder. Den har dannet grundlag for at lave en forventningsafstemning. Derudover er det generelt godt for en gruppe at have nogle fælles retningslinjer.

5.2.3 Peer programmering og inspektioner

Under hele forløbet har gruppen gjort meget ud af peer programmering. Dette er gjort fordi at nogle af de udviklede programmer og brugte koncepter har høj teknisk sværhedsgrad. Derudover giver det også automatisk knowledge-sharing og indsigt i det kode, som man ellers havde lavet hver for sig. Gruppen har i stor grad brugt inspektioner til at gennemgå kode, arkitektur og rapport skrivning.

5.2.4 Scrum

Gruppen har tilpasset Scrum på den måde, som har givet det bedste udbytte i projektet. For at bevare fokus på nogle bestemte delmål, har gruppen benyttet sig af sprints på 14 dage. Det betyder at der én gang hver anden uge, bliver holdt sprint-planlægning. Her bliver det kommende sprint-mål diskuteret og fastlagt. Sprintets mål er som udgangspunkt prioriteret ud fra gruppens risiko-matrix. Derudover er der lavet daily-scrum for at opretholde indsigt i hinandens opgaver, samt diskutere eventuelle forhindringer. Der henvises til bilag afsnit 3.5 for en længere forklaring af gruppens brug af Scrum.

5.2.5 Arbejdsfordeling

Arbejdsfordelingen i gruppen har i stor stil været præget af, hvad hvert medlem har haft lyst til at lave. Alle kender deres kompetencer, men nogle gange vil man gerne udfordres i nogle koncepter, som ligger udenfor ens komfortzone, og det har der været plads til.

5.2.6 Tidsestimering

Gruppen har gjort brug af tidsestimering af opgaver med story points. Dette er gjort af to grunde. For det første tegner der sig efter noget tid, et mønster af hvor mange story points man kan nå i et sprint. Dermed bliver det nemmere at scope de kommende sprints. En anden grund, var for at afklare opgavens mål og omfang. Hvis to gruppemedlemmer har haft to forskellige estimeret af en opgave, er det højest sandsynligt fordi, at opgaven er uklar, eller at de to medlemmer er uenige om hvad opgaven indebærer.

5.2.7 Continuous integration

Der er opsat en build-server, som sørger for at bygge nye versioner af NuGet-pakken vha. semantisk versionering, køre tests og deploy Docker images til Docker Hub. Build-serveren er en TeamCity build-server. Valget til dette kan læses i bilag afsnit 4.2.

Build-serveren sørger for alle de fornævnte ting og hjælper derved gruppen med at lave continuous integration. For en mere detaljeret forklaring omkring opsætning af dette refereres der til bilag afsnit 3.11.

5.2.8 Coronavirus tilpasning

Dette projekt har været underlagt nogle specielle omstændigheder pga. coronavirussen, som har ramt Danmark. Det betyder, at gruppen ikke har kunne sidde sammen i gruppelokalet på campus. Generelt har det skabt usikkerhed omkring projektet og eksamen. Gruppen har tilpasset sig situationen og har snakket sammen online over Teamspeak hver dag. Peer-programmering er sket over programmer som bla. Discord og Zoom, som giver mulighed for skærmedeling. Der kan læses mere om gruppens håndtering af coronavirussen i bilag afsnit 3.13.

6 Kommercielt

6.1 Forretningsmodel

For at skabe en retning for produktet, blev der tidligt i projektet besluttet hvem den primære kunde er og med udgangspunkt i det, hvad produktet skulle bestå af. Forretningsmodellen kan ses i bilag afsnit 2.6. Produktet er endt med at være en service, hvor en kunde køber adgang til systemet.

De primære kunder er private softwareudviklere, virksomheder, organisationer eller videnskabelige fakulteter, som ikke har det hardware eller de ressourcer det kræver at skabe et effektivt system til parallelisering/dataprocesseering. Produktet løser dette og kunden behøver ikke at investere i hardware, udviklingen af software eller vedligeholdelse af et sådan system.

Produktet giver også værdi for kunden i og med at produktet er fleksibelt, således at en kunden kun betaler for det arbejde, der bliver eksekveret i clusteret - det er ikke et abonnement.

6.2 Funktionaliteter på baggrund af forretningsmodellen

Følgende features er udviklet udelukkende for at imødekomme forretningsmodellen.

- **API-key**, salg af adgang til systemet
- **Dokumentation**, guide for softwareudviklere til brug af systemet.
- **Proof of work**, for at vise kunden/softwareudvikleren, hvilken performance der kan forventes af systemet.
- **Sikkerhed/SSL**, sikre integritet og fortrolighed af kundens data og source-kode.

For flere informationer omkring brugen om SSL, henvises der til bilag afsnit 7.8.

7 Analyse

I følgende afsnit beskrives de overvejelser og mulige løsninger for forskellige koncepter i projektet, samt valget af disse på baggrund af fordele og ulemper. Der er under analysen udarbejdet 'Proof of concepts' for at verificere teori og koncepter.

Valg af programmeringssprog og frameworks er beskrevet i bilag afsnit 4.7.

7.1 Clusterets hardware

Hardwaren til systemet er opdelt i primær og sekundær hardware. Det primære hardware er det hardware, som har størst indflydelse på systemets performance, se tabel 2. Det sekundære hardware er supporterende og har en mindre indflydelse på systemets performance, se tabel 3.

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 Pen-tium silver J5005	4	1.500 Mhz

Table 2: Primær HW.

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	

Table 3: Sekundær HW.

Master (J5005 mini-ITX)

Grundet en formodning om, at masteren i systemet kommer til at have en tungere arbejdsbyrde end de enkelte workers, er der af denne grund valgt en PC med en stærkere CPU, mere RAM og et bedre netværkskort. Operativ systemet er en UNIX kernel, Ubuntu Desktop. Valget til os'et kan læses i bilag afsnit 4.3.

Worker (Raspberry Pi 3B/3B+)

Worker PC'erne er af serien Raspberry Pi 3B og 3B+. Grunden til dette er, at denne model har en LAN-port og en fire kernet CPU-arkitektur [24]. Med flere kerner kan flere programsekveringer køre parallel, hvilket giver mulighed for bedre performance. Raspberry Pi'en har også dens eget Unix OS, Raspbian, som er designet til at kunne udnytte de tilgængelige ressourcer bedst muligt. En mere detaljeret forklaring på valg af Raspbian som OS, kan læses i bilag afsnit 4.4.

Yderligere hardwarespecifikationer kan ses i bilag afsnit 4.1.

7.2 Fjerneksekvering af kode

Når klientens kode skal eksekveres på clusteret, skal koden sendes til en ekstern maskine for at blive eksekveret. Hertil er der to forskellige muligheder som ses på figur 4 og disse uddybes efterfølgende. En detaljeret analyse findes i bilag afsnit 4.10.

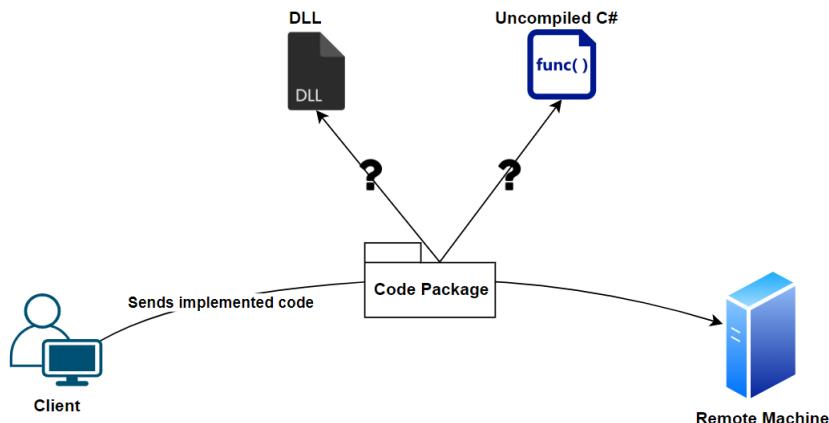


Figure 4: Fjerneksekveringsmuligheder.

Ikke-kompileret C#

Klienten kan sende ikke-kompileret C# kode til den eksterne maskine. Med denne løsning vil det være en stor udfordring at finde frem til de dependencies der måtte være i koden. Derudover har den eksterne maskine brug for at kompilere koden på runtime, før den kan eksekveres [4].

DLL

Klienten kan sende DLL'er til den eksterne maskine. Den eksterne maskine kan via reflection kalde klientens metoder direkte, da koden allerede er kompileret fra klienten side. Se bilag afsnit 4.10.1 for mere information om reflection. Det kræver, at alle DLL-dependencies kommer med over til den eksterne maskine.

7.2.1 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. Det er derfor valgt at lade klienten sende hans kode som kompileret DLL'er.

Kontrakt mellem klient og eksterne maskine

Når den eksterne maskine modtager klientens DLL'er, har den brug for at vide, hvad den skal lede efter i DLL'en. Det er derfor valgt at lave en kontrakt mellem klient og den eksterne maskine i form af et interface. Hvis klienten lover at implementere et interface med specifikke metoder, har den eksterne maskine mulighed for at eksekvere metoderne i interfacet. På figur 5 ses de forskellige trin til fjerneksekvering af kode.

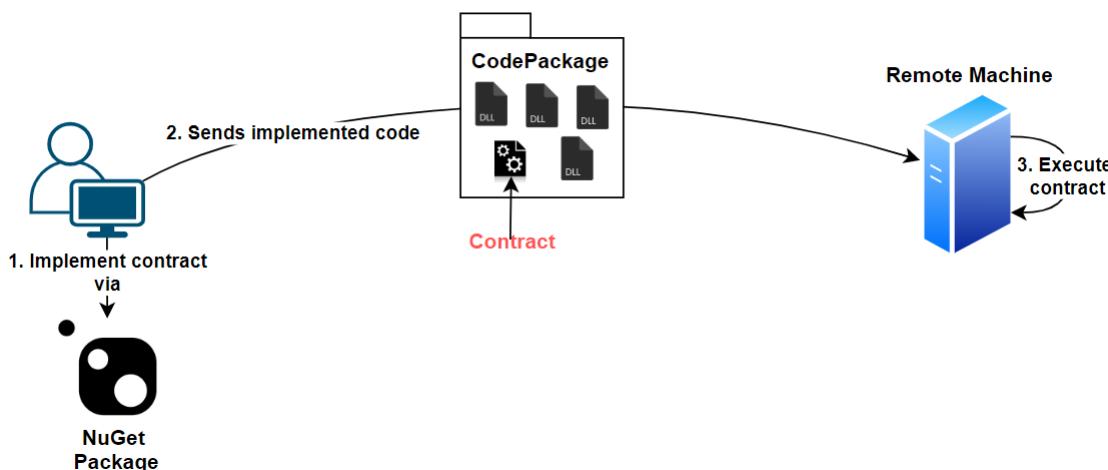


Figure 5: Fjerneksekvering af DLL'er.

7.3 Orkestreringsværktøj

I og med at systemet skal bestå af mange PC-enheder og applikationer, er det oplagt at benytte et orkestreringsværktøj til håndtering af dette, i stedet for manuelt at orkestre systemet. Et orkestreringsværktøj gør det nemt at pakke, administrere og skalere systemer.

Kubernetes bruges som orkestreringsværktøj, i og med at det udstiller features som giver systemet nogle fordele [14], i forhold til alternativer som Docker Swarm og Microsoft Service Fabric, se bilag afsnit 4.6.2 og 4.5. Kubernetes benytter containerized deployment fremfor traditionel og virtualized deployment, se fordele og ulemper for disse typer i bilag afsnit 4.5.1. Derfor bliver denne deployment-type benyttet i systemet, se bilag afsnit 4.5. Derudover bliver gruppens medlemmer undervist i faget ITONK Objektorienteret netværkskommunikation [27], samt at projektgruppens vejlederen Jesper Rosholm Tørresø har teknisk viden indenfor værktøjet.

I og med at Kubernetes benytter container-based deployment [17], er det valgt at systemets API'er skal bygges som Docker images og hostes med Docker Engine, for mere information se bilag afsnit 4.6.1 og 4.6.3.

7.3.1 POD skalering

Da systemets hardware består af 20 Raspberry Pi's som henholdsvis har fire kerner hver, er der to forskellige løsninger hvorpå systemet effektivt kan gøre brug alle kerner. Som det ses på figur 6 er den første mulighed at oprette fire workers-pods pr. Raspberry Pi, således kan hver worker-pod få tildelt én kerne. Den anden mulighed er at sætte en worker-pod pr. Raspberry Pi således at den givne worker-pod har fire kerner.

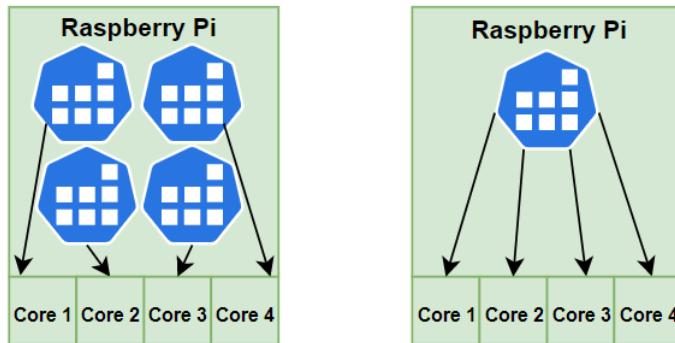


Figure 6: Forskellige node-konfigurationer.

7.3.2 Én pod pr. Raspberry Pi

Med denne løsning vil clusteret bestå af 20 workers-pods. Klientens DLL'er som bliver sendt til en worker, skal være implementeret således at koden kan udnytte de x-antal kerner, der er på en worker-node. Klienten har derfor brug for at vide 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 kerner i clusteret i pakker af fire. En klient vil f.eks. ikke kunne requeste et skævt antal kerner 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 én worker-pod gøre brug af fire kerner er, at en klient kan bruge flere tråde i hans program. Klientens implementeringen kan være mere krævende, men det har også visse fordele bla. at de forskellige tråde kan kommunikere med hinanden.

7.3.3 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 kerne. Ved denne løsningen har klienten ikke brug for at vide hvor mange kerner 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 requeste pakker af fire kerner ad gangen. Klientens implementeringen vil også blive mere simpel, da klienten bare skal dele hans arbejde op i x-antal stykker, uden at tænke selv at oprette tråde.

Endelig valg

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

- Pros:

- Klientens implementering bliver simplificeret da klienten ikke selv skal oprette tråde.
- Klienten skal ikke kende hardwaren i clusteret for at kunne udnytte nodes i clusteret mest optimalt.
- 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 fire altid skal være nul.

- Cons:

- Klienten vil med denne løsning ikke have mulighed for internt at kommunikere imellem tråde.

7.4 Distribuering af arbejde til workers

Når arbejde skal distribueres ud på clusteret, er det vigtigt at alle kerner i clusteret kører parallelt og at hver kerne får tildelt et stykke arbejde. Kubernetes supporterer load balancing via services [15]. For at teste denne load balancing funktionalitet, er der lavet en undersøgelse i form af et "Proof of concept" på distribuering af arbejde i Kubernetes. Undersøgelse kan ses i bilag afsnit 4.11.1. Resultatet af denne undersøgelse viser at fordelingen af arbejde på clusteret fungerer som det er vist på figur 7. Det ses at arbejdet på clusteret bliver fordelt tilfældigt. Dette gør at nogle workers f.eks. får dobbelt så meget arbejde som andre, imens nogle workers intet arbejde får og dermed står i idle. Processingstiden bliver dobbelt så lang, når bare én worker får to stykker arbejde.

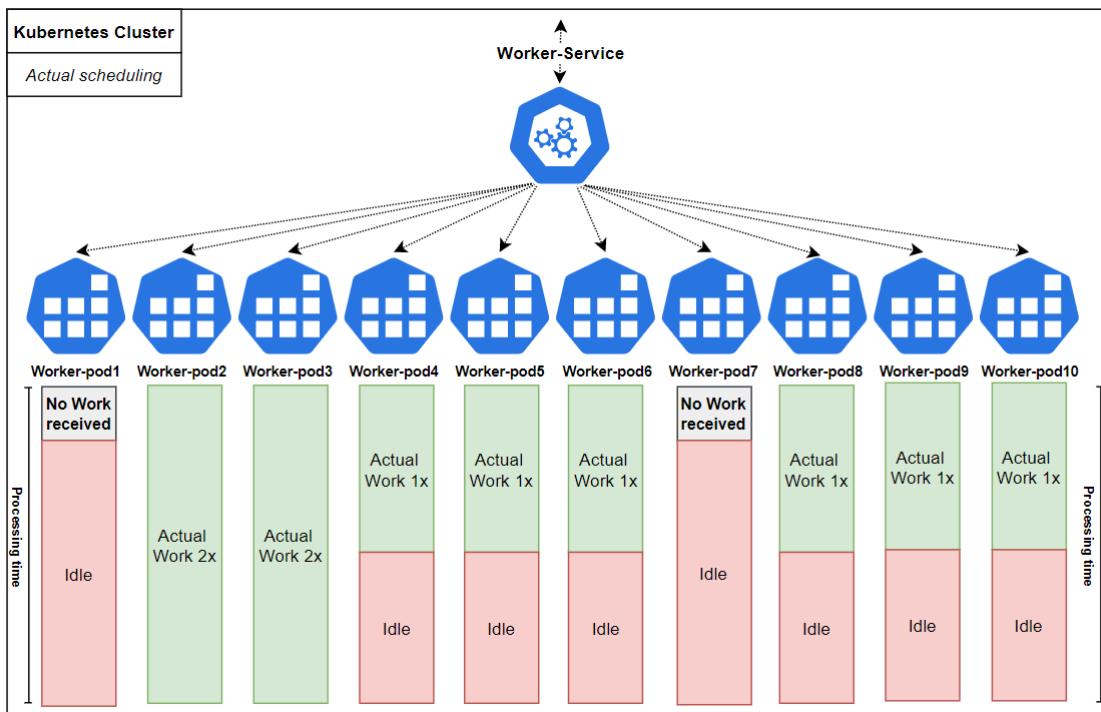


Figure 7: Kubernetes default skedulering.

Når en kerne får mere end ét stykke arbejde, går paralleliseringen fra "true" parallelisme til concurrency. Her context switcher operativsystemet på forskelligt arbejde, da der ikke er nok kerner til rådighed. På figur 8 ses forskellen på "true" parallelisme og concurrent eksekvering. Det ses at med en lige fordeling, kører arbejdet parallelt på hver kerne. Det ses også at med en ulige fordeling, vil operativsystemet lave context switching for at dele kernen.

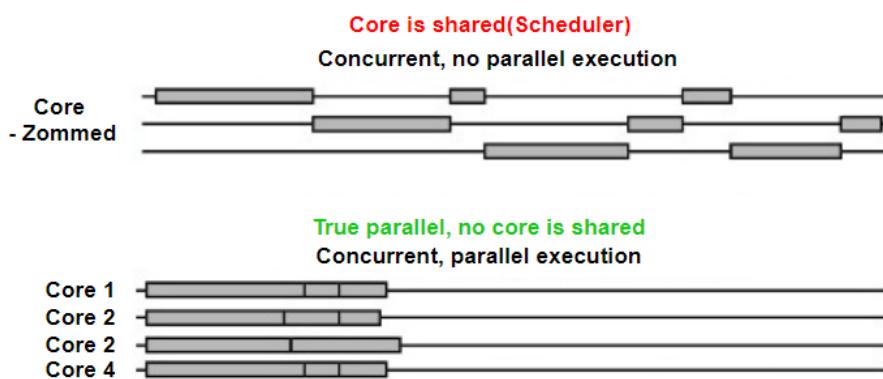


Figure 8: Concurrent vs. true parallel [26].

For at opnå "true" parallelisme og derved opnå den bedste performance, er der designet en custom load balancer. Dette giver den ønskede fordeling og kan ses på figur 9. Designet af custom load balanceren kan ses i afsnit 9.3.

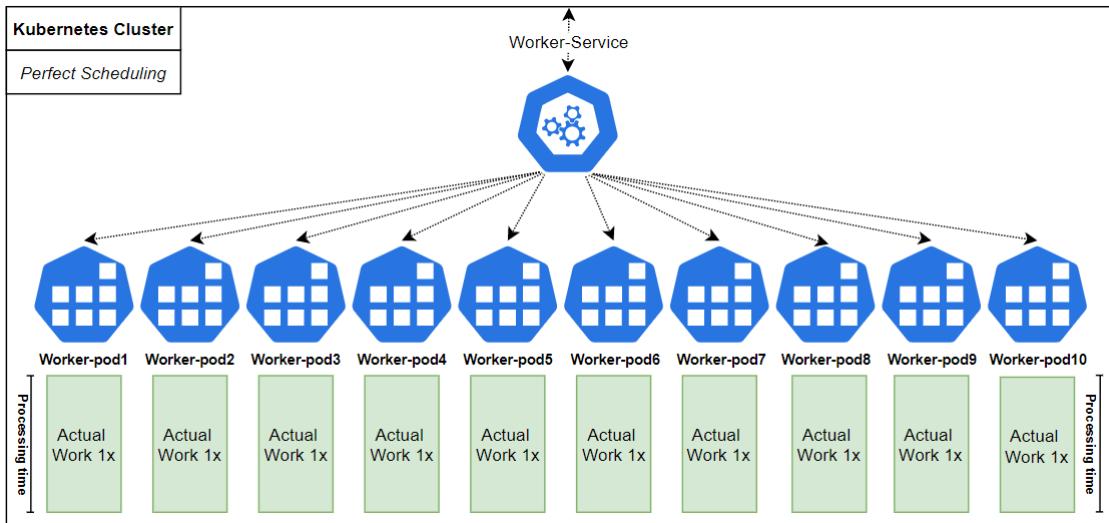


Figure 9: Optimal skedulering.

7.5 Responsiv data fra clusteret til hjemmesiden

For at opnå data i realtid benyttes Google's Cloud Firestore database [7]. Databasen supportere at man kan subscribe til datatabeller og få events med det opdateret data, hver gang data ændrer sig. Der findes forskellige alternativer til at opnå data i realtid, disse kan der læses om i bilag afsnit 4.12.

Der er fordele og ulemper ved at bruge Firebase, frem for selv at lave et API med WebSocket [33].

Fordele ved Firebase:

- Firebase har en masse indbygget features som eksempelvis create account og login, via et simpelt API.
- Der er SSL på alt kommunikation med Firebase.
- Kortere udviklingstid.
- Firebase funktionaliteterne er gennemtestet af Google.

Fordele ved selv at lave et API:

- Det er konfigurerbart
- Der kan implementeres features som Firebase ikke supporterer.
- 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 [6]. En anden pointe er, at dette projekt blot er en prototype og derfor ikke skal kunne håndtere 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.

8 Arkitektur

Følgende afsnit giver en beskrivelse af produktets arkitektur, fra system niveau og ned til komponentniveau. Arkitekturen er dannet på baggrund af C4[2]-modellen. Dette afsnit omhandler C4-modeller fra level 1 til 3, for level 4 modeller henvises der til bilag afsnit 5.5.

8.1 C4 level 1 & 2 - Systemets opbygning

På figur 10 ses level 1 af C4 modellen. Dette diagram kaldes et context diagram og viser systemets primære og sekundære aktører. Her ses det at systemet gør brug af henholdsvis en NuGet-server til hosting af NuGet-pakker, samt Googles Firebase til hosting af hjemmeside og database. Derudover kan det udledes at en bruger i form af en developer, gør brug af systemet.

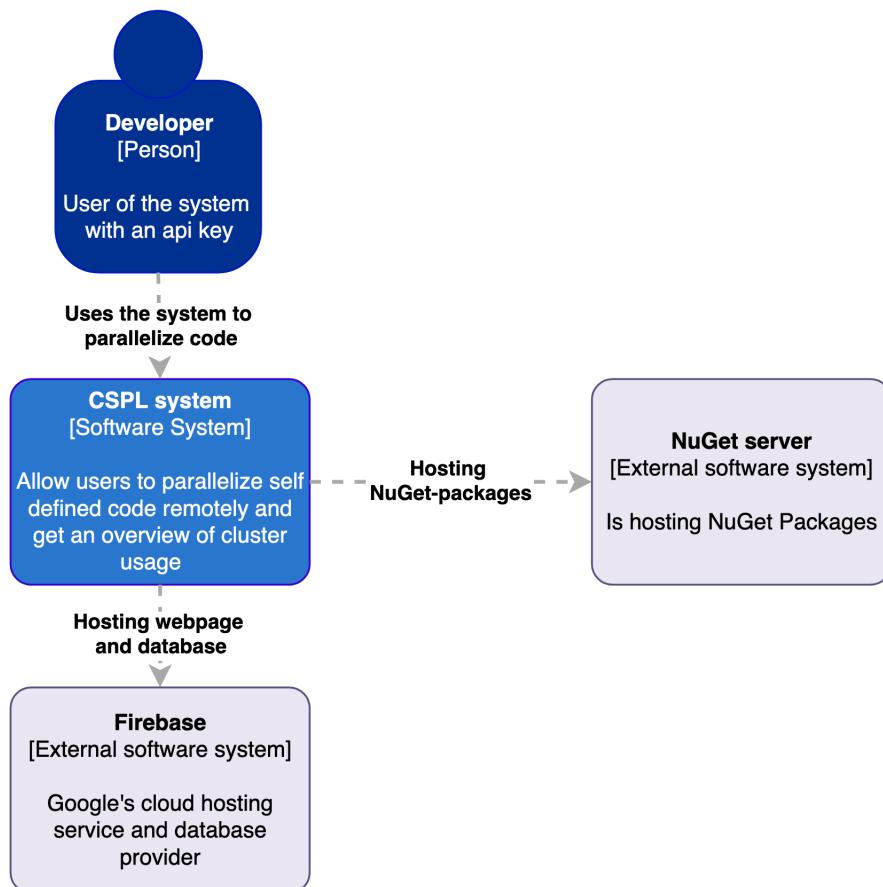


Figure 10: Context diagram, C4 level 1.

System sekvensdiagrammet på figur 11 viser interaktionerne imellem den primære aktør, de sekundære aktører og selve systemet. Det viser i overordnet form det flow som sker fra brugeren downloader NuGet-pakken og bruger den op i mod systemet. Det er ikke alle udfald som er medtaget, da diagrammet blot har som hensigt at vise en simplificeret form af flowet.

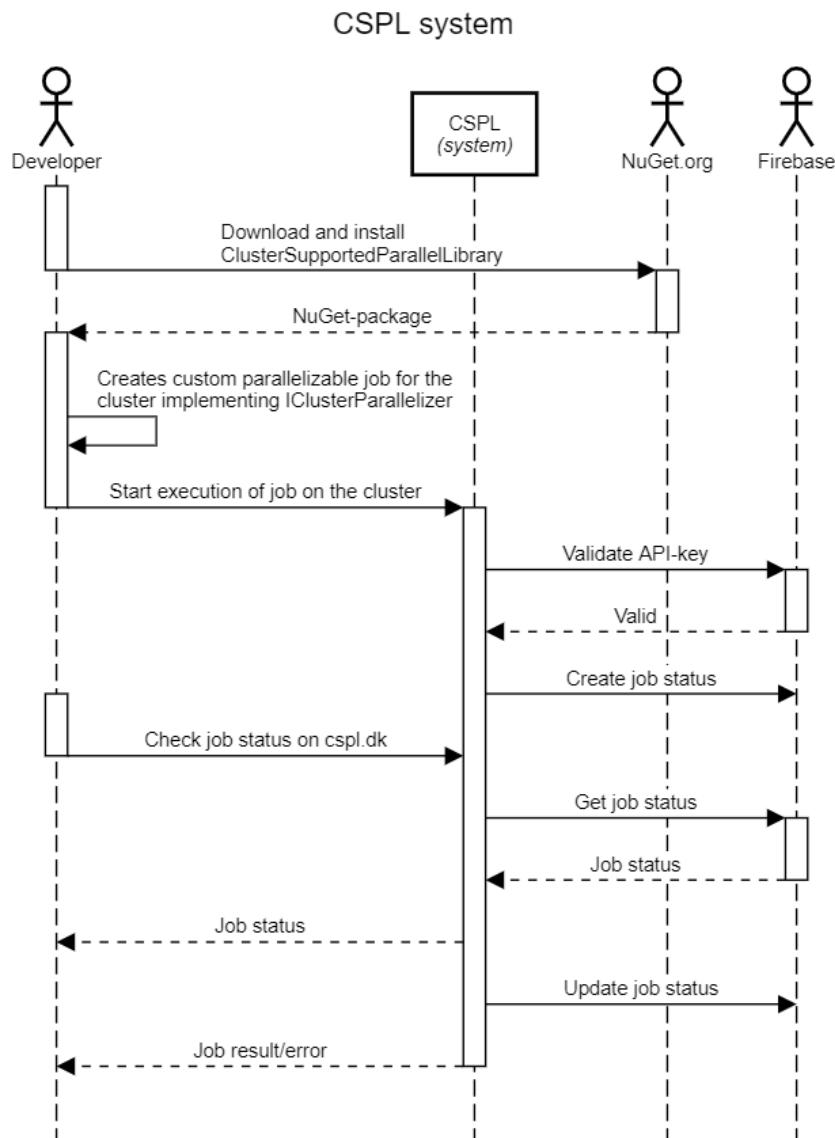


Figure 11: Systemsekvensdiagram for det samlede CSPL system.

På figur 12 ses det interne system. Dette er et container diagram som udgør level 2 af C4 modellen. Det ses at systemet består af en single page application (SPA), to API'er og to NuGet-pakker, som hver i sær udgør komponenter i systemet. Derudover kan det udledes dels hvor komponenterne er hosted og hvordan de interagerer med hinanden. For en mere fyldestgørende gennemgang af systemet komponenter se bilag afsnit 5.5.

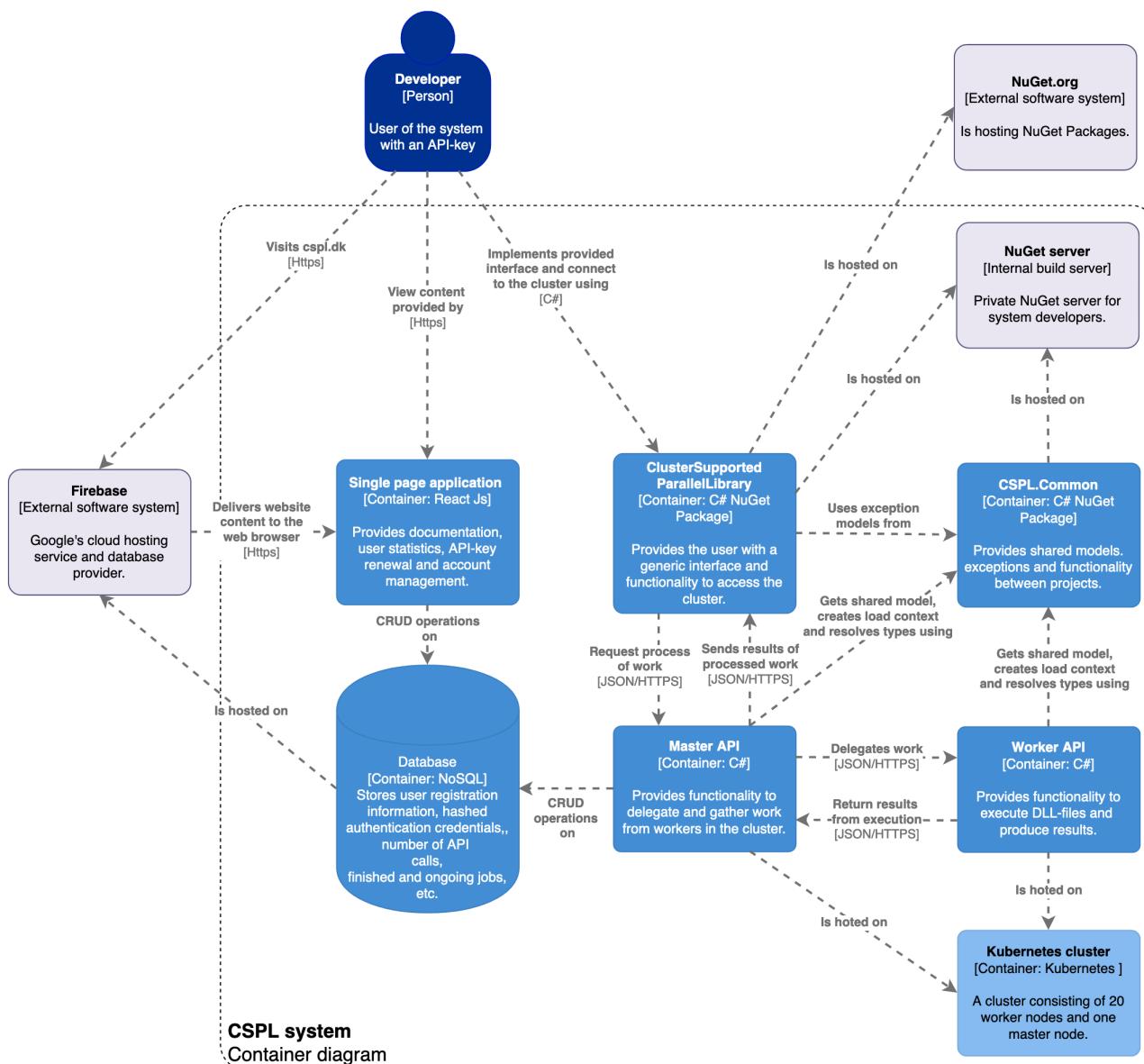


Figure 12: Container diagram, C4 level 2.

8.2 Hosting af applikationer

I følgende afsnit gives en kort beskrivelse af, hvilke arkitektoniske valg der er taget i forbindelse med hosting af systemets komponenter med udgangspunkt i figur 12. En mere dybdegående forklaring kan læses i bilag afsnit 5.6.

Hjemmeside

Hjemmesiden er hosted på Google Firebase [5]. Dette er gjort således at klienter kan se clusterets status, selv hvis clusteret er nede.

NuGet-pakken

Systemets består af to NuGet-pakker. CSPL.Common og ClusterSupportedParallelLibrary NuGet-pakkerne hostes på en privat NuGet-server til brug for systemets udviklere. ClusterSupportedParallelLibrary bliver publiceret til NuGet.org og der har brugere adgang til den.

Master API

Master API'et er hostet i clusteret. Masterens load balancing gør brug af workers virtuelle IP-adresser til at skedulere og uddeletere arbejde. Er masteren ikke hostet i clusteret, kan de virtuelle IP'er ikke tilgås og arbejde kan dertil ikke uddelegeres.

Worker API

Worker API'erne er hostet i clusteret. Det er for at kunne bruge de features som Kubernetes har, bla. skaling af antal replicas og restart af crashed pods.

8.3 C4 level 3 - komponentdiagrammer

8.3.1 NuGet-pakken

Komponentdiagrammet for NuGet-pakken ClusterSupportedParallelLibrary på figur 13 viser de komponenter den består af. ClusterSupportedParallelLibrary er den NuGet-pakke som systemets brugere benytter sig af for at tilgå clusteret samt til at implementere den kode clusteret skal eksekvere. Komponenten kommunikerer til master API'et ved brug af DTO'er og derudover benyttes der delt kode fra CSPL.Common til bla. håndtering af assemblies, type resolving og exceptions.

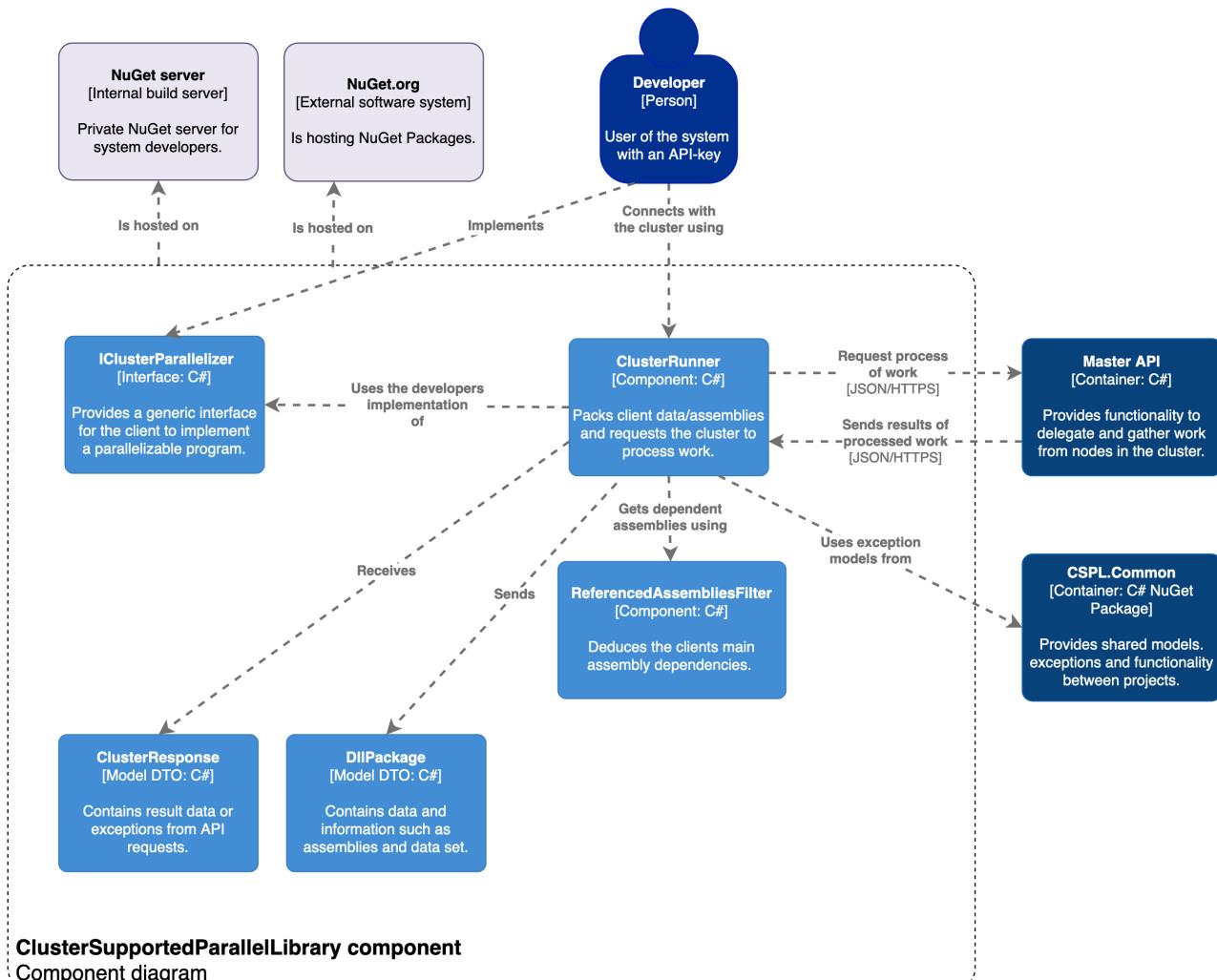


Figure 13: ClusterSupportedParallelLibrary komponentdiagram, C4 level 3.

8.3.2 Master API

På figur 14 ses komponent-diagrammet for master API'et. Master API'et er det program som 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.

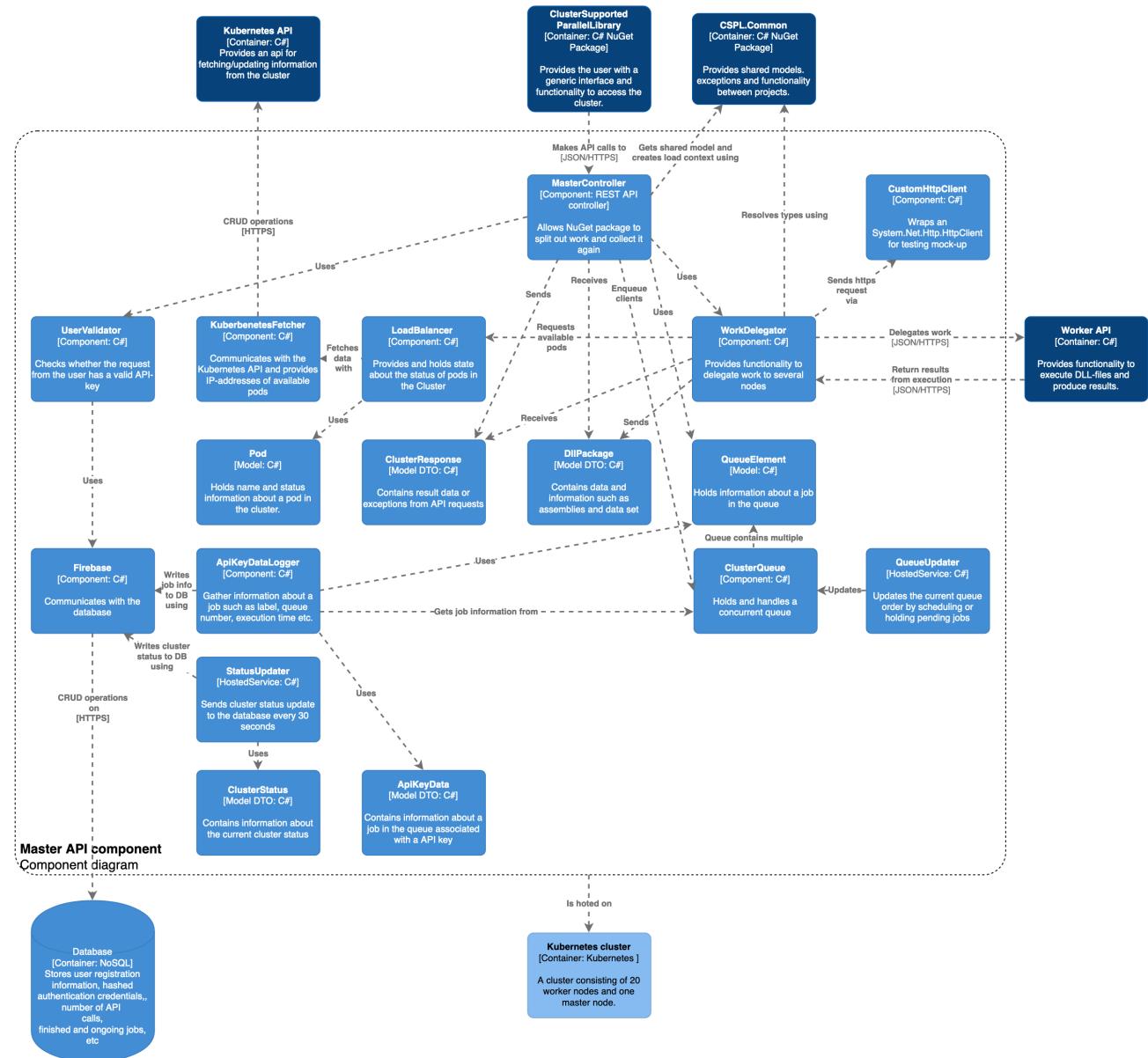


Figure 14: Master API komponentdiagram, C4 level 3.

8.3.3 Worker API

På figur 15 ses komponent-diagrammet for worker API'et. Det er det program som hostes ude på worker-noderne i clusteret. Det indeholder en WorkerController som er den klasse der modtager API-kald fra master API'et. WorkerController'en benytter nogle DTO'er til kommunikation til master API'et, derudover bruges funktionlilitet fra CSPL.Common til bla. loading af assemblies og fejlhåndtering.

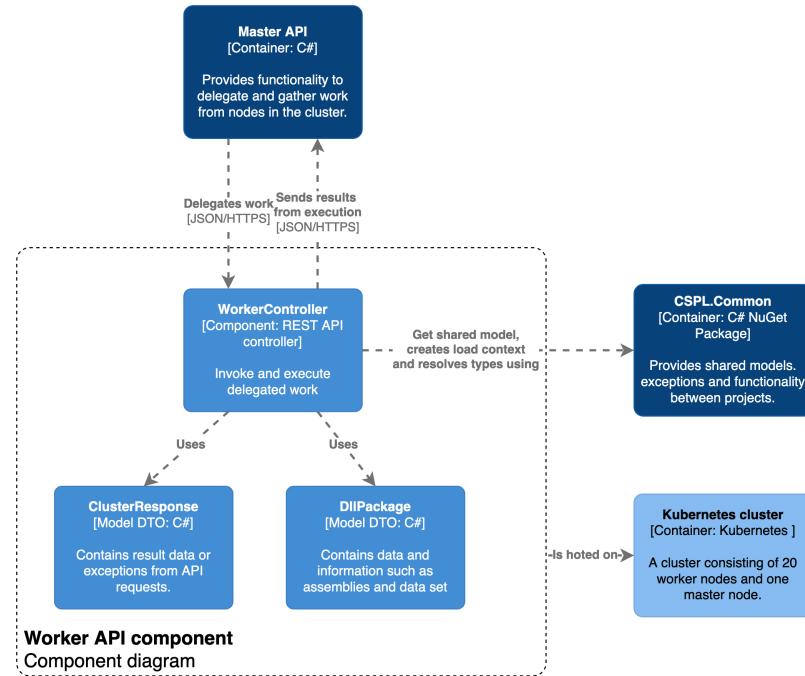


Figure 15: Worker API komponentdiagram, C4 level 3.

8.3.4 CSPL.Common

På figur 16 ses komponentdiagrammet for CSPL.Common. CSPL.Common er en NuGet-pakke som indeholder delte modeller og funktionalitet mellem containere. Komponenten benyttes bla. til at beregne typer på runtime og loadning af assemblies. En forklaring på hvorfor CSPL.Common blev lavet, kan læses i bilag afsnit 7.2.

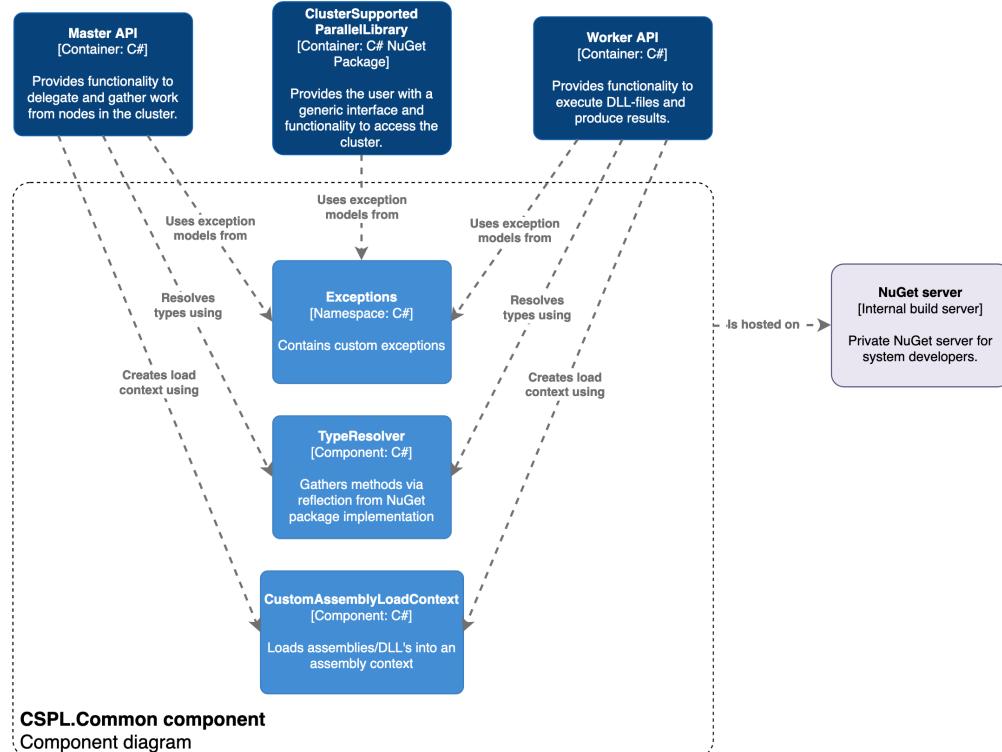


Figure 16: CSPL.Common komponentdiagram, C4 level 3.

8.3.5 Hjemmesiden

På figur 17 ses et komponentdiagram for hjemmesiden. Hjemmesiden bruger Googles Firebase database "Cloud Firestore" til persistering af data. For en gennemgang af hjemmesidens komponenter, se bilag afsnit 5.5.5.

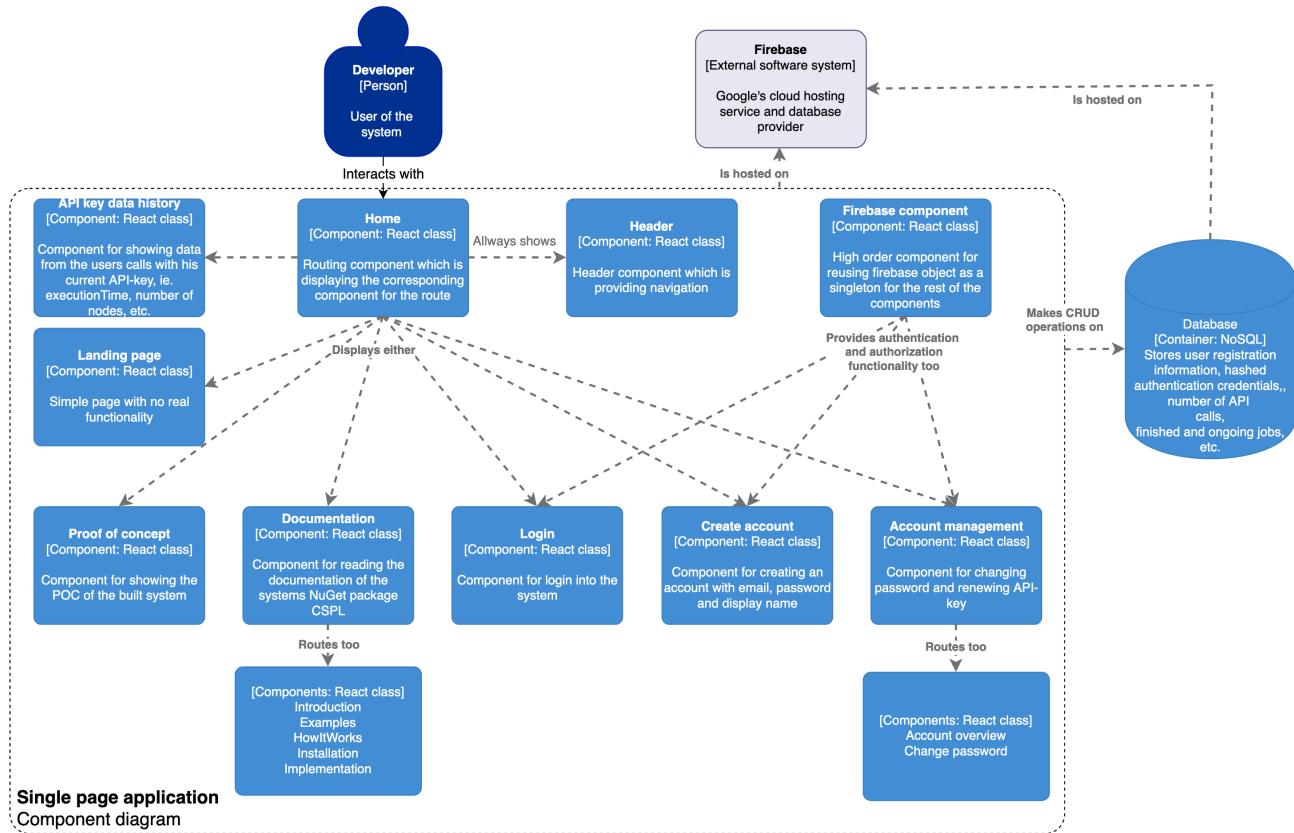


Figure 17: Hjemmeside komponentdiagram, C4 level 3.

8.4 Kubernetes arkitektur

På figur 18 ses arkitekturen for Kubernetes clusteret. Hver Raspberry Pi har fire worker pods. Der kommunikeres over et internt virtuelt overlay netværk. Det virtuelle netværk kan kun tilgås internt i Kubernetes, hvilket gør at det ikke er muligt for udefrakommende at kommunikere direkte til specifikke workers. Alt kommunikation imellem komponenterne og Kubernetes API'et sker over kube-proxy [13], som ligger på hver node. Masteren hoster et dashboard hvor alt Kubernetes data kan overvåges og administreres. Som virtuelt overlay network er der benyttet Flannel, valget for dette kan ses i bilag afsnit 7.5.2.

Når en klient requester clusteret gennem NuGet-pakken, sendes klientens request via HTTPS til cluster.cspl.dk. Domæne navnet cluster.cspl.dk oversættes til en statisk IP via en DNS server, som er indgangen til clusteret. Den statiske IP peger på en Kubernetes service, som fungerer som en reverseproxy. Det betyder at den videredirigere requestet på det interne virtuelle netværk til master-pod'en. Derved skjules det hvilken maskine klienten faktisk snakker med.

Master API'et snakker med Kubernetes API'et, for løbende at opdatere hvilke workers der er klar til at arbejde, samt for at få deres virtuelle IP'er på det virtuelle netværk. Se bilag afsnit 6.1.2 for mere information om hvordan Kubernetes API'et benyttes til custom load balancing.

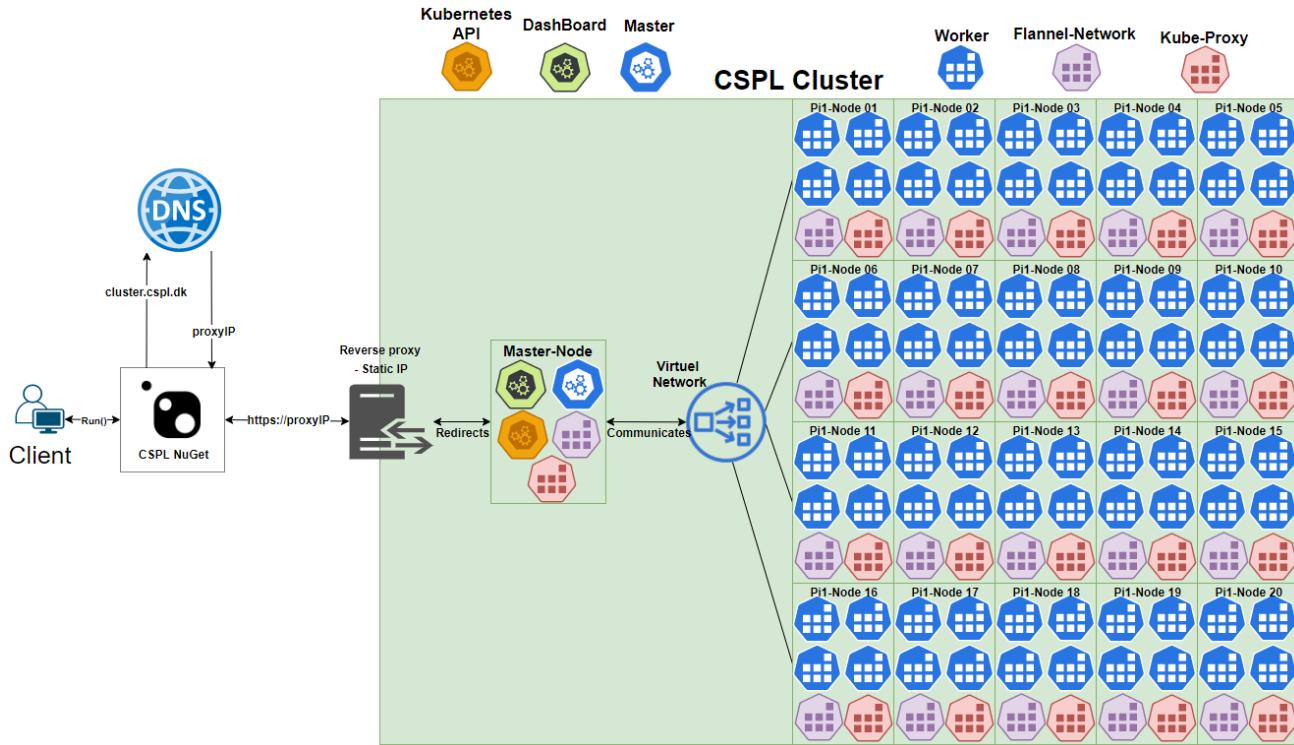


Figure 18: Kubernetes orkestrering i CSPL.

8.5 Kommunikation mellem frontend og master

Kommunikationen imellem frontend og master foregår ved subscription på data i tabellerne i Cloud Firestore databasen. Et sekvensdiagram som viser dette flow ses på figur 19.

Frontenden subscriber på data og når master så modificere tilføjer eller sletter den data, notificeres frontenden og den opdaterer viewet.

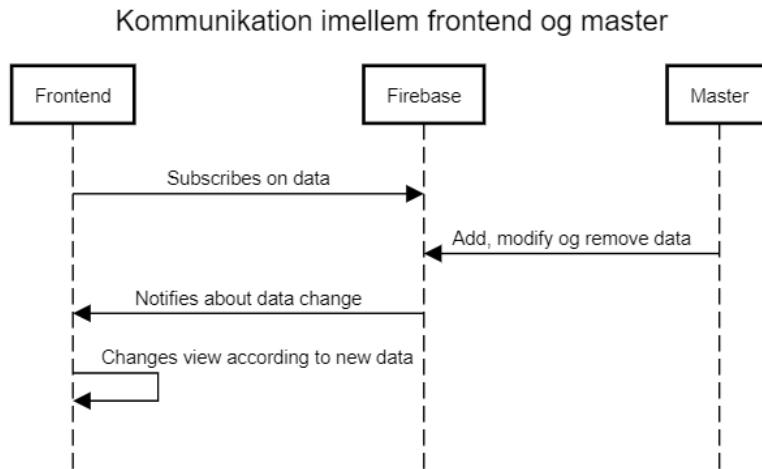


Figure 19: Kommunikation imellem frontend og master

8.6 Hardware arkitektur

På figur 20 er der vist en oversigt over alle hardware komponenter clusteret. Clusteret får internet igennem et Technicolor modem, hvor routeren i kombination med switchen giver internet til systemet. En mere detaljeret forklaring om hvert hardwarekomponent henvises der til afsnit 4.1.

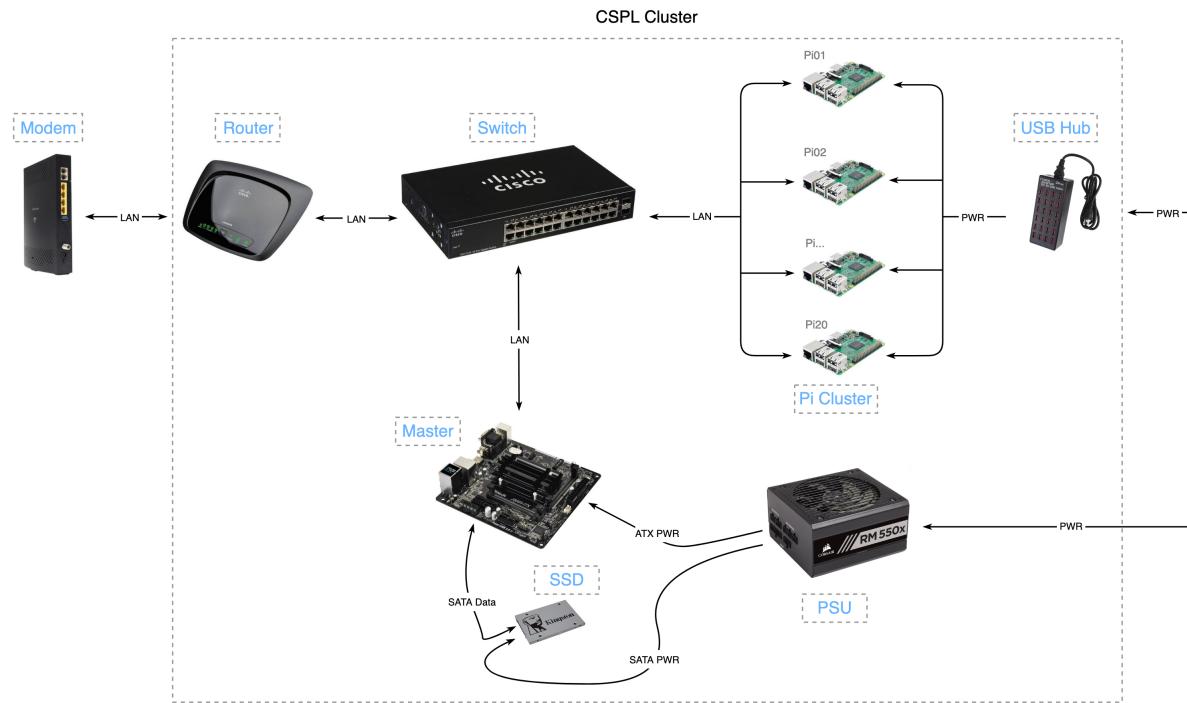


Figure 20: Oversigt over hardwarekomponenter for CSPL.

8.7 Port forwarding

Ved opsætning af systemet er der i alt åbnet seks porte.

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

På figur 21 ses en oversigt over hvordan master-servicen tilgås. Klienten tilgår domænet cluster.cspl.dk via NuGet-pakken og rammer igennem DNS'en modemets IP'en 62.107.0.222 og dens firewall på port 443. Denne port er normalt åbnet til standard TLS trafik, men grundet at porten skal route til en service, port forwardes den til router IP'en 192.168.87.126 på port 8020. Når et request rammer routeren, er der opsat en regel for hvor trafikken skal dirigeres hen, der port forwardes til master IP'en 192.168.1.100 på port 8020. Grundet at master-noden og de dertilhørende worker-nodes har deres firewalls deaktiveret, rammer klientens request direkte det ønskede endpoint.

For flere informationer omkring port forwarding, henvises der til bilag afsnit 7.12

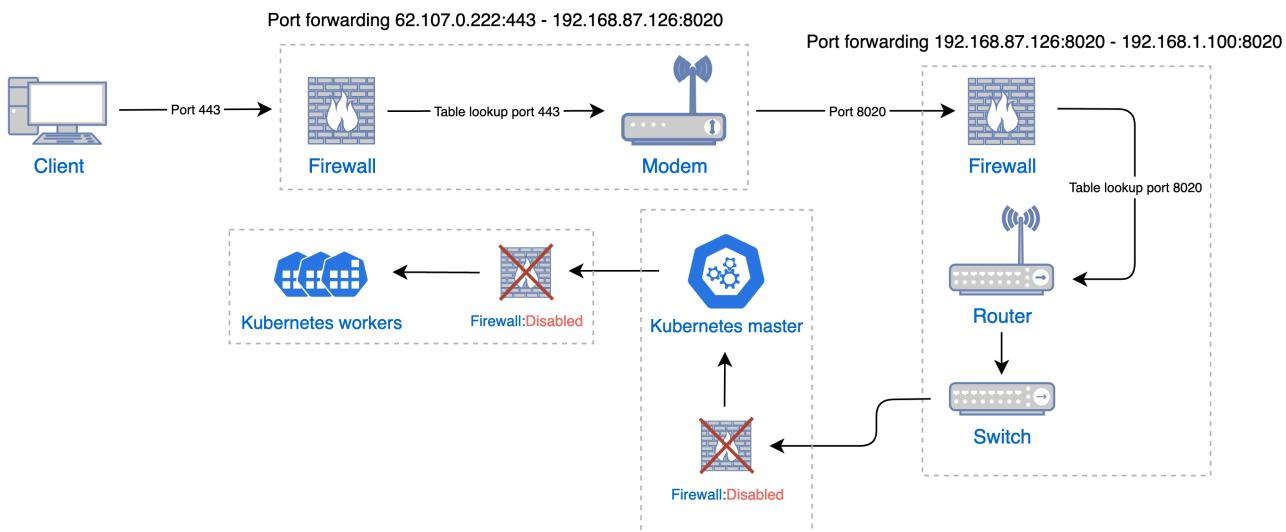


Figure 21: Port forwarding fra klient til master.

9 Design

Følgende afsnit beskriver designet af komponenter og funktionalitet i systemet.

9.1 ClusterSupportedParallelLibrary NuGet-pakken

For at en bruger/klient kan kommunikere med systemet, skal han gøre brug af ClusterSupportedParallelLibrary NuGet-pakken. NuGet-pakken er designet med henblik på at være brugervenlig for klienten. Da hver Raspberry Pi har fire workers/pods som beskrevet i afsnit 4.9, skal klienten ikke selv tænkt på at starte tråde. Han skal blot implementere interfacet IClusterParallelizer. Som det ses i koden på figur 22 skal klienten implementere tre forskellige metoder. Interfacet gør det muligt for API'er i clusteret at eksekvere koden vha. reflection, se bilag afsnit 7.4.2. Følgende er en liste over interfacets metoder.

- **Split(dataSource)** definerer hvordan det givne datasæt skal splittes op i delmængder. Disse delmængder sendes med som parameter til Parallel-metoden. Antallet af delmængder som Split returnerer, resultere i hvor mange workers kunden får tildelt.
- **I Parallel(subDataSource)** defineres det kode som skal eksekveres på hver worker-pod. Parallel genererer og returnerer et delresultat til det endelige resultat.
- **Collect(subResults)** definerer hvordan de forskellige delresultater fra Parallel-metoden skal samles for at producere det endelige resultat.
- **API-key** skal angives for at kunden kan få adgang til clusteret.

```
④ IClusterParallelizer.cs
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 22: Interface'et i CSPL NuGet-pakken.

IClusterParallelizer er generisk, hvilket gør, at klienten kan bruge forskellige typer i hans implementering. En liste over og beskrivelse af typerne i IClusterParallelizer kan ses i bilag afsnit 6.4. Klienten både kan implementere interfacet i et 32 eller 64-bit system, se bilag afsnit 4.4 for mere information.

På figur 23 ses et sekvensdiagram, som viser hvor i systemet, metoderne i interfacet bliver kaldt.

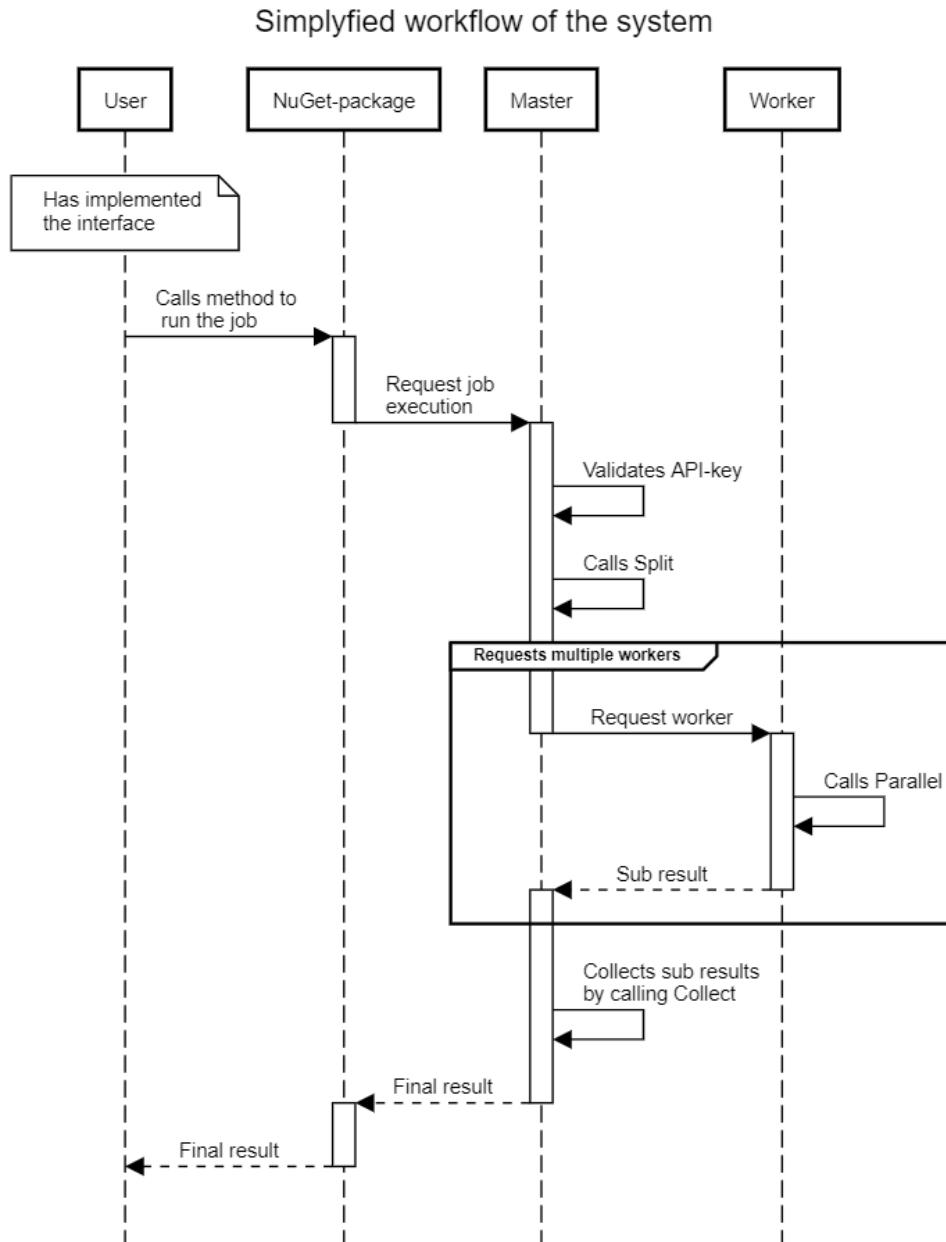


Figure 23: Sekvens for hvordan IClusterParallelizer metoder bliver eksekveret i CSPL systemet.

En klient kan definere properties i interfacet vha. runtime properties. Normale properties (compile time properties), bliver ikke reflekteret korrekt op på master, da objekter ligger i memory på klientens program. Derfor vil de properties have deres types default værdi. Har brugeren ikke gjort brug af NuGet-pakkens runtime properties egenskaber, vil hans properties af denne grund risikere ikke at have de rigtige værdier. En dybere og mere detaljeret forklaring kan læses i bilag afsnit 6.5.

9.2 DLL-pakke

På figur 24 ses den pakke af DLL'er, data og information som sendes fra NuGet-pakken til master og videre til worker. Disse attributter gør det bla. muligt at lave reflection og dermed eksekvere brugerens kode på clusteret, se afsnit 10.2 som beskriver hvad der sker ved modtagning af pakken på master. For yderligere beskrivelse af pakkens attributter henvises til afsnit i bilag 7.4.1.

```
④ 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 24: DLL-pakke til sending af DLL'er, data mm.

9.3 Load balancing

For at opnå den ønskede arbejdsfordeling beskrevet i analyse afsnit 4.11, er der designet en LoadBalancer. Valget stod mellem custom load balancing eller en label switching løsning via Kubernetes. Fordeler og ulemper for de to løsninger, samt valget af custom load balancing frem for label switching løsningen, kan ses i bilag afsnit 6.1.

9.3.1 Custom load balancing

LoadBalancer'en er designet med intentionen om at få lige fordeling i clusteret, således at alle cores i Raspberry Pi clusteret bliver udnyttet fuldt ud, se bilag afsnit 4.11.1. LoadBalancer'en er designet således, at den holder en cache i memory med information om de forskellige worker-pods. Heriblandt information om de er ledige eller optaget samt deres virtuelle-IP'er i clusteret. Grundet worker-pods livscyklus i clusteret, skal registreringen og fjernelse af workers opdateres løbende, hvilket bliver gjort vha. mængdeoperationer. For en dybere forklaring om Kuberneates pod's livscyklus henvises der til bilag afsnit 6.1.2.1

9.3.2 Kubernetes API

Kubernetes API'et som hostes på master-noden, har altid det nyeste data omkring pods i clusteret. LoadBalancer'en kontakter Kubernetes API'et, for at få information om alle worker-pods. Den kan ud fra disse informationer registrere de virtuelle worker-pod IP'er i sin cache, og samtidig holde styr på hvilke worker-pods som er "free" eller "occupied". Det er nødvendig, at LoadBalancer'en opdaterer cachen, hver gang der kommer et request til clusteret, da der kan være kommet nye pods til eller være pods som er døde.

9.3.3 Håndtering af nye og gamle worker-pods

Når LoadBalancer'en opdaterer sin cache, enten med nye eller døde, beregnes mængden af pods som skal fjernes eller tilføjes vha mængdeoperationen **difference** [21]. For de forskellige beregninger af dette, refereres til bilag afsnit 6.1.2.4 & 6.1.2.3.

9.4 Køstruktur

Når klienter requester systemet om at få x-antal workers, til eksekvering af deres arbejde, kan situationen opstå hvor der ikke er nok ledige workers. Følgende afsnit omhandler hvilket designvalg der er taget til køstrukturen, for flere informationer henvises der til bilag afsnit 6.3.

9.4.1 FIFO-kø vs. FLEX-kø

Det er valgt at designe køstrukturen som en FIFO-kø, grundet den starvation der kan risikere at opstå i FLEX-køen, se bilag afsnit 6.3. Hvis den forreste klient i en FLEX-kø requester 50 workers, men der kun er 20 ledige, vil den næste klient i køen kunne få lov til tage nogle af de 20 ledige worker-pods. Hvis dette fortsætter vil der opstå starvation og den forreste klients arbejde kan risikere aldrig at blive skeduleret. Det skal derfor ikke være muligt at udskyde andres arbejde ved at hoppe foran, og derfor er FIFO-køen det valgte design.

9.4.2 FIFO-kø design

På figur 25 ses en illustrering af designet for køstrukturen (ClusterQueue). Det ses at der ligger tre klienter i kø med følgende 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. Når et allerede igangværende job terminere, vil flere workers blive ledige for den forreste klient i køen. Præcist hvordan klienter venter i køen, samt venter på at få workers, refereres der til bilag afsnit 6.3.4 & 6.3.1

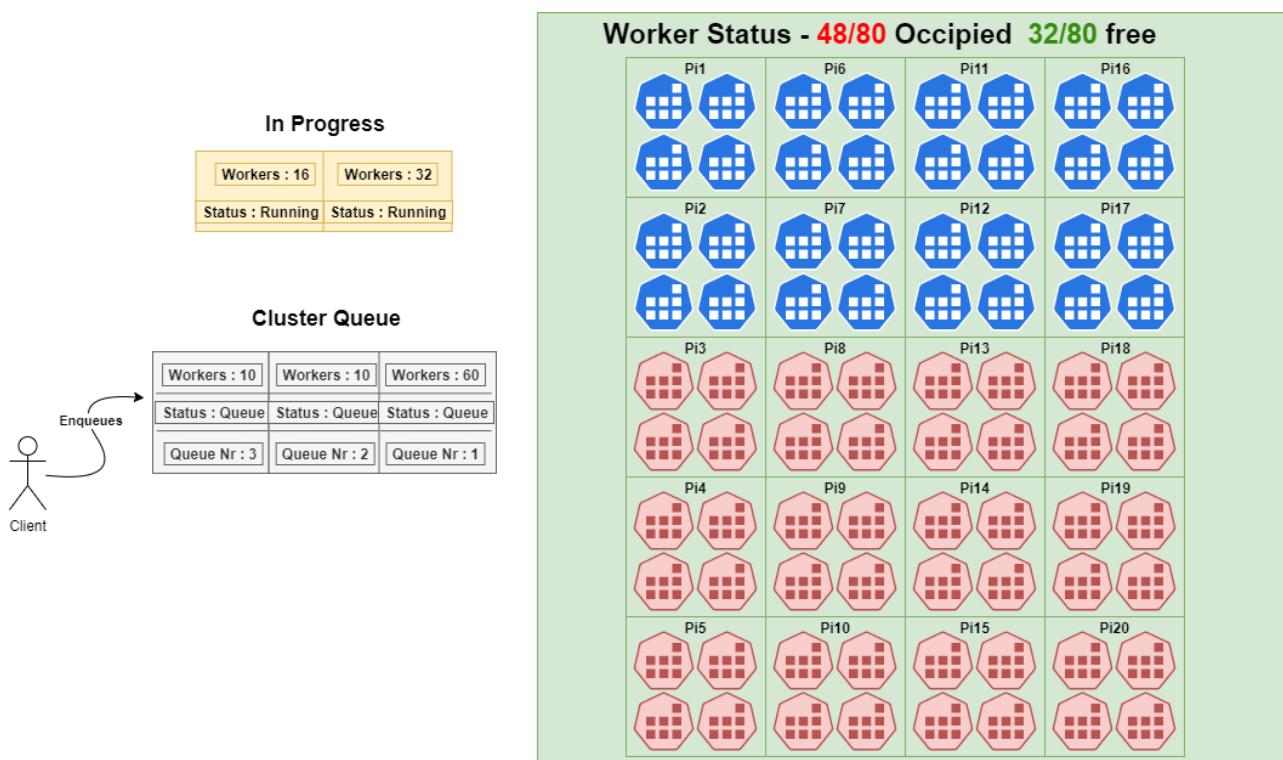


Figure 25: Køstruktur.

9.4.3 Klient-status

For at holde styr på hver klients job status, er der designet en QueueStatus. QueueStatus kan have fire forskellige værdier; InQueue, Running, Failed eller Completed. Denne status logges på Firebase databasen til brug for live data opdateringen på frontenden. På figur 26 ses de forskellige transitioner mellem de forskellige states. En mere detaljeret gennemgang kan ses i bilag afsnit 6.3.3.

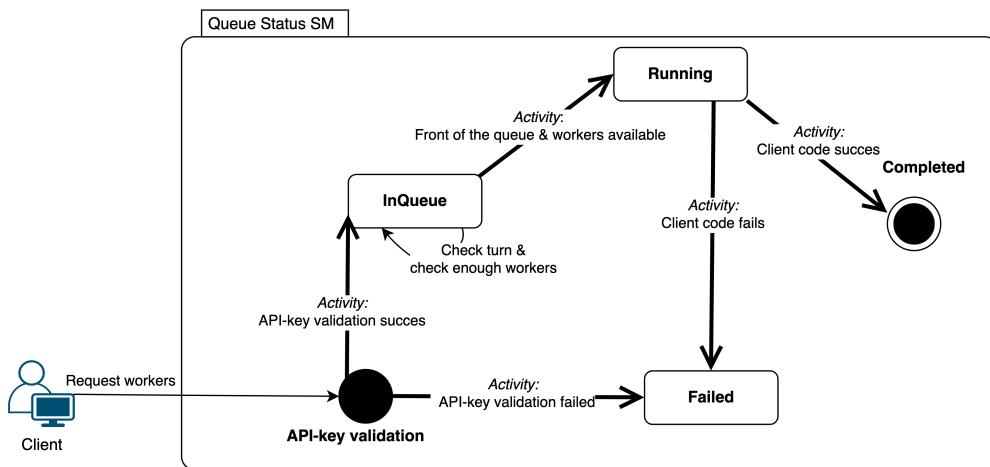


Figure 26: State machine Queue status

9.4.4 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, dette kan føre til race conditions. For at imødekomme problemet er der udviklet en QueueUpdater.

QueueUpdater'en er en baggrundstråd, som hele tiden lytter på køen for at tjekke status fra de forskellige klienter. Klienterne vil løbende opdatere deres status til enten Running, Completed, Failed og derved signalere til QueueUpdater'en. QueueUpdater'en holder styr på at opdatere køen og derefter give frontenden besked om at data har ændret sig. Når QueueUpdater'en 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. Når et job bliver sat i Running, skal den fjernes fra køen, således at det kan blive den næstes tur. Som det ses på figur 27 vil ClusterQueue'en kun have elementer som har status InQueue eller Running. JobInProgressList vil derimod kun have elementer som er Running, Completed eller Failed.

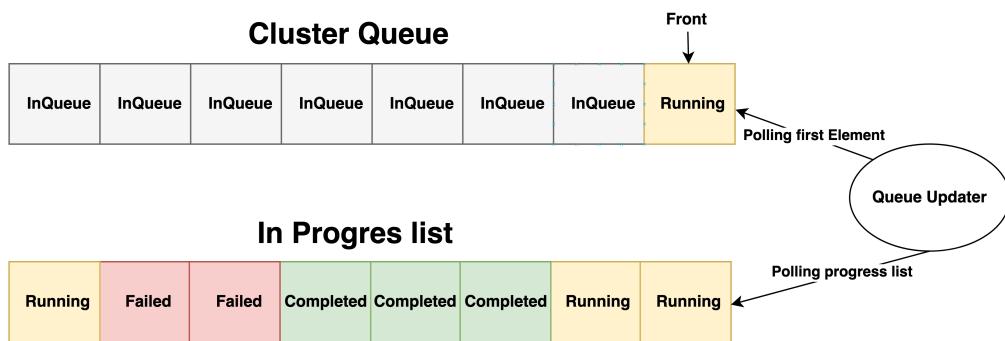


Figure 27: QueueUpdater håndterer ClusterQueue & Jobs in Progress

For informationer omkring hvordan QueueUpdater'en poller to de lister og håndtere QueueStatus states, henvises der til bilag 6.3.5.

10 Implementering

Følgende afsnit giver en beskrivelse af hvordan de vigtigste koncepter i systemets er implementeret.

10.1 ClusterSupportedParallelLibrary NuGet-pakken

NuGet-pakken udstiller interfacet IClusterParallelizer og funktionalitet til at starte eksekveringen i systemet. Det omhandler blandt andet at finde dependencies til klientes kode samt at pakke og sende de nødvendige

arfefakter afsted til clusteret. Se bilag afsnit 7.1.1 for et teknisk sekvensdiagram for dette.

10.1.1 DLL dependencies og filter

Der er implementeret en metode, som rekursivt finder alle dependencies for en given DLL-fil og deres dependencies. En DLL kan have multiple dependencies, hvorfra disse DLL'er også kan have yderligere dependencies. Dette kan visualiseres vha. et dependency-træ som ses på figur 28.

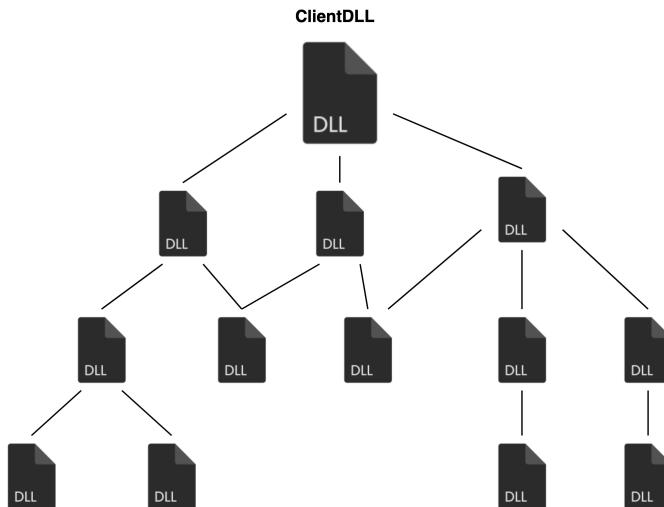


Figure 28: Dependency-træ.

Ovenstående er nødvendigt for, at API'erne i clusteret har mulighed for at loade disse assemblies ind og dermed eksekvere klientens program vha. reflection. Klientens program kan have dependencies som stammer fra .NET Core og de skal ikke sendes med til clusteret da workers/Pi's allerede har installeret .NET Core og de dertil-hørende DLL'er. Derfor bliver alle dependencies holdt op imod en liste af alle de DLL'er der ligger i .NET Core og dermed bliver filtreret fra. Listen/filteret af .NET Core-specifikke DLL'er er lavet og udledt vha. PowerShell, for mere information om filteret se bilag afsnit 7.3. DLL-dependencies som stammer fra 'third-party'-libraries som f.eks. en NuGet-pakke skal derimod sendes med til clusteret.

10.1.2 ClusterRunner

Der er implementeret funktionalitet som gør det let for brugeren af systemet at igangsætte hans arbejde på clusteret. Komponenten hedder ClusterRunner og gør brug af den rekursive DLL-dependency-metode nævnt i afsnit 10.1.1. Funktionen 'Run' starter programmet og returnerer det endelige resultat. Som det ses på figur 29 modtager ClusterRunner'en klientens kode implementeret via IClusterParallelizer interface'et i constructoren. Ud fra denne klasse-implementation finder ClusterRunner'en alle nødvendige DLL-dependencies.

```
 ClusterRunner.cs  
1 var monteCarloClient = new ClusterRunner<double, double, double, double>(new MonteCarlo());  
2 double pi = await monteCarloClient.Run(8000000);
```

Figure 29: Eksekvering af klient-kode på clusteret.

10.1.3 Sikker sending af klient-kode

NuGet-pakken sender DLL-pakken til <https://cluster.cspl.dk> som er en DNS for IP'en 62.107.0.222 med et SSL/TLS certifikat. Sending af DLL-pakken er sikret med HTTPS igennem hele clusteret, for mere information se bilag afsnit 7.8.3.2.

10.2 Master API

Når master API'et modtager DLL-pakken fra klienten, kan master API'et loade de sendte DLL'er og DLL-dependencies, ind i en assembly load context hvori klientens kode kan eksekveres. Ud fra den sendte DLL-pakke kan master API'et finde klientens implementerede klasse, ud fra navnet på klassen og hvilket namespace klassen ligger i. Da master API'et ved, at klienten har implementeret IClusterParallelizer, ved master API'et altid hvilke metoder der skal kaldes. Master API'et kalder klientens Split-metode for at opdele klientens datasæt. Gennem LoadBalancer'en requestes det antal workers, der er nødvendigt for at processere arbejdet. Collect-metoden bliver kaldt, når de forskellige delresultaterne returneres fra de forskellige workers og derefter returneres det endelige resultat til klienten. Specifikt hvordan disse metoder bliver kaldt via type resolving og reflection, henvises der til bilag afsnit 7.4.2. For et teknisk sekvensdiagram for dette se bilag afsnit 7.1.2.

10.3 Worker API

Worker API'et gør brug af de samme komponenter som master API'et til at loade DLL'er og eksekvere klientens metoder vha. type resolving og reflection. Den kalder IClusterParallelizer interfacets Parallel-metode og sender resultatet tilbage til masteren. For et teknisk sekvensdiagram for worker API'et se bilag afsnit 7.1.3.

10.4 Fejlhåndtering

For at håndtere de exceptions der måtte opstå i det kode, som bliver eksekveret på clusteret, er der implementeret funktionalitet, som gør at de bliver sendt tilbage til klienten og kastet igen i klientens program, dette ses på figur 30.

Derudover har klienten mulighed for at benytte og håndtere exceptions som er specifikke for systemet og som klienten kan forvente at få ved typiske fejl, som f.eks. forkert API-key. For yderligere information omkring se bilag afsnit 7.13.

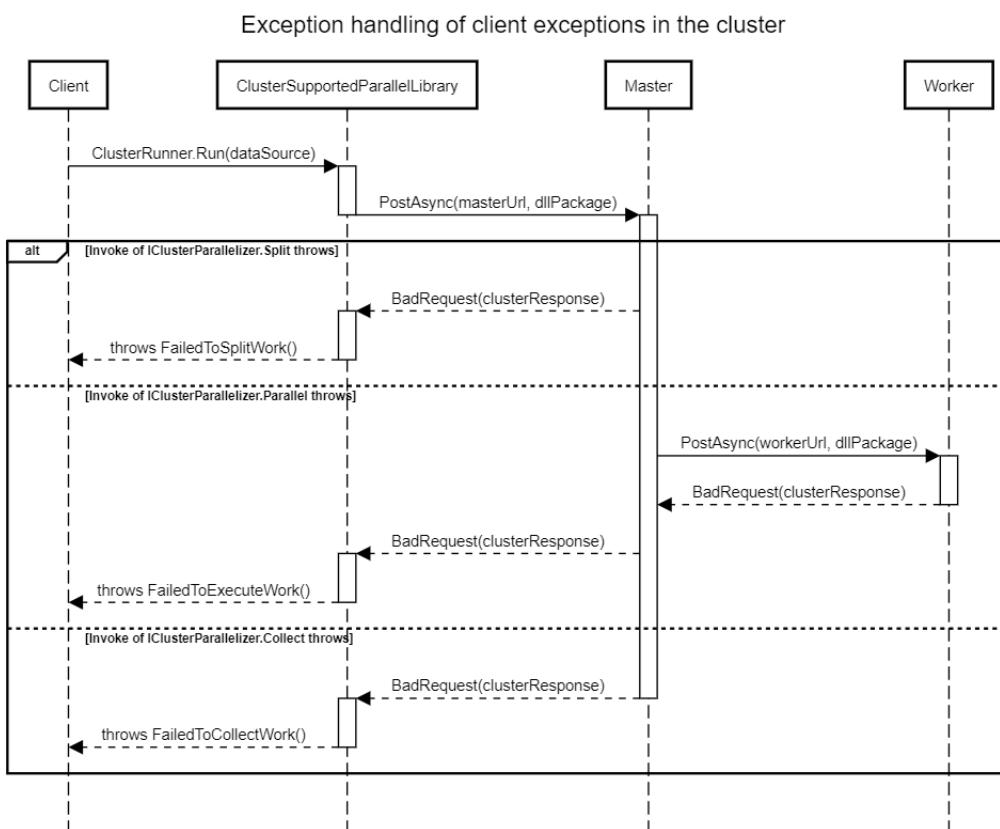


Figure 30: Sekvensdiagram for fejlhåndtering af klient-exceptions i clusteret.

10.5 Opsætning af Kubernetes

Kubernetes er opsat i clusteret og konfigureret vha. `kubeadm`. Det interne overlay netværk er opsat med Flannel [12]. Alle noder i clusteret er konfigureret med statiske IP'er, hvilket er en forudsætning for at opretholde et Kubernetes cluster. Opsætningen af Kubernetes har bragt forskellige problemstillinger som eksempelvis opsætning af firewall, dynamiske IP'er, overlay netværk mm. Håndteringen og adressering af disse problemer er beskrevet i bilag afsnit 7.5.

Der er udarbejdet en detaljeret guide til hvordan clusteret er opsat step for step, se bilag afsnit 8. I den forbindelse er der benyttet Cluster SSH til at administrere flere noder af gangen, se bilag afsnit 8.16.

10.5.1 Deployment og konfigurering

Deployering af master API og worker API'er bliver gjort via UNIX Shell på master-noden, vha. `.yaml`-filer som understøttes af `kubectl`. Heri er det bla. specificeret hvilket image der skal hostes i pod's. For yderligere information omkring deployering og opsætning se bilag afsnit 7.6 og 8.16.

Hver worker-node er konfigureret til kun at skedulere seks pods ad gangen, således at der maks kan være fire worker-pods (plus én Flannel-pod og én kube-proxy-pod) på en node. Denne konfiguration er nødvendigt ift. "true" parallelisme. Se bilag afsnit 7.6 for yderligere information omkring denne konfigurering.

10.6 Forurening af pods i clusteret

Ifm. at der importeres DLL'er og eksekveres kode derfra på master og worker API'erne, kan der opstå forurenning af systemet. I den forbindelse er der implementeret en CustomAssemblyLoadContext som loader og unloader assemblies. Således at brugte assemblies bliver fjernet og dermed ikke hober sig op, se bilag afsnit 7.7.1 for mere information. Derudover bliver worker-pods slettet efter brug, for at sikre at alle worker-pods er friske og identiske hver gang de bliver efterspurgt, se bilag afsnit 7.7.2.

10.7 Autorisation

API-nøglen bliver genereret på frontenden og valideret på backenden. API-nøglen er den eneste måde for brugeren at interagere med clusteret. For at en nøgle bliver unik hver gang den genereres, laves nøglen ved at hashe den nuværende dato i millisekunder med klokkeslettet og tilføje salt. Biblioteket som er brugt til dette er `bcrypt` [22], som er en npm pakke som bruges til at hashe.

Den hashet værdi er klientens API-nøgle. Denne sendes op til firestore via. en reference til klientens primæreøgle, hans e-mail. På figur 31 er der vist en genereret API-Key og dens antal kald tilbage.

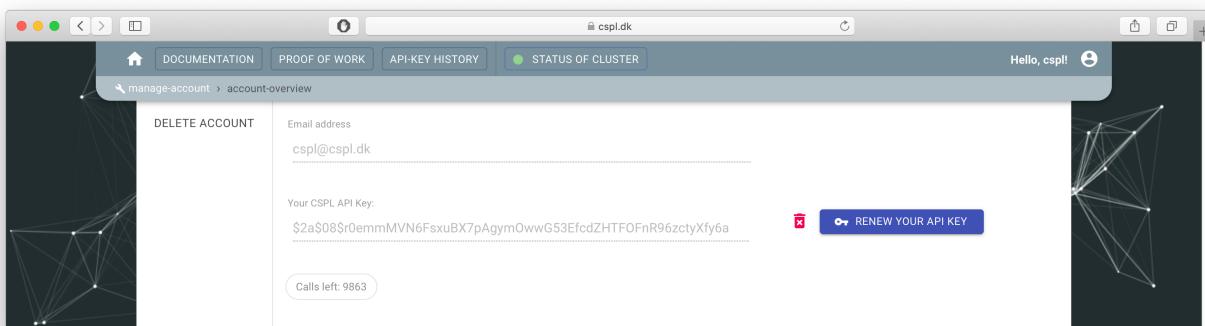


Figure 31: Account Management, API-key med antal kald tilbage

Når klienten efterspørger arbejde på clusteret, vil han først blive sat i kø, når hans API-nøgle er 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, bliver der kastet en exception og klienten får en besked om, at hans nøgle ikke er valid eller at der ikke er nok kald tilbage.

I MasterController'en, på det endpoint klienten rammer, vil der som noget af det første, kaldes `ValidateAPIKey(dllPackage.APIkey)`. Denne kaldes med API-nøglen fra klientens API-Key property i IClusterParallelizer. UserValidator henter et snapshot fra Firebase og på baggrund af dette valideres nøglen og om den har nok antal klient-kald tilbage.

For flere informationer om API-nøglen, henvises der til bilag afsnit 7.11.

10.8 API-Key history view

API-Key viewet på hjemmesiden gør brug af Firebase-komponenten, se bilag afsnit 7.10, og subscriber derigenom på "Calls"-collectionen i Firebase databasen. Målet med API-Key viewet, er at kunne vise data omkring brugerens historik og igangværende jobs i realtid.

Den tabel som der subscribes på i databasen er "Calls"-collectionen. På figur 32 ses denne collection for en tilfældig bruger. "Calls"-collectionen ligger som sub-collection til en specifik API-nøgle som tilhører brugeren. Når data i denne tabel ændrer sig, bliver hjemmesiden i klientens browser notificeret og tabellen på hjemmesiden opdateres med nyt data, hvilket giver brugeren en følelse af live data.

\$2a\$08\$SrFSocTIAYQqjjBhuJbhOqncyA... :	Calls :	GObGpmIf3TdHUhyHbf4A :
+ Start collection	+ Add document	+ Start collection
Calls >	GObGpmIf3TdHUhyHbf4A >	+ Add field
	ReYPtCPptRWMwobnswFe	ClassName: "MonteCarlo"
	afQD3U9RRwSf2Jtg0gZf	ExceptionMessage: null
	xd7lGYKgDgII0B5HaQHn	ExecutionTime: 8.723
	zWJpJvRC1jNJF3HB9yrs	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 32: Firestore dokument database collection for jobs

Den notification som applikation modtager indeholder et snapshot [8] af data som har ændret sig. Snapshotets data kan have tre forskellige states som er henholdsvis "Added", "Modified" eller "removed". Applikationen gør noget forskelligt i hver scenarie, der henvises til bilag afsnit 7.9 for en mere grundig gennemgang.

11 Test

Følgende afsnit beskriver forskellige former for testing, der er blevet lavet for at sikre kvaliteten af systemet, heriblandt unittests, integrationtests, manuelle tests, performancetest og accepttest. Der er opsat en byggeserver med TeamCity, som kører automatiske pipelines. Test frameworket NUnit [19] er benyttet til at implementere unittest og integrationstest.

Informationer omkring opsætningen, pipelines og versionering i byggeserveren, henvises der til bilag afsnit 3.11.

11.1 Unit test

I takt med at systemets forskellige programmer og de dertilhørende komponenterne er blevet implementeret eller refaktureret, er der skrevet unittests af de komponenter, som har stor indflydelse på systemet. Unittest har sikret softwaremodulernes funktionalitet gennem tilføjelse af nye funktionaliteter eller refaktureringer. Der er i nogle komponenter benyttet dependency injection til at stubbe afhængigheder ud vha. NSubstitute [23].

Alle unittest er blevet opsat med udgangspunkt i AAA test-patteren [9], for at holde test så overskuelige som muligt. Yderligere er der blevet lavet code coverage analyser for at få en pejling om hvor fyldestgørende disse test er.

Oversigt over de forskellige komponenter i systemet som der er skrevet unittests til:

- **Master:**

- UserValidatorTest
- ClusterQueueTest
- LoadBalancerTest
- KubernetesFectherTest
- WorkDelegatorTest

- **CSPL.Common**

- TypeResolverTest

- **CSPL (NuGet-pakke)**

- ReferencedAssembliesFilterTest

For mere information om modul/unittests og dependency injection, se bilag afsnit 9.4.

11.2 Integration

For løbende at sikre at systemet stadig er funktionelt, er der udviklet automatiske systemintegrationstests. De forskellige systemetsintegrationstests kører på det rigtige cluster og er derfor så tæt på et rigtigt scenarie som muligt. På denne måde sikres det, at nye features ikke bryder eksisterende funktionalitet. En figur og en dybere forklaring på integrationstests kan læses i bilag afsnit 9.5.

Oversigt over de forskellige integrationstests i CSPL systemet:

- **CSPL (NuGet-pakke)**

- MonteCarloTest, se bilag afsnit 9.5.1
- BruteForceTest, se bilag afsnit 9.5.2
- ExecutionHandlingTest, se bilag afsnit 9.5.3

Ideelt ville man have et test-cluster for ikke at belaste produktions clusteret.

11.3 Manuel test

Der har været benyttet manuel testing for at teste funktionalitet, som ikke kunne dækkes med automatiserede tests. Der er benyttet forskellige metoder og værktøjer til manuel testing, alt fra info på hjemmesidens API-key view til kubectl logs.

Når nyt funktionalitet er blevet implementeret, har en af de andre gruppemedlemmer testet funktionaliteten. Dette har givet et bedre resultat, da der er fundet kritiske fejl og der er opstået diskussioner om eventuelle forbedringer. Manuelt testing har af denne grund været et essentielt værktøj til kvalitetssikring af produktet, både backend og frontend.

11.3.1 Test med ekstern softwareudvikler

Da alle gruppemedlemmer er bias til produktet og af denne grund forventer, at produktet bliver benyttet på en specifik måde, er det en fordel at have en ekstern softwareudvikler til at afprøve produktet. En ekstern softwareudvikler kan give feedback i form af ideer og forbedring af både hjemmeside og backend. Af denne grund har en studerende på 6. semester af Software Teknologi, været med til at afprøve systemet. Dette har givet feedback på bla. brugervenlighed af systemet, fejl, mangler og forbedringer. Tilbagemeldingen kan findes i bilag afsnit 9.3.

11.4 Performancetests

Følgende test 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.

Der er taget inspiration fra Josh Kiepert's rapport om et Raspberry Pi cluster [10].

Til at teste systemets performance benyttes tre forskellige algoritmer. Algoritmerne er henholdsvis, Monte-Carle Pi approximation [1], Brute force SHA256 hash algoritme [3] og en billedkomprimerings-algoritme som kan ses i bilag afsnit 9.6.8.

Følgende performancetests vil sammenligne systemet med en laptop Lenovo T480s og en mere kraftig desktop. Figur 33 viser en oversigt over clusterets hardware og det eksterne hardware brugt til benchmarkstests.

Følgende tests er også beskrevet og vist på hjemmesiden under 'Proof of Work'

<https://csp1.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

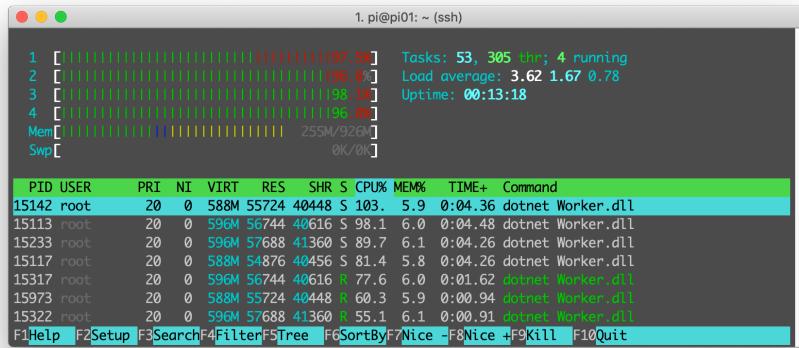
Figure 33: Test Specs

Prisforsk

- **Laptop Lenovo T480s** 10.000kr.
- **Desktop** 11.000kr.
- **CSPL System - 20 nodes Raspberry Pi's** 7.000kr.

11.4.1 Udnyttelse af CPU i systemet

For at teste at systemet udnytter så meget CPU i clusteret som muligt, er der udført manuelle test vha. htop til at monitorere, at worker API'erne har kunne udnytte alle CPU-kerner på Raspberry Pi's, se figur 34.



PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
15142	root	20	0	588M	55724	40448	S	103.	5.9	0:04.36	dotnet Worker.dll
15113	root	20	0	596M	56744	40616	S	98.1	6.0	0:04.48	dotnet Worker.dll
15233	root	20	0	596M	57688	41360	S	89.7	6.1	0:04.26	dotnet Worker.dll
15117	root	20	0	588M	54876	40456	S	81.4	5.8	0:04.26	dotnet Worker.dll
15317	root	20	0	596M	56744	40616	R	77.6	6.0	0:01.62	dotnet Worker.dll
15973	root	20	0	588M	55724	40448	R	60.3	5.9	0:00.94	dotnet Worker.dll
15322	root	20	0	596M	57688	41360	R	55.1	6.1	0:00.91	dotnet Worker.dll

Figure 34: htop kommando på en Raspberry Pi i clusteret.

11.4.2 Raspberry Pi 3B vs. Raspberry Pi 3B+

Som klient i systemet kan der forekomme varierende eksekveringstider grundet forskelligt hardware på workers i clusteret. Clusterets worker noder består af både Raspberry Pi 3B og 3B+, som ikke er lige kraftige. Der er estimeret en potentiel forøgelsen i performance til at være omkring 16,35%, hvis en klient udelukkende for tildelt Raspberry Pi 3B+. Får klienten tildelt bare én enkelt Raspberry Pi 3B, vil denne være en flaskehals. For mere information om dette se bilag afsnit 9.6.1.

11.4.3 Docker container overhead vs. traditional deployment

For at finde ud af om Docker giver et overhead, er det blevet testet hvad det koster i performance at hoste systemets applikationer i Docker containers. Der er blevet kørt en MonteCarlo-algoritme i en Docker container og i en traditional deployment. Der er fundet frem til et gennemsnitligt overhead ved brug af Docker på 4,39%. Der refereres til bilag afsnit 9.6.2 for en nærmere forklaring.

11.4.4 CSPL with Kubernetes vs. Docker container overhead

For at teste det fulde overhead i systemet hostet i Kubernetes, er der blevet kørt nogle tests ud fra MonteCarlo-algoritmen. Disse test sætter performance af det fulde system op imod performance fra samme algoritme kørt i Docker containere uden systemet og Kubernetes. I denne case med MonteCarlo-algoritmen er der fundet et overhead for systemet med Kubernetes på 1,90%. Overheadet ligger alene i at skulle køre programmet med Kubernetes, assembly loading, type beregning mm. For yderligere information omkring dette se bilag afsnit 9.6.3.

11.4.5 Desktop vs. Laptop vs. CSPL

Der er udført test som sammenligner systemet med en laptop og en desktop ift. skalering af antal af pods/tråde. Der er hovedsageligt taget udgangspunkt i MonteCarlo-algoritmen, men der er også udført test med en SHA256 brute force algoritme som sammenligner performance mellem disse.

11.4.5.1 MonteCarlo Pi approksimation

MonteCarlo - Basis

For at danne et udgangspunkt for sammenligning af selve systemet og de andre stykker hardware, kan der på figur 35 ses en graf for hvordan de hver i sær performer som single-node med forskellige antal tråde. Det er beregnet at desktop'en er 9,30 gange hurtigere end Rasberry Pi 3B og laptop'en er 6,46 gange hurtigere end Raspberry Pi 3B pr. tråd. For mere information omkring dette se bilag afsnit 9.6.5 og 9.6.5.1.

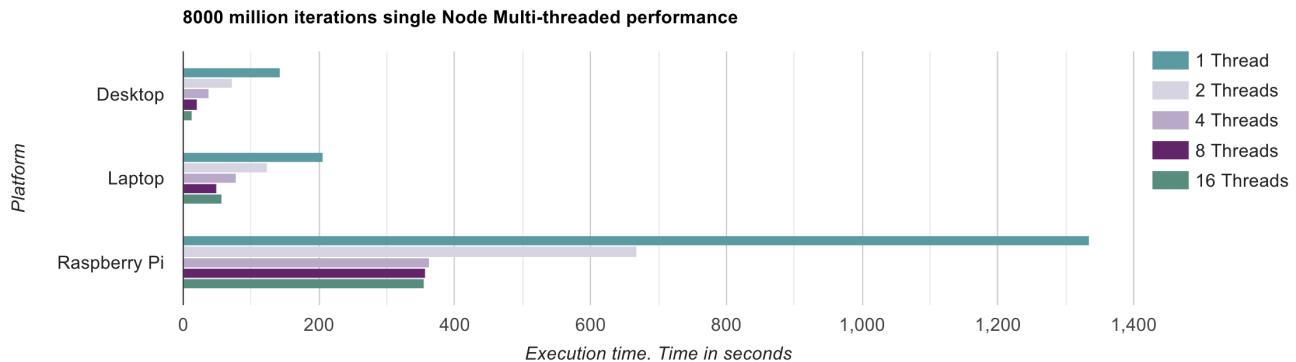


Figure 35: MonteCarlo 1-16 tråde - Desktop vs. Laptop vs. Raspberry Pi 3B.

MonteCarlo - Desktop vs. Laptop vs. CSPL

For at teste om systemet udkonkurrerer eller bliver udkonkurreret af de andre hardware komponenter, er systemet testet med MonteCarlo algoritmen med 8000 millioner iterationer. Testen presser systemet og de andre hardware komponenter til det yderste. Som det ses på figur 36 kan systemet ikke udkonkurrerer den kraftige desktop som har 8 kerner + 8 hyperthreads. Selv med 80 workers i systemet er den kraftige desktop stadig cirka tre gange hurtigere. Dermed ses det også at systemet allerede ud konkurrierer laptop'en efter brug af 32 workers.

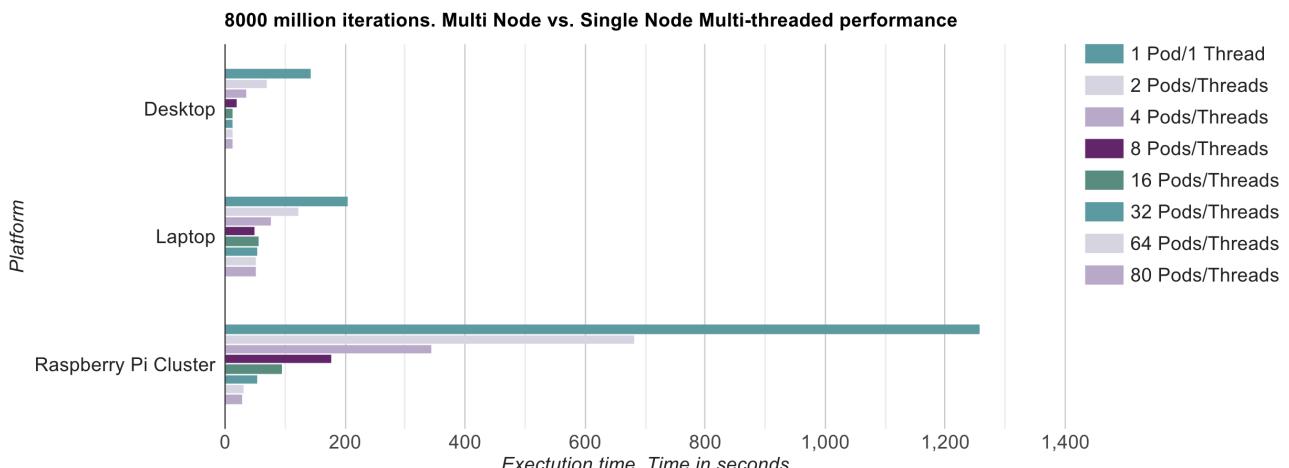


Figure 36: MonteCarlo CSPL 1-80 worker-pods/threads vs. Desktop vs. Laptop.

MonteCarlo - lineær gain skalering

Som det ses på figur 37 er MonteCarlo algoritmen blevet kørt med 8 milliarder iterationer vs. 100 milliarder iterationer. Derudover ses den teoretiske og ønskede lineær gain-faktor. Den teoretisk bedste skalering, er dér hvor det dobbelte antal workers resulterer i den halve eksekverings tid. På figur 37 ses det at desto mindre et job er, desto mere fremtrædende bliver det overhead, der er ift. at distribuere og samle data. Det er beregnet, at for 100 milliarder iterationer afviger den faktiske gain-faktor fra den teoretiske med 24,22%. For mere information omkring dette henvises til bilag afsnit 9.6.5 og 9.6.5.1.

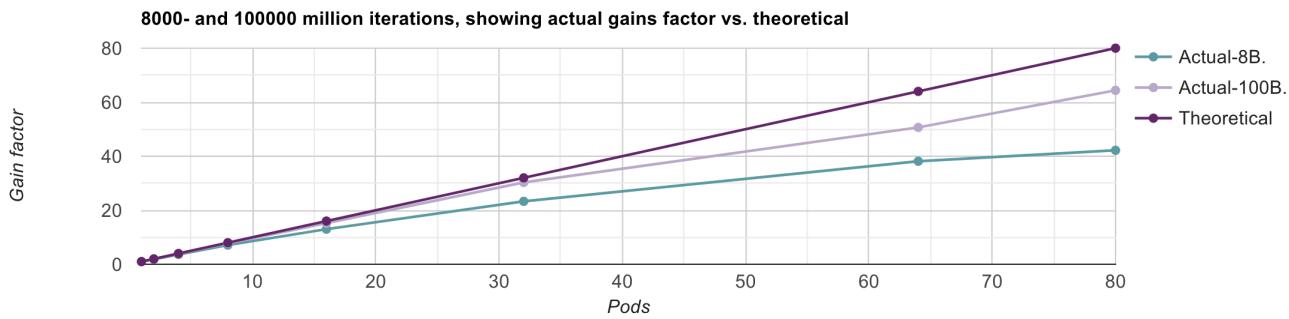


Figure 37: MonteCarlo lineær gain skalering.

11.4.5.2 Brute force SHA256 hash

På figur 38 ses en sammenligning mellem systemet, laptop'en og desktop'en. Algoritmen som der sammenlignes med, beregner parallelt hash-værdien for alle kombinationer af et fem-tegns password, som indeholder ASCII-værdier fra 48 til 122. Det ses at desktop'en er hurtigere end systemet, trods de færre CPU kerner. Dog udkonkurrerer systemet laptop'en. For mere information om dette se bilag afsnit 9.6.5 og 9.6.5.2.

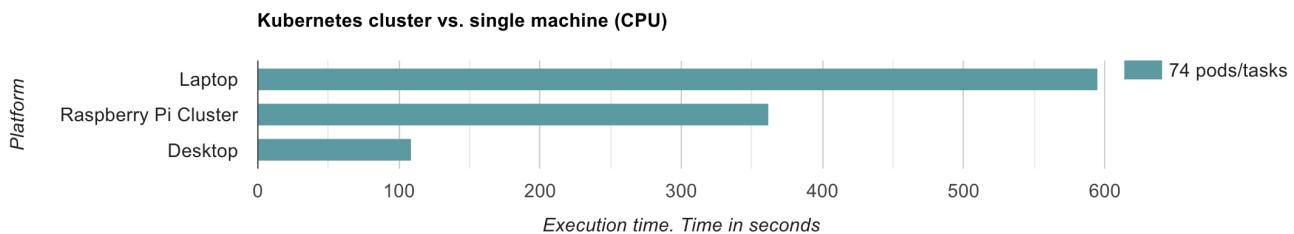


Figure 38: Parallel brute force algoritme der beregner SHA256 hash-værdier for alle kombinationer med 74 forskellige ASCII-værdier (fra 48 til 122). Algoritmen er kørt på hhv. laptop, Raspberry Pi cluster og desktop.

11.4.6 Data sending overhead

Der kan være et stort overhead, hvis en klient/bruger f.eks. sender et stort datasæt og arbejdet på workers ikke tager særlig lang tid. Overheadet ligger i at sende data til systemet, videredistribuere internt i systemet og distribuer af alle resultater hele vejen tilbage til klienten, se figur 39.

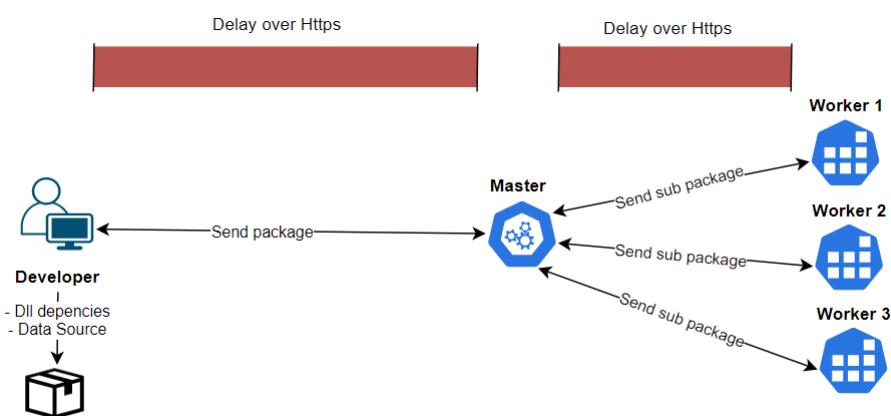


Figure 39: Delay over HTTPS.

MonteCarlo vs. billedekomprimering

På figur 40 ses to tests som er blevet udført på systemet for at finde ud af tidsfordelingen mellem CPU-tid vs. HTTPS-delay, med hhv. et lille og stort datasæt. 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.

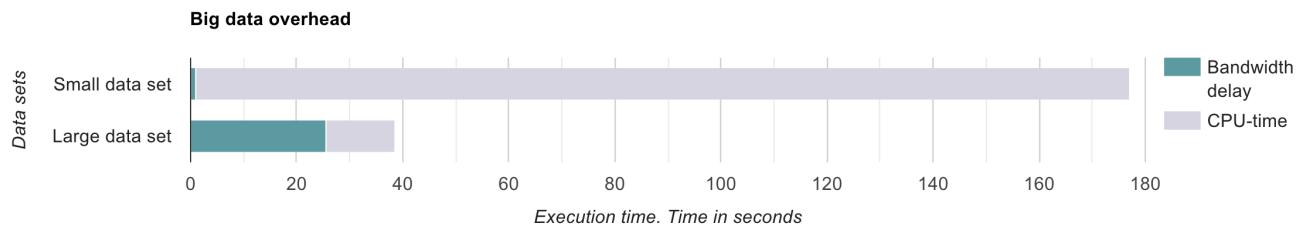


Figure 40: HTTPS-delay vs CPU-tid - MonteCarlo vs. billedekomprimering. Testet med en klient internet-forbindelse på 30/30 Mbps.

For mere information om dette se bilag afsnit 9.6.6.

11.4.7 Billedekomprimering

Der er kørt tests af systemet med en billedekomprimerings-algoritmen, disse tests har givet RAM overflows i worker-pods og har dermed vist svage sider af systemet. Dette er grundet svag hardware og dermed RAM-restriktioner i clusteret. For mere information se bilag afsnit 9.6.8.

11.4.8 MapReduce

Systemet blev testet med en MapReduce-algoritme som skulle finde alle antal af hvert ord i en text-fil. Dette har vist et overhead, i at sende datasættet til clusteret ift. den tid det tog at processere dataen. For mere information se bilag afsnit 9.6.9.

11.5 Accepttest

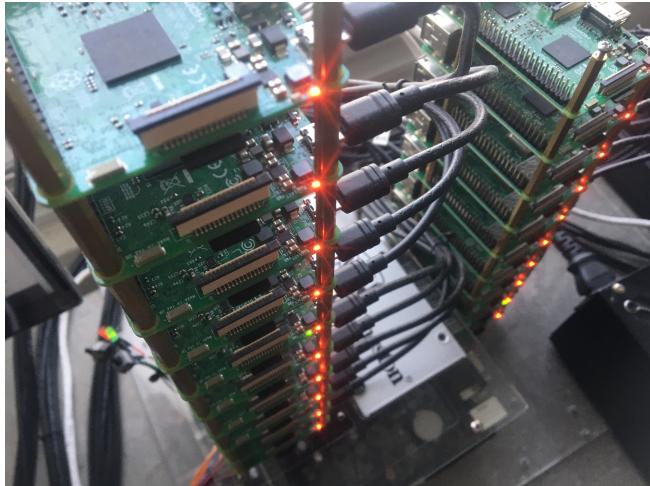
Der er udarbejdet en accepttest ud fra produktetes opstillede krav. Accepttestspezifikationen kan ses i bilag afsnit 9.7 og resultatet deraf kan ses i bilag afsnit 9.8.

12 Resultater

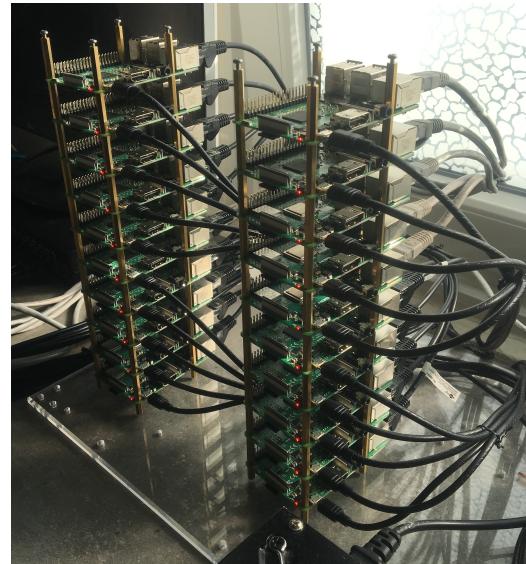
Disse resultater er opnået på baggrund af den kravsspecifikation og problemformulering/projekt mål, som er udarbejdet i forbindelse med dette projekt. Kravsspecifikation og accepttesten er med få undtagelser opfyldt, se bilag afsnit 9.8.

12.1 Produkt

Der er bygget og opsat et cluster af 20 Raspberry Pi's, én master PC, én switch og én router, se figur 41 og 42.

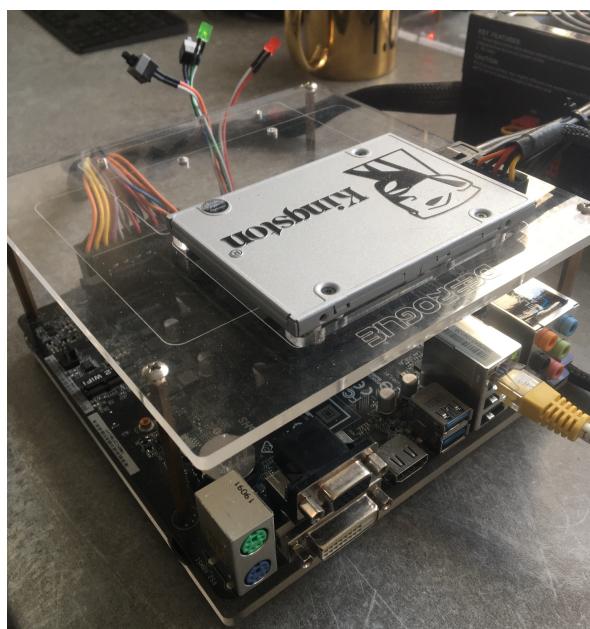


(a)



(b)

Figure 41: Raspberry Pi's i clusteret.



(a)



(b)

Figure 42: Master PC, switch og router i clusteret.

Det er lykkedes at opbygge clusteret som et cloudsysten orkestreret med Kubernetes, se figur 43.

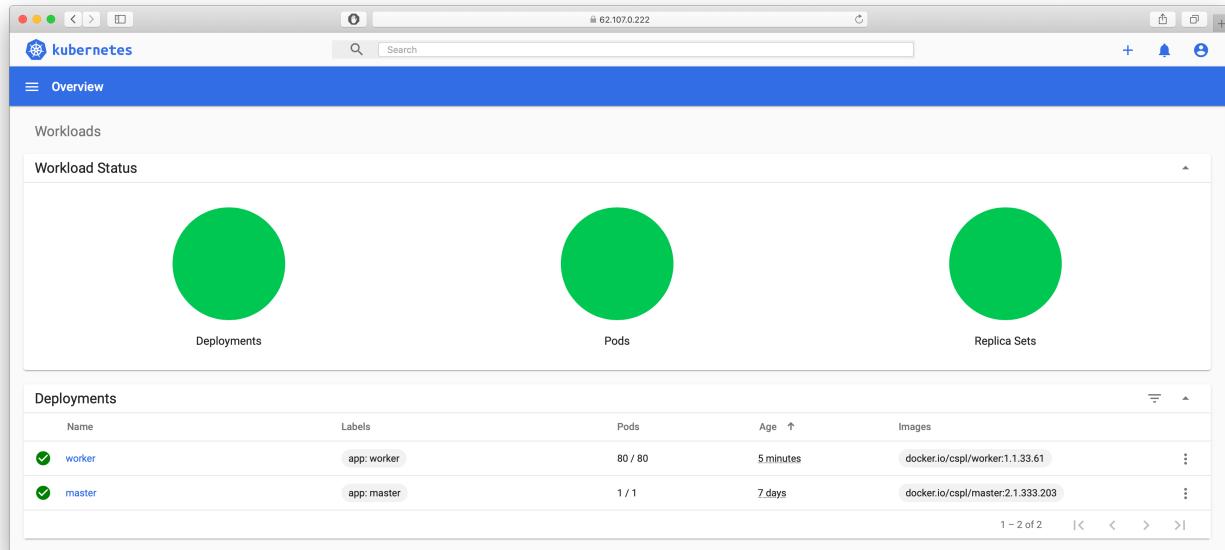


Figure 43: Kubernetes DashBoard

Brugere har mulighed for at parallelisere og fjerneksekvere kode i clusteret vha. en offentlig NuGet-pakke, som giver adgang til systemet, se figur 44.



Figure 44: ClusterSupportedParallelLibrary på NuGet.org.

NuGet-pakken indeholder IClusterParallelizer-interfacet som giver brugeren mulighed for at implementere kode, som kan eksekveres i clusteret, se figur 45. NuGet-pakken og interfacet giver bruger mulighed for en 'plug and play'-løsning til at få sit eget program til at eksekvere hurtigere vha. cloud computing.

Der er lavet brugerundersøgelse hvor en softwareudviklerstuderende har prøvet at benytte systemet. Der var både positiv og konstruktiv respons, se bilag afsnit 9.3.

```

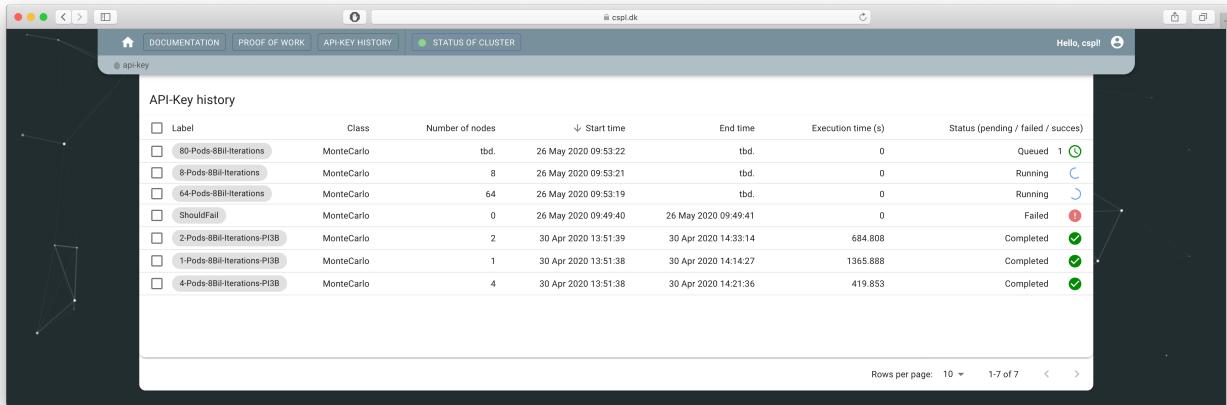
IClusterParallelizer.cs

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 45: Generisk IClusterParallelizer interface.

Der er med henblik på kommercialisering af produktet, implementeret logik til håndtering af API-nøgler og i den forbindelse monitorering af brugerens forbrug af systemet. På hjemmesiden er det muligt at anskaffe en API-nøgle samt tilkøbe adgang til systemet. Hjemmesiden viser brugerens information om de jobs han/hun har kørende eller har haft kørende i clusteret. Som det ses på figur 46 viser hjemmesiden label, klassenavnet, antallet af pods, starttid, sluttid, eksekveringstid og status.



Label	Class	Number of nodes	Start time	End time	Execution time (s)	Status (pending / failed / success)
80-Pods-8Bit-Iterations	MonteCarlo	tbd.	26 May 2020 09:53:22	tbd.	0	Queued 1 (green)
8-Pods-8Bit-Iterations	MonteCarlo	8	26 May 2020 09:53:21	tbd.	0	Running (blue)
64-Pods-8Bit-Iterations	MonteCarlo	64	26 May 2020 09:53:19	tbd.	0	Running (blue)
ShouldFail	MonteCarlo	0	26 May 2020 09:49:40	26 May 2020 09:49:41	0	Failed (red)
2-Pods-8Bit-Iterations-PI3B	MonteCarlo	2	30 Apr 2020 13:51:39	30 Apr 2020 14:33:14	684.808	Completed (green)
1-Pods-8Bit-Iterations-PI3B	MonteCarlo	1	30 Apr 2020 13:51:38	30 Apr 2020 14:14:27	1365.888	Completed (green)
4-Pods-8Bit-Iterations-PI3B	MonteCarlo	4	30 Apr 2020 13:51:38	30 Apr 2020 14:21:36	419.653	Completed (green)

Figure 46: Hjemmesidens API-key history view.

12.2 Performance

Performancetests fra afsnit 9.6 har vist at der for nogle algoritmer, kan være en performancemæssig gevinst for en bruger, ved at benytte systemet til at eksekvere et program hurtigere i et Raspberry Pi cluster end på en laptop. Det har også vist sig at nogle typer algoritmer ikke har givet en performancemæssig gevinst ved at benytte systemet, specielt ift. når der skal sendes og processeres store datasæt. Nogle algoritmer har slet ikke været mulige at eksekvere i clusteret.

12.3 Proces

Udviklingsprocessen er blevet styret med den agile metode Scrum som værktøj. Der er holdt daglige stand-up møder som har givet gruppen indblik i hinandens arbejde, se figur 47.

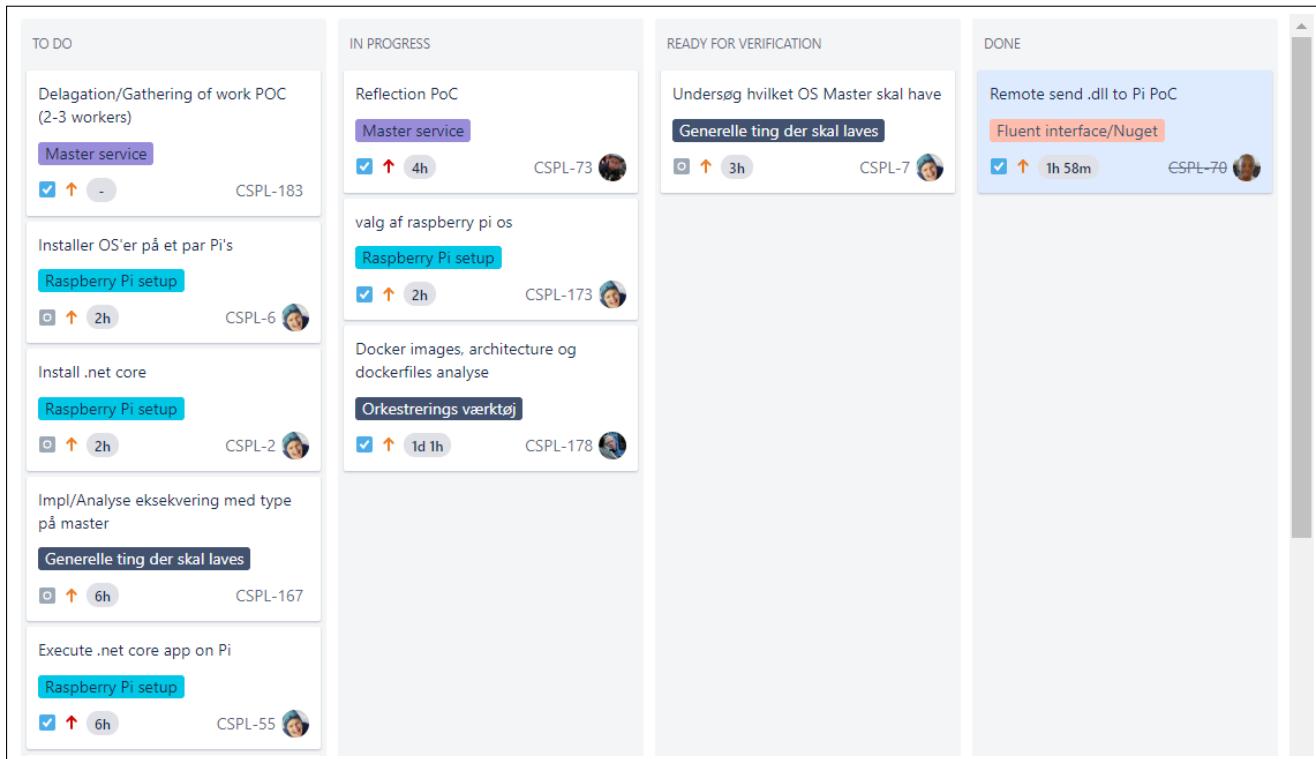


Figure 47: Jira scrumboard, aktivt sprint.

Der er anvendt continuous integration under hele udviklingsprocessen og i den forbindelse er det opsat en TeamCity byggeserver, se figur 48.

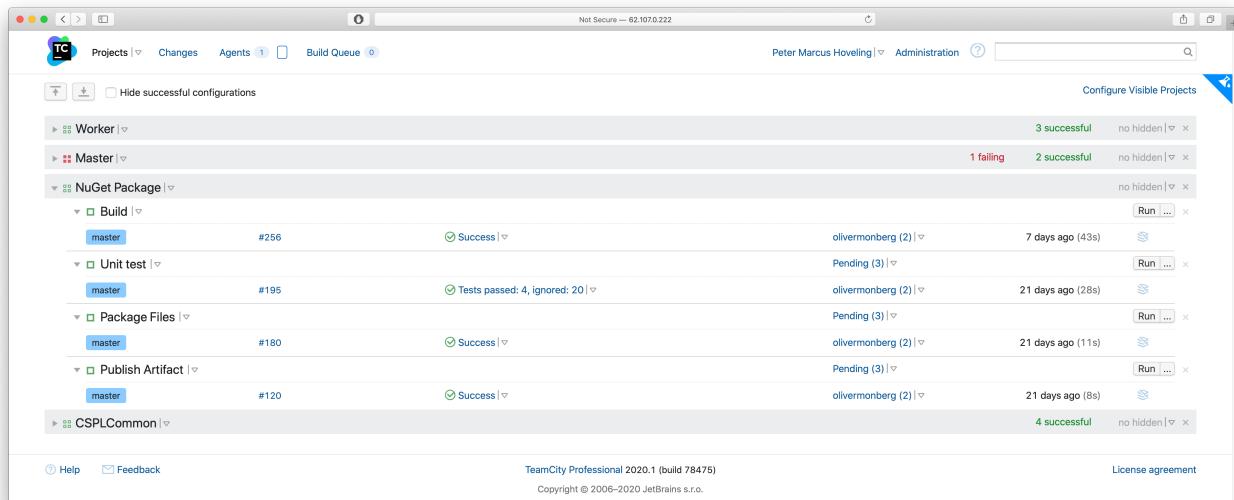


Figure 48: TeamCity pipelines.

Der er lavet unittests og integrationtests af systemet, automatisk versionering af NuGet-pakker og Docker images og deployment af disse. Nedenstående tabel viser statistik fra byggeserveren.

• Worker API

- 80 builds
- 68 gange er alle tests kørt

- 61 Docker Hub publiseringer

- **Master API**

- 297 builds
- 268 gange er alle tests kørt
- 203 Docker Hub publiseringer

- **ClusterSupportedParallelLibrary NuGet-pakken**

- 265 builds
- 204 gange er alle tests kørt
- 61 NuGet-server publiseringer

- **CSPL.Common**

- 237 builds
- 179 gange er alle tests kørt
- 132 NuGet-server publiseringer

13 Diskussion

13.1 Produkt

13.1.1 CPU overlap

Der er ikke sat nogle restriktioner for, om en klient må oprette tråde/tasks i IClusterParalellizer.Parrallel. Det kan give problemer i og med at der i konfigurationen af worker-pods i Kubernetes heller ikke er sat nogen begrænsninger for, hvilke fysiske kerner der på noden bliver benyttet til at processere tråde. Én pod kan altså benytte alle kerner på en node. En klient kan dermed overlape/’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. Dette problem med CPU overlap kan forbedres ved at konfigurere CPU ressourcer for worker-pods i Kubernetes [11].

13.1.2 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 internettrafik i clusteret.

13.1.3 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. I og med at master API’et ikke kan skaleres, er dette også en flaskehals for skalering af worker-pods, da flere worker-pods vil kræve mere af master API’et.

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 refaktureres ud til at være selvstændige services.

13.1.4 Master svagheder

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 IClusterParalellizer.Split.

Manuel slowdown af master

Hvis en klient starter en masse krævende tråde i IClusterParalellizer.Split som aldrig terminere, 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.

Mulig løsning

For at modarbejde at klienter kan kalde malicious kode eller kode som forurener master containeren ved f.eks. leaking af filer eller memory, bør klientens kode ikke blive kaldt i master API'et. En eventuel løsning kunne være at refakturere master API'et, således at klientens Split og Collect-metoder bliver eksekveret i eksterne containerized services som kan nedlægges efter brug.

13.2 Performance

13.2.1 Overordnet performance

Med udgangspunkt i de performancetests som er blevet kørt i systemet, se afsnit 9.6, har det vist sig at 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, tilnærmedesvis 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 systemet.

Performancetests fra afsnit 9.6 har også vist at der er algoritmer som ikke performer godt i systemet. Heriblandt en billedekomprimerings-algoritme, som havde et for stort overhead ved at sende datasæt ift. processeringstiden. Derudover har hver worker-pod maksimalt kunne procesere 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 procesere dataen.

På baggrund af dette har det vist sig, at produktet har en bestemt målgruppe ift. hvilke programmer/algoritmer en softwareudvikler ønsker at optimere vha. systemet. Det er kun i bestemte use cases, at det vil være en fordel at benytte systemet og i nogle tilfælde vil systemet slet ikke kunne benyttes.

13.2.2 Datasæt sendes som JSON

Det datasæt som klienter sender til clusteret bliver serialiseres til JSON-tekst. Dette er ikke optimalt da datasætene kan ende med at fyldes 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 [30].

13.2.3 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 Core er en grænse for hvor meget objekter må fylde, er der altså en teknisk grænse for, hvor meget data der kan sendes på 32-bit systemer på 2GB [18]. 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 [20]. 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.

13.2.4 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.

13.2.5 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.

14 Konklusion

Dette bachelor projekt har haft hensigt på at undersøge hvorledes det er muligt og om det giver mening at udvikle et cluster af Raspberry Pi's til parallelisering af kode. Målet har været at stå tilbage med et produkt som kunne bruges af andre udviklere. Igennem projektet er der benyttet metoder og processer for at fremme udviklingen og afslutningsvist produktet.

14.1 Produkt

Det er lykkes at lave et plugin til softwareudviklere, som giver nem adgang til cloud computing og fjerne-sekvering af kode på et Raspberry Pi cluster. Det er vist at nogle implementeringer af algoritmer/programmer vha. NuGet-pakkens generiske interface kan få en performancemæssig gevinst. Det er dog også vist at systemet ikke i sig selv garanterer en performancemæssig gevinst. Det afhænger i høj grad af brugerens domæne og implementering af det generiske interface.

Cluster

Der er lavet en prototype på et system, som kan skaleres og benyttes af multible brugere med henblik på kommercialisering. Det at opsætte clusteret i Kubernetes har givet problemer ift. effektivisering og implementering. Til gengæld har det gjort, at målet om at kunne skalere horisontalt og dermed kommercialisere produktet, er mere opnåeligt. Skalerbarheden kan være med til at sikre produktets fremtid.

Hjemmeside

Den udviklede hjemmeside gør, at produktet er mere klar til en evt. kommercialisering. Den giver en bruger mulighed for at administrere sit forbrug af clusteret. Det at brugeren kan observere sine jobs, imens de kører, giver brugeren en følelse af responsivitet og service.

Kommercialisering

Generelt er produktet ikke klar til kommercialisering. Det vil sætte højere krav til f.eks. sikkerhed og misbrug af systemet. Derudover er forretningsmodellen ikke fyldestgørende nok ift. målgruppe, markedsføring, økonomi og prissætning af servicen. Det ville kræve større viden indenfor forretningsdrift, end hvad gruppen besidder.

14.2 Proces

Risikovurdering & PoC's

Gruppens forståelse for vigtigheden om at lave en risiko matrix og dermed tidligt imødegå de mest kritiske punkter i projektet, har forøget sandsynligheden for at projektet er lykkes. Gruppen har effektivt adresseret de mest kritiske punkter ved at lave PoCs på små dele/koncepter af systemet.

Scrum

Brugen af Scrum som processstyringsværktøj har fungeret godt i projektgruppen. Det har givet struktur til processen og mulighed for at lave iterationer over projektets features. Grunden til at det har virket godt, er bla. at det har givet et overblik over hvilke opgaver, hvert gruppemedlem arbejder med. Samt hvilke opgaver der skulle prioriteres og hvilke mål der skulle være for det næste sprint. Oprettelse af epics og tasks har givet god anledning til diskussion i projektgruppen om produktet og systemet.

Continuous integration

Brugen af continuous integration fra starten af projektforløbet har fungeret rigtig godt. Automatisk byg, versionering og publisering af NuGet-pakker og Docker images, har effektiviseret processen/udviklingen i form af tidsbesparelser. Automatiserede unittests og integrationstests har løbende sikret systemets kvalitet under refakturering og tilføjelse af funktionaliteter.

15 Fremtidigt arbejde

15.1 Forretningsmodel

I fremtiden ville det være nødvendigt at modifcere forretningsmodellen, således at den fokuserer mere på økonomi og prissætning af servicen. Der skulle laves markedsanalyse for at finde potentielle muligheder for brugen af system. Derudover kan man overveje at ændre på salgs modellen, således at man sælger CSPL som et fuldt system snarere end en service. Hertil kunne man lave et admin-panel som skræddersyes til den enkelte virksomhed. Så kan virksomheder selv bestemme hardware, skalering og selv administrere clusteret.

15.2 Cloud-løsning

Man kunne undersøge mulighederne for at deployere CSPL til skyen. Dette ville gøre at clusteret ikke behøver at stå samlet som et fysisk cluster. Man skulle dertil også undersøge om man stadig vil kunne se et gain i performance, da kommunikationshastighederne i det interne system ville blive højere. Det ville være nemmere at skalere clusteret op og ned, ved at "leje" hardware/noder til clusteret.

15.3 Systemet som simuleringsværkøj

I og med at klienter kan implementere generisk kode og distribuere dette til x-antal workers i systemet, kan systemet også benyttes i andre sammenhænge end med henblik på hurtigere eksekvering af en klientens program. Det kunne være en mulighed at modifcere systemet således at softwareudviklere kan benytte det til simuleringer. F.eks. hvis der skal laves stressstests af headendsystemer som håndterer data og kommunikation fra x-antal IoT-devices. Det skulle i så fald være muligt for softwareudvikleren nemt at implementere noget kode, som simulerer disse devices og derefter få det distribueret ud i clusteret.

16 Litteraturliste

Hjemmesider

- [1] Academo.org. *Estimating Pi using the Monte Carlo Method*. URL: <https://academo.org/demos/estimating-pi-monte-carlo/>.
- [2] Simon Brown. *The C4 model for visualising software architecture*. URL: <https://c4model.com>.
- [3] Rosetta Code. *Parallel Brute Force*. URL: https://rosettacode.org/wiki/Parallel_Brute_Force.
- [4] CodeProject. *Compiling C# Code at Runtime*. URL: <https://www.codeproject.com/Tips/715891/Compiling-Csharp-Code-at-Runtime>.
- [5] Google developer. *Firebase Hosting*. URL: <https://firebase.google.com/docs/hosting>.
- [6] Google Developers. *Pricing plans*. URL: <https://firebase.google.com/pricing>.
- [7] Firebase. *Cloud Firestore*. URL: <https://firebase.google.com/docs/firestore>.
- [8] Firebase. *View changes between snapshots*. URL: https://firebase.google.com/docs/firestore/query-data/listen#view_changes_between_snapshots.
- [9] Paulo Gomes. *Unit Testing and the Arrange, Act and Assert (AAA) Pattern*. URL: <https://medium.com/@pjbgf/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80>.
- [10] Josh Kiepert. *Creating a Raspberry Pi-Based Beowulf Cluster*. URL: [https://bitbucket.org/jkiepert/rpicluster/src/master/docs/Creating.a.Raspberry.Pi-Based.Beowulf.Cluster\[BSU\].pdf](https://bitbucket.org/jkiepert/rpicluster/src/master/docs/Creating.a.Raspberry.Pi-Based.Beowulf.Cluster[BSU].pdf).
- [11] Kubernetes. *Assign CPU Resources to Containers and Pods*. URL: <https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/>.
- [12] Kubernetes. *Cluster Networking*. URL: <https://kubernetes.io/docs/concepts/cluster-administration/networking/>.
- [13] Kubernetes. *kube-proxy*. URL: <https://kubernetes.io/docs/docs/reference/command-line-tools-reference/kube-proxy/>.
- [14] Kubernetes. *Kubernetes Features*. URL: <https://kubernetes.io/>.
- [15] Kubernetes. *Kubernetes services*. URL: <https://kubernetes.io/docs/concepts/services-networking/service/>.
- [16] Kubernetes. *What is Kubernetes?* URL: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>.
- [17] Kubernetes. *What is kubernetes*. URL: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>.
- [18] Microsoft. *<gcAllowVeryLargeObjects> Element*. URL: <https://docs.microsoft.com/en-us/dotnet/framework/configure-apps/file-schema/runtime/gcallowverylargeobjects-element>.
- [19] Microsoft. *Unit testing C# with NUnit and .NET Core*. URL: <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit>.
- [20] Microsoft. *Using Streams on the Network*. URL: <https://docs.microsoft.com/en-us/dotnet/framework/network-programming/using-streams-on-the-network>.
- [21] Hossien Pishro Nik. *Introduction to Probability, Statistics, and Random Processes*. URL: https://www.probabilitycourse.com/chapter1/1_2_2_set_operations.php.
- [22] npmjs. *node.bcrypt.js*. URL: <https://www.npmjs.com/package/bcrypt>.
- [23] NSubstitute. *NSubstitute*. URL: <https://nsubstitute.github.io/>.
- [24] Seralo. *RaspberryPI models comparison*. URL: https://socialcompare.com/en/comparison/raspberrypi-models-comparison?fbclid=IwAR3xE7XpbmcSijxhITtZ2JpAglq_whCQ4f510bGsN7Oo-sE4bQx3EfWkcf0.
- [25] Mountain Goat Software. *User stories*. URL: <https://www.mountaingoatsoftware.com/agile/user-stories>.
- [26] 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>.

- [27] Aarhus Universitet. *ITONK - Objektorienteret netværkskommunikation*. URL: <https://studerende.au.dk/studier/fagportaler/diplomingenioer/undervisningdiplom/valgfag-for-diplomingenioerer-katrinebjerg/valgfag-kun-foraar/itong-objektorienteret-netvaerkskommunikation/>.
- [28] Wikipedia. *Cloud computing*. URL: https://en.wikipedia.org/wiki/Cloud_computing.
- [29] Wikipedia. *FURPS*. URL: <https://en.wikipedia.org/wiki/FURPS>.
- [30] Wikipedia. *gzip*. URL: <https://en.wikipedia.org/wiki/Gzip>.
- [31] Wikipedia. *MoSCoW method*. URL: https://en.wikipedia.org/wiki/MoSCoW_method.
- [32] Wikipedia. *Proof of concept*. URL: https://en.wikipedia.org/wiki/Proof_of_concept.
- [33] Wikipedia. *WebSocket*. URL: <https://en.wikipedia.org/wiki/WebSocket>.

17 Bilagsoversigt

1 Forord

1.1 Ordliste

2 Kravspecifikation

2.1 Aktør kontekst diagram

2.2 FURPS

2.3 Funktionelle krav - MoSCoW analyse

2.4 Ikke-funktionelle krav

2.5 User stories

2.6 Forretningsmodel

2.6.1 Hvad er produktet egentligt?

2.6.2 Hvem er den primære kunde?

2.6.3 Hvilke problem løser produktet for kunden?

2.6.4 Hvad er værdien for kunden?

2.6.5 Hvem er konkurrenterne?

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

2.6.7 Hvilke indtægter har vi?

2.6.8 Hvordan fastsættes priserne?

3 Metode og Process

3.1 Indledning

3.2 Gruppeddannelsel

3.3 Samarbejds aftale

3.4 Projektledelse

3.5 Scrum

3.6 Udviklingsforløb

3.7 Risikovurdering

3.8 Møder

3.9 Planlægning

3.10 Arbejdsfordeling

3.11 Continuous Integration/DevOps

3.11.1 Opsætning af Byggeserver

3.11.2 Opsætning af pipelines

3.11.2.1 Versionering af komponenter

3.12 Udviklingsværktøjer og programmeringssprog

3.13 Coronavirus tilpasning

3.14 Mødeindkaldelser og mødereferater

4 Analyse

4.1 Cluster hardware

4.2 Jenkins vs. TeamCity

4.3 Master OS

4.3.1 Internetdeling og pålidelighed

4.3.2 Kubernetes og Docker på Linux vs. Windows

4.4 Raspberry Pi OS

4.5 Orkestrering

4.5.1 Kubernetes som orkestreringsværktøj

4.6 Docker

4.6.1 Docker Engine

4.6.2 Docker Swarm (vs Kubernetes)

4.6.3 Dockerfiler og images

4.7 Valg af programmeringssprog & frameworks

4.8 .NET Core i clusteret

4.9 Én vs fire pods pr. Raspberry Pi

4.10 Fjerneksekvering af kode

4.10.1 Reflection

4.11 Distribuering af arbejde til workers

4.11.1 Undersøgelse af kubernetes loadbalancing

4.12 Firebase

5 Arkitektur

5.1 Introduktion

5.2 Systemsekvensdiagram

5.3 Kontekstdiagram

5.4 Containerdiagram

5.5 Komponent- og kodediagrammer

5.5.1 CSPL NuGet-pakken

5.5.1.1 Komponentdiagram

5.5.1.2 Kodediagrammer

5.5.2 Master API

5.5.2.1 Komponentdiagram

5.5.2.2 Kodediagrammer

5.5.3 Worker API

5.5.3.1 Komponentdiagram

5.5.3.2 Kodediagrammer

5.5.4 CSPL.Common

5.5.4.1 Komponentdiagram

5.5.4.2 Kodediagrammer

5.5.5 Hjemmesiden

5.5.5.1 Home

5.5.5.2 Firebase

5.5.5.3 Dokumentation

5.5.5.4 Login og create account

5.5.5.5 Account Management

5.5.5.6 API-Key History

5.6 Hosting af applikationer

5.6.1 Hjemmesiden

5.6.2 CSPL NuGet-pakken

5.6.3 Master API

5.6.4 Worker API

5.7 Kommunikation mellem frontend og Master

5.7.1 Hardware oversigt

6 Design

6.1 Load balancing

6.1.1 Løsning 1 - Label switching

6.1.2 Løsning 2 - Custom load balancer

6.1.2.1 Worker pod's livscyklus i Kubernetes

6.1.2.2 Custom Loadbalancer Cache

6.1.2.3 Håndterering af døde worker-pods

6.1.2.4 Håndterering af nye worker-pods

6.1.2.5 Valgte løsning

6.2 Forurening af pods i clusteret

6.3 Køstruktur

6.3.1 Hvornår får klienter tildelt workers?

6.3.2 Thread safety

6.3.3 Køstatus

6.3.4 Hvornår er det ens tur?

6.3.5 QueueUpdater

6.4 NuGet-pakke

6.5 Compile time properties problem

7 Implementering

7.1 Sekvensdiagrammer

7.1.1 NuGet-pakken

7.1.2 Master API

7.1.3 Worker API

7.2 Delt NuGet-pakke

7.3 DLL dependencies og filter

7.4 DLL-pakke

7.4.1 Sending af DLL-pakke

7.4.2 Eksekvering af DLL-pakke

- 7.5 Opsætning og problemer med Kubernetes cluster
 - 7.5.1 Kubeadm
 - 7.5.2 Pod network
 - 7.5.3 Dynamiske IP-adresser
 - 7.6 Deployment og konfigurering
 - 7.7 Forurening af pods i clusteret
 - 7.7.1 CustomAssemblyLoadContext
 - 7.7.2 Oprydning af worker-pods
 - 7.8 SSL
 - 7.8.1 TLS tradeoff
 - 7.8.2 TLS hjemmeside, master og workers
 - 7.8.3 Opsætning
 - 7.8.3.1 Hjemmeside
 - 7.8.3.2 Master & Workers
 - 7.9 API-Key history view
 - 7.10 Firebase abstraktion
 - 7.11 Autorisation
 - 7.12 Portforwarding
 - 7.13 Fejlhåndtering
 - 7.13.1 Generel fejlhåndtering
 - 7.13.2 Klient-exceptions
-
- ## 8 Kubernetes installation and opsætningsguide
- 8.1 Prerequisite
 - 8.2 Install OS
 - 8.3 Enable SSH
 - 8.4 Disable firewall
 - 8.5 Edit host name
 - 8.6 Configure cgroup boot options
 - 8.7 Install updates
 - 8.8 Permanently disable swap
 - 8.9 Install Docker
 - 8.10 Configure Docker daemon options
- 8.11 Setup Kubernetes repository
 - 8.12 Install Kubernetes.
 - 8.13 Setup Kubernetes.
 - 8.14 Setup pod network.
 - 8.15 Join worker nodes to the Kubernetes cluster
 - 8.16 Cluster SSH
-
- ## 9 Test
- 9.1 Introduktion
 - 9.2 ManuelTest
 - 9.2.1 Manuel test - Adminstrative produkt
 - 9.2.2 Monkey tests
 - 9.3 Feedback
 - 9.4 Modultest
 - 9.4.1 Master
 - 9.4.1.1 LoadBalancer
 - 9.4.1.2 KubernetesFecether
 - 9.4.1.3 UserValidator
 - 9.4.1.4 ClusterQueue
 - 9.4.1.5 WorkDelegator
 - 9.4.2 ClusterSupportedParallelLibrary (NuGet-pakke)
 - 9.4.2.1 ReferencedAssembliesFilter
 - 9.4.3 CSPL.Common (NuGet-pakke)
 - 9.4.3.1 TypeResolver
 - 9.5 Integrationtest
 - 9.5.1 MonteCarlo
 - 9.5.2 Brute force hash
 - 9.5.3 Fejlhåndtering af klient-exceptions
 - 9.6 Performancetest
 - 9.6.1 Raspberry Pi 3B vs. Raspberry Pi 3B+
 - 9.6.2 Docker overhead vs. normal proces
 - 9.6.3 CSPL with Kubernetes vs. Docker container overhead
 - 9.6.4 Opsummering - hosting overhead i CSPL systemtet

- 9.6.5 Desktop vs. Laptop vs. CSPL
 - 9.6.5.1 MonteCarlo Pi approksimation
 - 9.6.5.2 Brute force SHA256 hash
- 9.6.6 Data sending overhead
- 9.6.7 HTTP vs. HTTPS
- 9.6.8 Billedekomprimering
- 9.6.9 MapReduce
- 9.6.10 Performance diskussion
- 9.7 Accepttestspezifikation
 - 9.7.1 Download NuGet-pakke
 - 9.7.2 Implementere NuGet-pakkens interface
 - 9.7.3 Start et job fra Nuget-pakket
 - 9.7.4 Cluster status
 - 9.7.5 Opret en bruger på hjemmesiden
 - 9.7.6 Login på hjemmesiden
 - 9.7.7 Log ud af hjemmesiden
 - 9.7.8 Ændrer password
 - 9.7.9 Tilsending af nyt password
 - 9.7.10 Slet account på hjemmesiden
 - 9.7.11 Se dokumentationen
 - 9.7.12 Se krav for brug af NuGet-pakken
 - 9.7.13 Se eksempler for brug af NuGet-pakken
 - 9.7.14 Se proof of work
 - 9.7.15 Få genereret en API-nøgle
 - 9.7.16 Se hvor mange kald der er tilbage på API-nøglen
 - 9.7.17 Se tabel over jobs som er associeret med specifik API-nøgle
- 9.8 Accepttest
 - 9.8.1 Funktionelle krav
 - 9.8.2 Ikke-funktionelle krav

10 Resultater

- 10.1 Viden om cloud computing
- 10.2 Raspberry pi Cluster

- 10.3 Scrum
- 10.4 NuGet-pakke
- 10.5 Kommercialisering
- 10.6 Performance
- 10.7 Accepttest
- 10.8 DockerHub

11 Diskussion

- 11.1 Fejl og fremtidige forbedringer
 - 11.1.1 CPU overlap
 - 11.1.2 Manuel crashing af master
 - 11.1.3 Manuel slowdown af master
 - 11.1.4 Ulovigheder/sikkerhed
 - 11.1.5 Datasæt sendes som JSON
 - 11.1.6 Streaming af klient-data
 - 11.1.7 Compile properties
 - 11.1.8 DLLPackage på completime
 - 11.1.9 DLL Cache
 - 11.1.10 Stateful master
- 11.2 Skalering af CSPL

12 Logbog

13 Litteraturliste