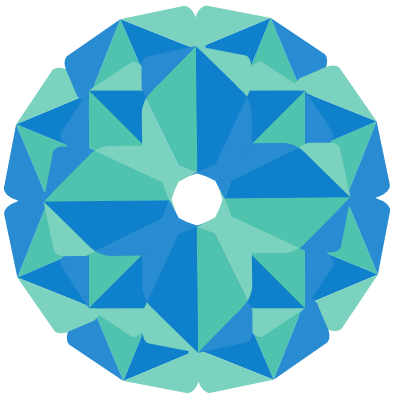
Aalborg Universitet
Institut for Datalogi



Distribueret Dart

Brian Astrup Mikkelsen
&
Jacob Bang

Foråret 2012

Aalborg Universitet

Institut for Datalogi
Selma Lagerlöfs Vej 300
9220 Aalborg Ø
Telefon 99 40 99 40
http://www.cs.aau.dk

Titel:
Distribueret Dart

Tema:
Programmerings Teknologi

Semester:
Software Engineering
10. semester
1. Februar - 12. Juni 2013

Gruppe:
sw108f13

Gruppe Medlemmer:
Brian Astrup Mikkelsen
Jacob Bang

{bmikke05, jbang08}
@student.aau.dk

Vejleder:
Bent Thomsen

Oplæg: 4

Sidetal: 73

Abstract:

This report is a master thesis, which is based upon a study performed on the previous semester.
This report documents the development of a library used to enable better support for distributed programming in the Dart programming language.
The report poses three questions: Is it possible to develop a library to support distributed programming in the Dart programming language? Is it possible to use the same Dart semantics as those used for concurrent programming? What are the limitations of imposing Dart concurrent semantics on the library? Furthermore, the problem statement required that the implementation must be done entirely in Dart, without making any modifications to the Virtual Machine.
Then follows a walkthrough of central elements of the Dart programming language, and an analysis and an implementation part.
It is concluded that although the library is not in a working state, it is not impossible to support distributed programming with a library, and also, that it is possible to do so using the semantics of darts concurrency model. It is found that the limitation of not having any way to tell whether an isolate is alive or dead, is more problematic in a network context than in a local context. a
The rest of this report is written in danish.

Forord

Denne rapport dokumenterer 10. semester projektet på kandidatuddannelsen for software udarbejdet af gruppe sw108f13 fra Institut for Datalogi ved Aalborg Universitet. Projektet er blevet udarbejdet i perioden fra 1. Februar til 12. Juni 2013, og dokumenterer udviklingen af et programbibliotek til programmeringssproget Dart, der gør det anvendeligt til distribueret programmering.

Programbiblioteket der er udviklet under projektet er navngivet "Distributed Dart", og benyttes under navnet *distributed_dart* ved almindelig brug og her i rapporten. Til programbiblioteket er der desuden udarbejdet et logo der er repræsenteret i figur 1.

Figur 1: *Logo for programbiblioteket der er udarbejdet i projektet, og består af en blomst med tilhørende navn for programbiblioteket. Blomsten vil blive benyttet i enkelte figurer i rapporten frem for det fulde navn på programbiblioteket.*

Dart er stadig under udvikling, hvilket har den konsekvens at ikke alle sprogkonstruktioner der benyttes i *distributed_dart* kan garanteres at virke med seneste version af Dart. Ligeledes udvikles der stadig aktivt på Dart API'et, der består af en række standard programbiblioteker, og disse ændringer er sjældent kompatibel med eksisterende kode, der gør brug af disse. Under projekts udarbejdelse, er koden løbende blevet vedligeholdt med seneste ændringer indenfor Dart projektet. Ligeledes vil kildekoden blive vedligeholdt i perioden mellem afleveringen af rapporten, og frem til eksaminationen således projektet så vidt muligt at kompatibelt med seneste version af Dart.

Kildekoden til *distributed_dart* er frigivet under en simplificeret BSD-licens og seneste version er tilgængelig igennem GitHub:
`https://github.com/SW108F13-AAU/distributed_dart`

Dokumentationen holdes løbende opdateret og kan findes på følgende URL:
`http://dart.dyndns.dk/`

Vi vil gerne takke vores vejleder Bent Thomsen for god vejledning og samarbejde under hele specialeforløbet, samt de mange interessante historier om forskellige programmeringssprog.

Brug af Fagtermer

I rapporten benyttes der engelske fagtermer hvor det vurderes at brugen af den engelske term er mere anvendt på dansk end en eventuel tilsvarende dansk udgave af udtrykket (f.eks. firewall frem for brandmur).

Indhold

Kapitel 1

Introduktion

Trenden for web udvikling, de seneste par år er at skrive, såkaldt rige web applikationer, som er istand til at konkurrere både på feature og ydelse, med klassiske skrivebords applikationer. For disse rige web applikationer gælder det at bruge grænsefladen skrives i JavaScript, og denne kommunikerer med en server applikation via HTTP protokollen. Server applikationen kan være skrevet i et vilkårligt sprog, så længe at den kan tilbyde en HTTP grænseflade som klient siden kan kommunikere med. Af populære server side teknologier, kan nævnes: PHP, ASP.NET og Java. Endvidere har er det for nyligt blevet muligt at benytte JavaScript på server siden også, ved hjælp af Node.Js frameworket, som benytter en tilpasning af V8 JavaScript motoren, til at afvikle JavaScript på serveren.

Dart er et nyt programmeringssprog som er målrettet mod udvikling af komplekse rige web applikationer. Dart er ikke det eneste programmeringssprog som har til formål at give et alternativ til JavaScript på klient siden. Sprog som Typescript og CoffeeScript er eksempler på alternative sprog, og Google Web Toolkit (GWT) er et eksempel på et grafisk toolkit til Java.

Fælles for de nævnte alternativer, er at de alle skal oversættes til JavaScript, før de kan benyttes i en webbrowser. Dart projektet indeholder også en oversætter som kan producere JavaScript, men målet er at Dart skal eksekveres direkte i browseren, af en Dart virtuel maskine. På skrivende tidspunkt eksisterer der en udgave af browseren chromium (dartium) som har implementeret Dart understøttelse, og da Dart VM'en er udgivet som open source software under en BSD licens, er der teknisk set ikke nogen hindring for at øvrige browser producenter kan implementere Dart understøttelse, om end, der kan være forretningsmæssige forhold som kan vise sig som større forhindringer.

Ligeledes kan Dart's VM også benyttes til at afvikle Dart kode på en server, og det gør det således muligt at skrive både klient og server logikken i samme sprog. I [12] argumenterer Thamsen et al. at webudvikling i et enkelt sprog er fordelagtigt, ifht. at dele viden omkring kodebasen imellem frontend og

backend udviklings hold. Som eksempel gives, at pga. de store forskelle i programmerings model og sprog, på hhv. frontend og backend, vil de implicerede udviklere ikke nødvendigvis besidde de fornødne kompetencer til at kunne sætte sig ind i koden på det andet hold.

Kapitel 2

Problemformulering

I rapporten [1] udarbejdet under forundersøgelsen blev programmeringssproget Dart undersøgt overordnet med det formål at skabe et overblik over sproget. Udover et overblik over sproget blev flere interessante detaljer i sproget udpeget som blev undersøgt i nærmere detaljer.

Følgende blev der specielt bemærket under det forrige projekt vedrørende isolates og hvilke muligheder der er for at oprette disse på nuværende tidspunkt:

"In the current stable branch, Isolates can be spawned, from either a top-level function, or from a URI. If spawned from a URI, the code is copied to, and evaluated on the calling side. Which means, there is currently no built-in support for spawning distributed Isolates." - Evaluating the Dart Programming Language.[1, s.22-23]

Ud fra denne mangel i Dart sproget er dette projekt inspireret til at gå i yderligere detaljer omkring hvilke muligheder der er for at tilføje denne funktionalitet til sproget samt hvilke begræsninger der vil være. Hertil er der blevet udarbejdet følgende spørgsmål som projektet har som formål at besvare som en del af problemformuleringen:

- Kan det lade sig gøre, at udvikle et programbibliotek til sproget Dart, der understøtter en distribueret programmeringsmodel, uden at ændre i Dart VM?

- Er det muligt at overholde semantikken for den måde Dart i forvejen understøtter flertrådet programmering?

- Hvilke begrænsninger opstår, når Dart semantikken for flertrådet programmering, skal overholdes?

2.1 Afgrænsninger

Grundet projektets omfang er der foretaget en følgende begrænsninger.

2.1.1 Implementering lavet i Dart

Implementeringen af programbiblioteket i dette projekt er udelukkende skrevet i Dart. Det er muligt at udvide Dart ved at skrive udvidelser i C hvilket vil kunne indlæses dynamisk af Dart VM'en. API'et for disse udvidelser er meget omfangsrigt og giver en række avanceret muligheder som der ellers ikke gives adgang til igennem det officielle Dart API.

Begrundelsen for dette valg er et ønske om at undersøge hvor langt det er muligt at komme udelukkende ved at benytte det API som Dart i dag stiller til rådighed. De steder det har været nødvendigt at foretage en begrænsning pga. manglende muligheder i Dart API'et er der beskrevet hvorledes det ville være muligt at løse begrænsningen ved brug af det mere avanceret Dart C API.

2.1.2 Begrænset fokus på sikkerhed over netværk

Kommunikation mellem enheder igennem et netværk er et stort område der indeholder en lang række problemstillinger. En af de største problemstillinger er her hvorledes det er muligt at lave en sikker forbindelse der ikke kan aflyttes, manipuleres eller på anden måde misbruges.

Grundet omfanget af dette emne er der ikke fokuseret på dette under udviklingen af dette projekt men derimod at forbindelsen mellem enhederne er sikker.

Der vil fortsat være fokus på opretholdelse af en stabil forbindelse samt hvilke udfordringer der er forbundet omkring håndtering af tab af netværksforbindelsen.

Kapitel 3

Overblik over Dart

Dette kapitel er et uddrag af kapitlet "Dart overview" fra rapporten udarbejdet under forundersøgelsen [1, kapitel 2]. Afsnit der ikke er fundet relevant for beskrivelsen af distributed_dart er fjernet og kapitlet er oversat således det passer med sproget i denne rapport.

Dette kapitel er et overblik over programmeringssproget Dart. Kapitlet vil fokusere på de basale elementer i sproget som f.eks. typesystemet, typer, funktioner, klasser samt brugen af programbiblioteker. Hvad angår operatorer og strukturen for programudførsel så er disse ikke en del af dette overblik fordi disse er meget ens med hvad andre mainstream sprog som Java, C# og lignende. Dette kapitel er baseret på kapitel 2 (*A Tour of the Dart Language*) fra bogen *Dart up and running* [3], hvor også de fleste af kodeeksemplerne nedstammer.

3.1 Typer

Typesystemet i Dart er en smule specielt ved sammenligning af mange andre sprog fordi typesystemet i Dart er valgfrit. Det betyder at udviklere selv kan bestemme hvis han ønsker at bruge statiske eller dynamiske typer og kan frit vælge mellem dem alt efter hvor det passer bedst i koden. Typerne har ikke nogen indflydelse under programudførsel fordi alle statiske typer fjernes og bliver ændret til dynamiske typer under programudførsel. Begrundelsen for dette designvalg er et ønske om at undgå at programudførsel ændres pga. de angivet typer. De statiske typer bruges således kun til at kunne hjælpe udvikleren igennem f.eks. et IDE og for at validere at interfaces bruges korrekt. Ved at angive statiske typer ved parametre til metoder er det nemmere for udviklere at vide hvordan disse metoder skal benyttes. Hvis typer derimod var dynamiske ville det være nødvendigt at forklare i dokumentationen hvilken type indput en given metode accepterer. Et eksempel på dette kan ses i kodeeksempel 3.1 hvor der er specificeret typer for parametre samt

for returværdien af metoden *sayHello*. Typen *var* angiver at typen ikke er specificeret og bruges ved variablen *hello* i eksemplet.

```
1 String sayHello(String text) {
2     var hello = "Hello $text";
3     return hello;
4 }
```

Kodeeksempel 3.1: *Eksempel på brug af statiske og dynamiske typer i Dart.*

Et Dart program kan køre i to forskellige tilstande: *production* og *checked*. I tilstanden *checked* benytter Dart VM'en alle defineret typer til at tjekke løbende om disse overholdes under kørsel af programmet hvorimod tilstanden *production* fjerner alle typer under kørsel. Begrundelsen for ikke altid at køre i tilstanden *production* er pga. tab i ydeevne pga. de mange tjek af typer og det anbefales derfor kun at bruge *checked* tilstanden ved fejlfinding.

Dart består af følgende indbygget typer:

- num (integers, double)
- String
- bool
- List
- Map

Foruden de indbygget typer er det også muligt at definere egne typer i form af klasser. I Dart er der to forskellige forskellige nøgleord til at angive en konstant variabel: *final* og *const*. Forskellen på disse er at ved *const* variabler skal disse kunne defineres ved programmet kører mens *final* variabler først er konstante når de får tildelt deres første værdi. *const* bruges derfor til at definere konstanter som f.eks. *const var pi = 3.14* mens *final* ofte bruges til at få tildelt værdier af *constructor* metoden i en klasse.

num

num-typen er speciel fordi den repræsenterer både heltal og decimaltal. Typen gør det herved muligt for en metode at angive at den tager imod et

hvilket som helst tal frem for kun en af typer *int* eller *double*. Heltal kan i Dart have en arbitrær størrelse mens *double*-typen er en decimaltalværdi specificeret ved IEEE 754 standarden og defineret til at være på 64-bit.

String

Tekststrenge kan i Dart defineres ved brug af enten enkelt eller dobbelt anførselstegn. Et eksempel på brugen af strenge i Dart kan ses i kodeeksempel 3.2[1]:

```
1 String a = "car";                          // "car"
2 String b = 'apple ';                       // "apple "
3 String c = b + a;                          // "apple car"
4 String d = "My best '$c'";                 // "My best 'apple car'"
5 String e = 'I have an ${b.concat(a)}';// "I have an apple car"
```

Kodeeksempel 3.2: *Eksempler på sammensætning af tekststrenge i Dart.*

Resultatet fra et *$expression* skal være af typen *String*. I tilfældet af anden type benyttes *toString()* metoden på objektet automatisk.

bool

Der er kun to objekter i Dart der har typen *bool*: *true* og *false*. Til forskel fra C og JavaScript så er det kun *true* der kan repræsentere værdien "sandt". Begrundelsen for dette valg er for at undgå forvirring når mere end en datatype kan repræsentere værdien "sandt". Et eksempel hvor dette er vigtigt kan ses i kodeeksempel 3.3 [13]:

```
1 if (1) {
2   print('JavaScript prints this line.');
3 } else {
4   print('Dart in production mode prints this line.');
5   // However, in checked mode, (1) throws an exception
6   // because 1 is not boolean in Dart.
7 }
```

[1]Denne del af rapporten afviger væsentlige fra den oprindelige beskrivelse fordi det nu er muligt i Dart at sammensætte strenge ved brug af +-operationen som vist i linje 3 i kodeeksempel 3.2. Samtidig er det ikke længere muligt at benytte *concat* metoden på strenge.

Kodeeksempel 3.3: *Eksempel på hvordan Dart behandler bool-værdier i forhold til JavaScript.*

Som kodeeksempel 3.3 viser er der stor forskel på hvordan Dart i *production*-tilstand fungerer i forhold til *checked*-tilstand. Det kan derfor generelt anbefales aldrig at bruge andet end *bool*-værdier som udtryk i f.eks. kontrolstrukturer.

List

I stedet for at implementere *arrays* har Dart en *List* klasse der kan sammenlignes med *ArrayList* klassen fra Java. *List* fungerer som *arrays* men kan i nogle tilfælde ændre størrelse som *ArrayList* hvis der tilføjes eller fjernes elementer som vist i kodeeksempel 3.4. Indekset for elementer starter ved 0 og går op til *list.length-1*.

```
1 var list = [1,2,3];
2 list.add("4");
3 list.removeAt(0);
4
5 for (var i = 0; i < list.length; i++) {
6     print(list[i]);
7 }
```

Kodeeksempel 3.4: *Eksempel på brug af en liste i Dart.*

Fordi Dart ikke har noget statisk typesystem er det muligt under kørsel at tilføje forskellige typer til den samme liste og efterfølgende gennemløbe elementerne.

Map

Den sidste indbygget type der vil blive gennemgået er *Map* der kan sammenlignes med *HashMap* klassen i Java. *Map* gør det muligt at sammenbinde en værdi til en nøgle og gemme dette som et par. Der stilles intet krav til værdien og kan således også bestå af værdien *null*. Der er to måder oprette en instans af klassen *Map*: indbygget sprogkonstruktion eller som en instans af klassen

Map. Ved brug af den indbygget sprogkonstruktion skal alle nøgleværdier være af typen *String* som i kodeeksempel 3.5:

```
var stuff = {
    'key 1'  : 'apple',
    'key 2'  : 5,
    'key 3'  : () => print("hello world")
};
```

Kodeeksempel 3.5: *Eksempel på opsætning af et Map objekt ved at brug af indbygget sprogkonstruktion i Dart. I eksemplet gemmes der en tekststreng, et heltal samt en anonym metode der ikke tager imod parametre.*

Den anden måde at oprette et *Map* objekt er ved oprettelse af et objekt ud fra *Map* klassen. Som standard genererer dette et tomt *Map* objekt hvor der senere kan tilføjes og fjernes relationer mellem nøgler og værdier. Et eksempel på dette kan ses i kodeeksempel 3.6 der desuden viser det er muligt at have forskellige typer nøgler og værdier i det samme *Map* objekt:

```
var things = new Map();
things[1] = "car";
things["2"] = 244;
things[0.1] = "wood";

things.forEach((key, value) => print("$key = $value"));
// 0.1 = wood
// 1 = car
// 2 = 244
```

Kodeeksempel 3.6: *Oprettelse af et tomt Map objekt hvor der efterfølgende indsættes et antal relationer. Tilsidst udskrives alle relationer med metoden .forEach() der tager imod en anonym funktion med to parametre.*

3.2 Klasser

Dart er et objektorienteret programmeringssprog med mulighed for at definere klasser og lave instanser af disse for at oprette objekter. Til forskel fra Java er det ikke et krav at alt kode er defineret inde i klasser. Derimod tillader Dart at definere globale funktioner og variabler ligesom C. Et eksempel på en klasse og en globalt defineret funktion kan ses i kodeeksempel 3.7. Klassen definerer to variabler, en *constructor* og en metode. *Constructor* metoden

benytter en kompakt sprogkonstruktion der gør det nemmere at definere variabler der er medlem af klassen uden at skulle skrive en decideret metode. Hvis en *constructor* ikke definerer nogen metode skal den ende med *;* som i linje 3 i kodeeksemplet. *toString()* metoden er defineret for alle objekter i Dart og kaldes automatisk af *print()* metoden ligesom det er tilfældet i Java når der forsøges at udskrives et objekt. Ved at overskrive denne metode er det muligt at styre hvorledes et objekt skal kunne repræsenteres som et *String* objekt.

```
1  class Point {
2      num x,y;
3      Point(this.x,this.y);
4
5      String toString() {
6          return "Point:($x,$y)";
7      }
8  }
9
10 int main(){
11     var p = new Point(2,2);
12     print(p); // Point:(2,2)
13 }
```

Kodeeksempel 3.7: *Eksempel på en Point klasse med tilhørende constructor.*

Ved klasser er det f.eks. muligt i Java at definere hvilket adgangsniveau (*scope*) klassens variabler og metoder skal være tilgængelig ved og er delt op i tre niveauer: *public*, *protected* og *private*. Adgangsniveauet sættes ved definitionen af af de enkelte variabler og metoder der er medlem af klassen. Dette gør det muligt at definere præcist hvad for nogle metoder og variabler der ønskes andre klasser og pakker har adgang til. I Dart er der ikke nogen nøgleord der definerer adgangsniveauet til klassers variabler og metoder. Derimod defineres adgangsniveauet ud fra navnet på klassen, metoden eller variablen således at hvis navnet begynder med "_" så er klassen, metoden eller variablen privat. Et eksempel på dette kan ses i kodeeksempel 3.8.

```
1  class _InternalStuff {
2   String _secretMethod() {
3      return "MyPassword";
4   }
5  }
6
7  int main(){
8   var object = new _InternalStuff();
9   print(object._secretMethod());
10 }
```

Kodeeksempel 3.8: *Eksempel på privat scope for en klasse samt metode der er medlem af klassen.*

En ting der er værd at bemærke i kodeeksempel 3.8 er *_InternalStuff* kun er privat i forhold til andre biblioteker (og derfor er det muligt i eksemplet at tilgå *_InternalStuff* og *_secretMethod*). Det gør at det er muligt at få adgang til private klasser, metoder og variabler så længde koden er i det samme bibliotek som klassen, metoden eller variablen er erklæret i. Mere omkring biblioteker i Dart i afsnit 3.4.

Udover at kunne definere variabler og metoder i klasser er det også muligt at definere *properties* i form a *get* og *set*-metoder der giver adgang til et objekts data. Fra start er der allerede oprettet en *get*-metode for alle variabler i en klasse og eventuelt også en *set*-metode hvis variablen ikke er en *const* eller *final*. Derudover er det muligt i Dart at definere yderligere *properties* ved at bruge den syntaks der ses i kodeeksempel 3.9.

```
1 class House {
2  num _adults;
3  num _children;
4
5  num get sum => _adults + _children;
6  set output(String text) => print(text);
7 }
```

Kodeeksempel 3.9: *Eksempel på brug af properties i Dart.*

Ved at bruge *properties* er det muligt at give adgang til data der ikke umiddelbart ligger direkte i en variabel men som derimod skal genereres ved brug. Der er defineret en række anbefalinger for hvornår det er praktisk at bruge en *get*-metode i Dart [6]:

- **Metoden tager ikke imod argumenter.**
- **Metoden returnerer en værdi.**
- **Metoden har ingen side-effekter**. Ved brug af metoden skal der ikke ændre nogen tilstand i klassen der er synlig uden for klassen (intern cache og lignende er derfor OK). Ved kald af metoden gentagende gange skal der returneres den samme værdi med mindre objektet er blevet ændret eksplicit imellem kaldene.

- **Metoden er hurtig.** Brugeren forventer at kald som fx *foo.bar* udføres hurtigt.

For *set*-metoder er der ligeledes en række anbefalinger for hvornår der bør bruges en *set*-metode frem for en almindelig metode:

- **Metoden tager imod et enkelt argument.**
- **Metoden ændrer objektets tilstand.**
- **Metoden har en tilsvarende *get*-metode.** Det føles underligt for brugeren hvis det er muligt at have et stadie der kan modificeres men ikke kan læses igen (det modsatte er ikke et problem; det er fint at kun tilbyde en *get*-metode uden at tilbyde en tilsvarende *set*-metode).
- **Metoden er idempotent.** Ved kald af den samme *set*-metode flere gange i træk med samme værdi bør ikke ændre tilstand i objektet efter første gang at være blevet kaldt.
- **Metoden er hurtig.** Brugeren forventer at et udtryk som *foo.bar* = *value* udføres hurtigt.

3.3 Funktioner/metoder

Som før nævnt i dette kapitel er det muligt at definere funktioner og metoder i Dart. Grundet det valgfrie typesystem er det ikke nødvendigt at definere typer ved disse men betragtes som god praksis at definere typer for returværdien samt parameter til metoder. Hvis en funktion ikke returnere noget vil den automatisk returnere værdien *null*. I kodeeksempel 3.10 er der vist en række måder at definere den samme metode i Dart. Ved de sidste to eksempler er det kun muligt hvis det er et enkelt udtryk der er defineret. Det er derfor ikke muligt at definere en kontrolstruktur som f.eks. et *if-statement* men derimod er det muligt definere et betinget udtryk som fx en *short if*. Et eksempel på dette kan ses i linje 11 i kodeeksempel 3.10.

```
1 String sayHello(String name) {
2  return("Hello $name");
3 }
4
```

```
5  sayHelloShort(name) {
6   return("Hello $name");
7  }
8
9  String sayHelloShorter(String name) => "Hello $name";
10
11 sayHelloShortest(name) => \
12     name != null ? "Hello $name" : "Hello stranger";
```

Kodeeksempel 3.10: *Forskellige måder funktioner kan defineres i Dart. Typer bruges ikke under kørsel af programmet som forklaret i afsnit 3.1, men gør det kun lettere for udviklere at vide hvordan en funktion skal kaldes og benyttes.*

3.3.1 Valgfrie parametre

Dart giver mulighed for at definere valgfrie parametre for metoder således det er muligt at definere at det ikke er alle argumenter der er nødvendige at udfylde. Der er i Dart to sprogkonstruktioner til at angive valgfrie parametre:

Navngivet valgfrie parametre giver mulighed for at parametre angives i en vilkårlig rækkefølge men kræver at hvert valgfrit argument er angivet ved navn. Kan ikke kombineres med positionsbestemte valgfrie parametre.

Positionsbestemte valgfrie parametre kræver at de valgfrie parametre kommer i en bestemt rækkefølge. Alle valgfrie parametre skal komme efter de parametre der er et krav at opfylde. Kan ikke kombineres med navngivet valgfrie parametre.

Et eksempel på navngivet valgfrie parametre samt brugen af en funktion der har defineret navngivet valgfrie parametre kan ses i kodeeksempel 3.11. For at en funktion kan tage en kombination af parametre og valgfrie parametre er det et krav at alle valgfrie parametre (uanset slags) tilføjes til sidst i listen af parametre:

```
1  void attack(String hero, {bool magic: false, int damage}) {
2   if (?damage) {
3      if (magic) {
4      print("$hero attack and add $damage magic damage!");
5      } else {
```

```
6       print("$hero attack and add $damage damage!");
7       }
8   } else {
9       if (magic) {
10      print("$hero try using magic but missing target!");
11      } else {
12      print("$hero missing target!");
13      }
14  }
15  }
16
17  int main(){
18   attack("Bo");
19   attack("Bo", magic: true);
20   attack("Brian", damage: 50);
21   attack("Jacob", damage: 100, magic: true);
22
23   // Output:
24   // Bo missing target!
25   // Bo try using magic but missing target!
26   // Brian attack and add 50 damage!
27   // Jacob attack and add 100 magic damage!
28  }
```

Kodeeksempel 3.11: *Eksempel på brug af navngivet valgfrie parametre.*

Fælles for begge typer valgfrie parametre er det er muligt at definere standardværdier for dem. Et eksempel på dette kan ses ved linje 1 i kodeeksempel 3.11. For at tjekke om en værdi er sat som en del af kaldet af metoden eller der blot bruges standardværdien er det muligt at bruge *"?"* operatoren. Operatoren bruges blandt andet i linje 2 til at tjekke om *damage* er blevet sat ved kaldet af metoden. Som det kan ses i linke 21 i kodeeksempel 3.11 er der gjort brug af en navngiet valgfri parameter der gør det muligt at angive parametre i en valgfri rækkefølge.

Den anden måde at angive valgfrie parametre er ved brug af positionsbestemte parametre som vist i kodeeksempel 3.12. Som navnet antyder antyder er det positionen af de enkelte parametre der har betydning her.

```
1  void count(int number, [String name, String color="red"]) {
2   for (int i = 1; i < number+1; i++) {
3       if (?name) {
4         print("$i $color $name");
5       } else {
6         print("$i $color thing");
7       }
8   }
```

```
}

int main(){
 count(1);
 count(2, "car");
 count(3, "dragon", "black");

 // Output:
 // 1 red thing
 // 1 red car, 2 red car
 // 1 black dragon, 2 black dragon, 3 black dragon
}
```

Kodeeksempel 3.12: *Eksempel på positionsbestemte valgfrie parametre.*

Ligesom ved de navngivet valgfrie parametre er det muligt at angive standardværdier for parametre der ikke bliver sat ved kald af metoden. Fælles for både navngivet og positionsbestemte parametre er at standardværdien skal kunne fastsættes før udførsel af programmet (*compile time*). Hvis en parameter ikke har nogen standardværdi og ikke bliver sat ved kald af metoden så vil den automatisk blive sat til værdien *null*.

3.3.2 Funktioner som første-ordens objekter

I Dart er alle funktioner første-ordens objekter hvilket gør at funktioner kan blive brugt som argumenter til andre funktioner og blive gemt som en variabel. I kodeeksempel 3.13 kan der ses et eksempel på hvordan dette kan benyttes med *attack()* metoden fra kodeeksempel 3.11:

```
var enemyAttack = (enemy, hp) => attack(enemy, damage:hp);
var hp = {"Dragon": 500, "Horse": 10, "Human": 5};
hp.forEach(enemyAttack);
```

Kodeeksempel 3.13: *Eksempel på brugen af funktioner som første-ordens objekter der gør det muligt at bruge funktioner som parametre til andre funktioner.*

I eksemplet defineres en ny metode som en variabel ved navn *enemyAttack* der tager imod to parametre. Funktionen sendes herefter som parametre til *forEach()* metoden der er defineret ved *void forEach(void f(K key, V value))* og som accepterer en funktion der tager imod to parametre.

3.4 Programbiblioteker

Programbiblioteker (*libraries*) er en vigtig del af Dart som gør det muligt at specificere hvilke dele af sprogets standard biblioteker der skal kunne anvendes i en given applikation. Derudover giver biblioteker mulighed for at definere egne biblioteker der kan inkluderes i andre projekter. For at benytte biblioteker i Dart er der tre vigtige nøgleord: *import*, *part* og *library*.

Alle Dart programmer inkluderer automatisk biblioteket *dart:core* som indeholder de mest brugte funktioner og klasser i Dart. Ved brug af yderligere funktionalitet er det muligt at importere yderligere biblioteker som vist i kodeeksempel 3.14. I eksemplet inkluderes biblioteket *dart:io* der kun kan benyttes hvis programmet kører i Dart VM udenfor browseren og giver blandt andet adgang til læse/skrive filer på systemet. Alle *import* skal oprettes før det aktuelle program (erklæring af klasser og metoder).

```
1 import 'dart:io';
```

Kodeeksempel 3.14: *Eksempel på importering af kode fra standardbibliotek. Standardbiblioteker i Dart API'et begynder altid med "dart:".*

Udover at benytte biblioteker giver Dart også mulighed for at definere egne biblioteker med nøgleordene: *library* og *part*. Et bibliotek definere som en fil der i starten indeholder *library <navnet på biblioteket>* og efterfølgende ingen eller flere *part*. Formålet med *part* er at pege på filer der hører ind under biblioteket. For at undgå at en fil kan tilhøre flere biblioteker skal alle filer der inkluderes i et bibliotek indeholde *part of <library name>* som det første i filen. Et eksempel på dette kan ses i kodeeksempel 3.15 der består af to filer.

```
1  /// data.dart
2  library data;
3  part "message.dart";
4
5  /// message.dart
6  part of data;
7  class Message {
8   String text;
9   String nickname;
10  int _secret = 42;
11  Message(this.nickname, this.text);
12 }
```

Kodeeksempel 3.15: *Definition af programbiblioteket "data". Eksemplet består af filerne "data.dart" og "message.dart".*

En vigtig detalje omkring biblioteker er der kun kan opnås adgang til private data inde fra samme bibliotek som de private data er erklæret. Et program der importerer "data" biblioteket som vist i kodeeksempel 3.15 vil ikke have mulighed for at få direkte adgang til variablen *_secret* i *Message* klassen. Dette er også beskrevet i afsnit 3.2.

Dart tilbyder igennem tjenesten `pub.dartlang.org` at indsende samt hente Dart programbiblioteker og linke disse til projekter. Dette gør det enkelt at udvikle nye biblioteker til sproget og efterfølgende sende dem ind således andre udviklere kan genbruge disse. Mængden af funktionalitet på `pub.dartlang.org` er stadig begrænset på nuværende tidspunkt.

3.5 Asynkron programmering

Dette afsnit er et uddrag af kapitlet "Asynchronous Programming" fra rapporten udarbejdet under forundersøgelsen [1, kapitel 3]. Kapitlet er oversat således det passer med sproget i denne rapport.

Asynkron programmering er en måde at optimere svartider på i et givent program ved at undgå at udtryk i programmet blokerer imens der ventes på udtrykket evalueres. Typiske tilfælde af blokering er ved IO metoder som f.eks. adgang til filsystemet eller netværksressourcer. Problemet med asynkron programmering er at det er vanskeligt at skrive på en måde der er nemt efterfølgende at læse. Årsagen til dette er at i stedet for at returnere en værdi så vil en asynkron metode kalde en *callback*-metode. Dette har den konsekvens at programudførslen ikke længere er lineær hvilket gør at det kan være svært at finde frem til hvad en metode gør uden at se det i den kontekst som den bliver kaldt fra.

3.5.1 Futures

Dart's måde at håndtere asynkrone funktioner, der returnerer en værdi, er ved brug af objekter oprettet ud fra klassen *Future*. En asynkron metode vil omgående returnere et *Future* objekt som er en henvisning til en værdi der er tilgængelig på et senere tidspunkt. Den asynkrone metode vil herefter fortsætte i f.eks. en anden tråd og senere aflevere værdien til *Future* objektet.

Kodeeksempel 3.16 viser et eksempel på hvorledes *Future* objekter benyttes i Dart. Funktionen *delayed* er en asynkron metode der tager imod en inputværdi og forsinker værdien med 500ms. I stedet for at programudførslen går i stå vil et kald af *delayed* returnere et *Future* objekt oprettet ud fra et *Completer* objekt. Den aktuelle værdi vil efterfølgende blive gjort tilgængelig for *Future* objektet når *Completer* objektet modtager værdien igennem *complete* metoden som det kan ses i linje 3 i kodeeksempel 3.16.

```
1 Future<T> delayed(T result){
2   var c = new Completer();
3   new Timer(500, (t) => c.complete(result));
4   return c.future;
5 }
6 Future<num> future = delayed(1+2*3);
7 future.then((v) => print("1+2*3 = $v"));
8 print("done");
```

Kodeeksempel 3.16: *Asynkrone kald ved brug af Future klassen.*

For at modtage en værdi fra en *Future* instans er det nødvendigt at tilknytte en *callback*-metode til objektet. Metoden vil senere blive kaldt med værdien som parameter.

Dette minder i høj grad om hvordan andre asynkrone sprog, som JavaScript, benytter *callback*-metoder. Faktisk, så kan man ved at se på den *callback* baseret implementering i kodeeksempel 3.17, som gør præcis det samme som den *Future* baseret løsning, godt argumentere, at brugen af *Future* ikke gør koden mere simpel, men derimod tilføjer yderligere unødvendig kode. Dog tilføjer *Future* klassen en række yderligere funktioner der gør asynkron programmering nemmere når koden bliver mere kompliceret end i de forrige eksempler.

```
void delayedCb(result,callback){
  new Timer(500, (t) => callback(result));
}
delayedCb(1+2*3,(v) => print("1+2*3 = $v"));
print("done");
```

Kodeeksempel 3.17: *Asynkron kald ved brug af "callback"-metode.*

Sammensatte Futures

En interessant mulighed ved *Future* objekter er muligheden for at sammensætte dem på en måde der minder om *pipes* i en Unix shell. Dette gør det muligt for programmøren at fjerne indlejret *callback*-metoder fra hinanden og i stedet sætte en *callback*-metode sammen med resultatet fra en anden *callback*-metode.

Som et eksempel på hvorledes dette virker udvides forrige eksempel ved at tilføje tre uafhængige langsomme matematiske operationer: *slowMultiply*, *slowAdder* og *slowDivide*. Disse operationer vil benyttes til at evaluere udtrykket: $\frac{5\times3+4}{2}$.

Uden *Futures* ville det være nødvendigt at lave samme eksempel ved brug af indlejret *callback*-metoder der sender returværdien fra en operation til den næste. I kodeeksempel 3.18 vises der et eksempel på hvorledes dette kan implementeres i Dart.

```
void slowMultiplyCb(a,b,callback) => delayedCb(a*b,callback);
void slowAdderCb(a,b,callback) => delayedCb(a+b,callback);
void slowDivideCb(a,b,callback) => delayedCb(a/b,callback);
slowMultiplyCb(5,3,
    (x) => slowAdderCb(x,4,
      (y) => slowDivideCb(y,2,
        (z) => print("(5 x 3 + 4) / 2 = $z"))));
print("done");
```

Kodeeksempel 3.18: *Langsom beregning der bruger "callbacks".*

Hvis der i stedet bruges sammenkædning af *Future* objekter er det muligt at skrive samme beregning som vist i kodeeksempel 3.19.

```
Future<num> slowMultiply(int a, int b) => delayed(a*b);
Future<num> slowAdder(int a, int b) => delayed(a+b);
Future<num> slowDivide(int a, int b) => delayed(a/b);
```

```
4 slowMultiply(5,3)
5   .then((x) => slowAdder(x,4))
6   .then((y) => slowDivide(y,2))
7   .then((z) => print("(5 x 3 + 4) / 2 = $z"));
8 print("done");
```

Kodeeksempel 3.19: *Langsom beregning der bruger sammenkædet "Futures".*

Metoden *then* returnerer et *Future* objekt der repræsenterer den værdi der returneres fra *callback*-metoden der er givet til *then* metoden. Hvis værdien fra *callback*-metoden ikke ikke i forvejen er et *Future* objekt vil værdien automatisk blive pakket ind som en *Future*. [2]

Synkronisere samtidige operationer

Indtil videre er der blevet vist hvorledes det er muligt at sammenkæde asynkrone operationer så hver operation venter på resultatet fra den forrige. En anden mulighed ved Dart's *Futures* er mulighed for at synkronisere en liste af *Futures* til et samlet *Future* objekt der repræsenterer alle resultater i en liste med værdien fra hver *Future*.

Et eksempel på dette kan ses i kodeeksempel 3.20. *Future.wait* metoden returnerer en *Future* der repræsenterer en liste af resultater fra de *Futures* metoden tager imod som argument. Resultatet kan senere blive behandlet som ethvert andet *Future* objekt.

I eksemplet beregnes en liste af primtal i et interval mellem 1 og 1000. For at gøre det muligt at udnytte kraften fra moderne multikerne processorer er opgaven splittet op i fire underopgaver som efterfølgende køres samtidigt. Sidst bliver resultatet samlet sammen til en samlet liste.

[2] Dette var ikke tilfældet da rapporten blev skrevet hvor *then* metoden var splittet op i tre metoder: *then*, *chain* og *transform*. Metoderne skulle dengang bruges alt efter om den returneret værdi var en *Future* eller en anden type objekt. I den forrige rapport blev det noteret at *then* løsningen var under udvikling og ville erstatte den forrige måde at sammenkæde *Futures*. *[7]*.

```
Future<List<int>> primes(int from, int to){
  return spawnFunction(_primes).call(range(from,to));
}
Futures.wait([
    primes(1,250),
    primes(251,500),
    primes(501,750),
    primes(751,1000)])
  .then((results){
    var composite = [];
    results.forEach((sublist) => composite.addAll(sublist));
    print(composite);
  });
```

Kodeeksempel 3.20: *Beregning af primtal i fire asynkrone parallelle beregninger.*

3.5.2 Isolates

Samtidighed (*concurrency*) i Dart opnås ved benyttes af *Isolates* som hver kører i sin egen tråd og kan kommunikere mellem hinanden ved brug af afsendelse og modtagelse af beskeder (*message- passing*). *Isolates* har ikke nogen delt hukommelse hvilket gør det muligt at lave flertrådet programmering uden brug af låse [3].

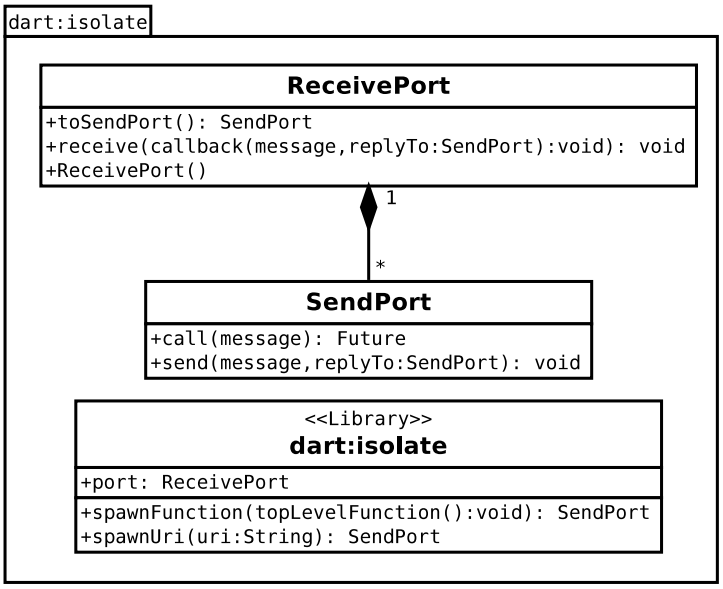
I det nuværende stadie er det muligt at oprette *Isolates* fra enten en funktion på top-niveau eller fra en URI. Hvis *Isolate* instansen oprettes ud fra en URI vil koden blive kørt på den samme maskine som forsøger at oprette instansen. Det er på nuværende tidspunkt ikke muligt at oprette en distribueret *Isolates* ved brug af indbygget metoder i Dart.

Kommunikation mellem *Isolates* sker ved brug af to klasser: *SendPort* og *ReceivePort*.

En *SendPort* er forbundet sammen med en enkelt *ReceivePort* og er den eneste måde at sende beskeder til en *Isolate*. Som standard har en *Isolate* en enkelt *ReceivePort* der er defineret som en top-niveau *final* variabel i *Isolate* biblioteket ved navn "*port*". Hvis der er behov for yderligere *ReceivePort* objekter kan de oprettes. *SendPort* objekter er oprettet ud fra et *ReceivePort* objekt og kan kopieres mellem *Isolates*.

Et UML klassediagram over de væsentligste klasser fra *dart:isolate* bibliote-

ket kan ses i figur 3.1.



Figur 3.1: *De vigtige dele af "dart:isolate" biblioteket der indeholder to klasser og nogle få top-niveau funktioner.*

Oprettelse af Isolates

spawnFunction opretter en ny *Isolate* baseret på en top-niveau funktion der ikke tager imod nogen argumenter. Metoden benyttes som textitmain-metode i den nyoprettede *Isolate*.

For at kommunikere med en *Isolate* er det nødvendigt at registrere en *callback*-metode der kaldes ved modtagelse af beskeder inde i den pågældende *Isolate*. *Callback*-metoden skal her tage imod to argumenter:

message kan være en primitiv type, en *SendPort* eller et *Map* eller *List* objekt der består af disse typer.

replyTo en *SendPort* som et eventuelt svar skal sendes til.

```
1 echo(){
2   port.receive((msg,replyTo){
3       var reply = msg.toUpperCase();
4       replyTo.send(reply);
5       });
6 }
7 SendPort echoPort = spawnFunction(echo);
8 echoPort.call("Hello, Isolated World").then(print);
```

Kodeeksempel 3.21: *Oprettelse af isolate, afsendelse af besked og herefter udskrivning af svaret.*

I kodeeksempel 3.21 oprettes en ny *Isolate* ud fra metoden *echo* som efterfølgende vil være *main*-funktion i den nyoprettede *Isolate*. *echo* metoden sætter efterfølgende en *callback*-metode op til håndtering af beskeder der konverterer alle modtaget beskeder til store bogstaver med *toUpperCase* og svarer derefter tilbage med resultatet til den *Isolate* der foretog kaldet.

Metoden *SendPort.call* benyttes til at sende en enkelt besked til en *Isolate* og fungerer ved der oprettes en midlertidig *ReceivePort* instans der bruges til at modtaget svaret fra den *Isolate* der kaldes. Når svaret modtages vil den *Future* der returneres af *SendPort.call* blive opfyldt med svaret således en given *callback*-metode kan kaldes.

Brugerdefineret Call implementering

Den anden måde at sende beskeder til *Isolates* er ved brug af metoden *SendPort.send*. Forskellen på *SendPort.send* og *SendPort.call* er at ved sidstnævnte er det muligt muligt at håndtere et enkelt svar hvilket håndteres internt. Ved *SendPort.send* er det derimod muligt at specificere hvilken *SendPort* en *Isolate* skal bruge til at sende svaret. Derved er det muligt at oprette en brugerdefineret håndtering af beskeder der tillader f.eks. flere beskeder som svar.

I kodeeksempel 3.22 vises et eksempel på en implementering der gør det samme som *SendPort.call* metoden ved at gøre brug af *SendPort.call* sammen med en brugerdefineret *RecievePort* med tilhørende *callback*-metode.

I eksemplet benyttes *echo* funktionen fra kodeeksempel 3.21 ved oprettelse af en ny *Isolate*. Men i stedet for at bruge *SendPort.call* metoden så afleveres

SendPort objektet til *myCall* metoden som returnere en *Future* der repræsenterer værdien fra den oprettet *Isolate*. Senere får denne *Future* tildelt sin værdi når der er opsat en intern *ReceivePort* og der er kommet svar tilbage fra den oprettede *Isolate*.

Selvfølgelig er dette kun brugbart som et eksempel til at illustrere hvordan *SendPort.call* fungerer internt. Men der er mange tilfælde hvor det er brugbart at benytte *SendPort.send* metoden frem for *SendPort.call*.

```
1  Future myCall(sendPort,msg){
2    var privatePort = new ReceivePort();
3    var c = new Completer();
4    privatePort.receive((msg,_){
5        c.complete(msg);
6        privatePort.close();
7        });
8    sendPort.send(msg,self.toSendPort());
9    return c.future;
10 }
11 myCall(spawnFunction(echo),"Hello, Isolated World").then(print);
```

Kodeeksempel 3.22: *Brugerdefineret implementering af den indbyggede funktion "SendPort.call".*

Intern kommunikation

Et sidste eksempel med kodeeksemplet 3.23 viser kommunikation mellem to *Isolate* instanser. I eksemplet oprettes der to *Isolate* instanser der begge er oprettet ud fra funktionen *autoreply* som *main*. Begge *Isolate* instanser er sat op til at tage imod en besked, ændre på beskeden og sende den tilbage til afsenderen. For at undgå dette varer evigt har de begge indbygget en begrænsning på et antal beskeder der kan afsendes før der lukkes for *RecieevePort*.

En interessant ting at bemærke ved dette eksempel er at efter kommunikationen er sat op og de to *Isolate* instanser, *alice* og *bob*, kommunikerer mellem hinanden så har den oprindelige *Isolate* ikke nogen indflydelse på kommunikationen. I dette tilfælde ville det betyde at hvis den *Isolate* der danner grundlag for det kørende Dart program ikke har nogen åben *ReceivePort* så ville Dart programmet lukke ned og automatisk lukke alle andre *Isolates* der måtte køre i baggrunden.

```
autoreply(){
  var replylimit = new Random().nextInt(10);
  port.receive((msg,reply){
    print("> $msg");
    reply.send("$msg $msg",port.toSendPort());
    if(replylimit-- == 0){
      print("> Goodbye");
      port.close();
    }
  });
}
var alice = spawnFunction(autoreply);
var bob = spawnFunction(autoreply);
alice.send("Hi",bob);
```

Kodeeksempel 3.23: *To "Isolate" instanser der sender beskeder til hinanden indtil den ene af dem lukker forbindelsen.*

Kapitel 4

Analyse

I dette kapitel undersøges der hvorledes det er praktisk muligt at implementere programbiblioteket *distributed_dart* med det formål at kunne sende programkode til noder i et netværk. Noderne skal efterfølgende køre den kode der bliver modtaget samt gøre det muligt at kommunikere mellem noden og den klient der afsendte koden. I analysen fokuseres der på mulige løsning til at løse problemstillingen samt hvilke designmæssige begrænsninger der er nødvendige at foretage.

Kapitlet er opdelt i en række afsnit og er opdelt ud fra hvilke delproblemer programbiblioteket *distributed_dart* løser for at løse det overordnet problem der er beskrevet i problemformuleringen i kapitel 2. Første afsnit præsenterer overordnet hvilke problemstillinger *distributed_dart* står overfor med det formål at analysere mulige løsninger og afgræsninger.

4.1 Overordnet model

Overordnet er formålet med *distributed_dart*, at gøre det muligt for Dart programmer på et netværk, at forespørge andre Dart programmer på netværket (her kaldet *noder*), om at køre programkode som kommer fra maskinerne der laver forespørgslerne. Herefter er det muligt at kommunikere mellem Dart programmet der oprettede forespørgselen, og den kode der er blevet startet på en node i netværket. Figur 4.1 illustrerer dette med et eksempel på et enkelt "Dart Program" der allerede har fået "Kode" til at køre på "Node".

Figur 4.1: *Overordnet model over arkitekturen ved et distribueret system der arbejdes med i dette projekt.*

Efter koden er sat til at køre på node i netværket i figur 4.1 er der oprettet en kommunikationskanal mellem Dart programmet og den nu kørende kode (som foregår igennem noden) og som giver mulighed for at sende data begge veje over netværket.

For er udarbejde løsningen er der en række problemstillinger der er nødvendige at analysere. Disse problemstillinger er inddelt i følgende områder:

Distribuerings Model omhandler teorien omkring forskellige typer af distribueret systemer der findes.

Dynamisk indlæsning af kode. Beskrivelse af hvilke muligheder der er i Dart for dynamisk under kørsel at inkludere yderligere programkode der skal udføres.

Håndtering af afhængigheder. Programkode kan være afhængig af programbiblioteker eller andre filer. Denne del undersøger problemstillinger der er omkring at indhente oplysninger om afhængigheder til et givent Dart program.

Kommunikation mellem noder. Beskriver hvilken tilgang til kommunikation mellem noder i et netværk er fordelagtig at benytte ved *distributed_dart*.

4.2 Distribuerings model

Der findes forskellige typer af distribuerede systemer, som hver især stiller krav til programmeringsmodellen. Disse typer kan groft inddeles i to forskellige typer.

Homogene Systemer I denne klasse af distribuerede systemer, er formålet at forøge mængden af processor kræfter via i et *cluster computing setup*. I sådan en opsætning spiller det ingen rolle hvilken node der fysisk eksekverer et givent program.

Heterogene Systemer I denne type af distribuerede systemer, har de forskellige noder i systemet dedikerede roller. Disse roller kan være bestemt ud fra specialiserede opgaver de enkelte noder skal stå for. Dette

kan være styret ud fra tilslutning af ekstern hardware, eller adgang til specielle services, som ikke er universelt tilgængeligt for hver node i netværket, men bliver gjort tilgængelig via et distribueret program.

For et homogent system, er det ikke nødvendigvis relevant fra en applikations synspunkt, at have nogen viden om netværkstopologien i systemet. Det er i stedet mere relevant at kunne skrive software som er i stand til at skalere, og opnå en maksimal udnyttelse af de tilgængelige ressourcer.

For Heterogene systemer, er situationen omvendt. Her er det nødvendigt at have viden om hvilke typer af ressourcer der eksisterer i systemet, og hvordan disse kan tilgås.

Der er altså lagt op til to forskellige måder at understøtte distribueret programmering. For homogene systemer, er det ikke i programmørens interesse at styre hvilke noder en given kode bliver startet på. Det er derfor en opgave som bibiblioteket bør tage sig af. Det er derfor nødvendigt at have et modul som har kendskab til hvad der eksisterer af noder, og hvilken belastning de hver især har, for automatisk at kunne tildele en node som en given kode skal køres på.

For heterogene systemer, har programmøren behov for at vide hvilke typer af ressourcer det distribuerede system skal tilbyde. Biblioteket kan derfor lade enkelte noder registrere hvilke ressourcer de tilbyder systemet, og dermed automatisk tildele ressourcer, tilsvarende den før beskrevne opsætning for homogene systemer. En sådan fremgangsmåde stiller således krav til at hver node beskriver sine ressourcer via et Interface Description Language (IDL).

At stille krav om en formel beskrivelse af services via et IDL, kan for mindre distribuerede systemer, være unødvendigt kompliceret. I sådanne tilfælde vil det være bedre at lade det være op til programmøren at holde styr på netværkstopologien, og eksplicit vælge hvilke nodes en given kode skal køres på.

Ingen af de nævnte modeller udelukker hinanden. Både den dynamiske skalarings model, og den dynamiske ressource model, kan implementeres som moduler der bygger oven på den direkte model.

4.3 Dynamisk indlæsning af kode

Problemet som dette afsnit forsøger at behandle er hvorledes det i Dart er muligt at indlæse ny programkode dynamisk mens et program i forvejen kører og samtidig gøre det muligt at kommunikere med den nye kode. Ligeledes er det relevant at undersøge om det er muligt at køre denne kode isoleret således det ikke er muligt at påvirke andre instanser der måtte køre på samme node.

Fælles for disse problemstillinger er det relevant at undersøge hvorledes Dart udnytter flertrådet programmering og hvorledes denne semantik kan genbruges til *distributed_dart*. Det følgende afsnit gør primært brug af kilden Walrath and Ladd [14] samt beskrivelsen af isolates i afsnit 3.5.2.

Eksekvering af Dart programmer sker ved, at Dart VM opretter en isolate og eksekverer det angivet Dart program i denne. For hver isolate er det kun muligt for en enkelt procestråd, at udføre operationer, og for at gøre brug af flere processortråde, er det derfor nødvendigt at oprette flere isolates som hver kan gøre brug af en enkelt tråd.

Isolates fungerer isoleret i forhold til hinanden, og en isolate kan derfor ikke påvirke variabler i en anden isolate. Kommunikation mellem isolates sker udelukkende igennem udveksling af beskeder, og de typer der er tilladt at kommunikere er begrænset til følgende:

- Primitive værdier i form af: *null*, *num*, *bool*, *double* eller *String*.
- *SendPort* objekter.
- *List* eller *Map* af gyldige typer (inkluderer også andre instanser af *List* eller *Map*).
- I specielle tilfælde er det muligt, at sende objekter af andre typer hvis den isolate der sendes til er lavet ud fra en allerede kørende isolate (ved brug af enten *streamSpawnFunction* eller *spawnFunction*). Dette er ikke tilfældet, hvis der er oprettet en isolate ved brug af *spawnUri* metoden (mere om de forskellige måder at oprette isolates på senere).

Semantikken ved brug af isolates giver mening i en distribueret sammenhæng og det er derfor nærliggende at lade *distributed_dart* inspirere ud fra isolates. Netop fordi der gøres brug af beskeder mellem isolates og ikke direkte

metodekald er det muligt at tillade større forsinkelser mellem oprettelse af en forespørgsel og et givent svar uden dette påvirker det enkelte Dart program væsentligt. Dette er muligt fordi semantikken i Dart, omkring brugen af *Future* objekter, i forvejen er designet med henblik på at kunne køre andre operationer mens der afventes svar. En beskrivelse af *Future* objekter og hvordan de bruges kan findes i 3.5.1.

Udfordringen ved *distributed_dart* omkring isolates, er at gøre det muligt at køre ny programkode der kommer fra en anden Dart instans, mens et givent Dart program allerede kører. I Dart er det kun muligt at oprette isolates, ved brug af følgende tre globale metoder fra standardbiblioteket *dart:isolate* [8]:

IsolateSink streamSpawnFunction(void topLevelFunction())
: Opretter en ny isolate baseret på en funktion der er defineret i det globale scope. Den nye isolate deler kode med resten af programmet, og tilstanden fra det oprindelige program klones. Isolates fungerer stadig isoleret fra hinanden, hvilket gør at selvom tilstanden er klonet, så påvirker det ikke andre isolates, hvis fx globale variabler ændres. Der returneres en *IsolateSink* der tager imod data, som senere kan modtages af den nye isolate instans.

SendPort spawnFunction(void topLevelFunction())
: Gør det samme som *streamSpawnFunction* med den forskel at der returneres et *sendPort* objekt. Der er i øjeblikket en debat om denne skal fjernes til fordel for *streamSpawnFunction*.

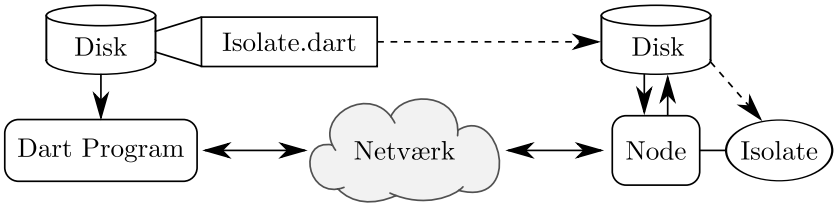
SendPort spawnUri(String uri)
: Benyttes til at oprette en ny isolate baseret på en URI, der peger på et Dart program på harddisken. Dart programmets *main* metode vil blive kørt efter oprettelse af den nye isolate [4]. En vigtig detalje ved denne metode er at der ikke bliver klonet nogen tilstand ud fra den oprindelige isolate, samt det ikke er muligt at sende andre objekter end dem der er specificeret i begyndelsen af afsnit 4.3.

Det er kun en ud af disse tre globale metoder, der gør det muligt at køre programkode i Dart, der ligger udenfor det allerede kørende program. Både *streamSpawnFunction* og *spawnFunction* kræver at metoden der køres, allerede er en del af det kørende Dart program. Denne løsning er realistisk hvis alle noder kører samme Dart program som hovedprogrammet, og derfor har adgang til samme metode. Men eftersom formålet med *distributed_dart* er,

at køre vilkårlig kode der ikke er kendt på forhånd af noden, er dette ikke nogen løsning.

En begrænsning ved *spawnUri* er at Dart koden skal ligge på harddisken i form af en eller flere filer, og kan derfor ikke indlæses fra et *String* objekt. Denne begrænsning gør at noder ikke blot kan modtage koden over netværket, og køre denne, men skal derimod først gemme koden på disken, og herefter køre *spawnUri* metoden, med en URI der peger på stedet hvor koden er blevet gemt på disken.

Problemstillingen er illustreret i figur 4.2, hvor et Dart program forsøger, at oprette en ny isolate på en Node ud fra filen *Isolate.dart*. Den stiplede linje symboliserer at filen, skal være repræsenteret på den disk som noden har adgang til, og må derfor kopieres over enten manuelt eller automatisk igennem netværket. Udover *Isolate.dart* er det sandsynligvis også nødvendigt, at flytte afhængigheder til programmet som f.eks. programbiblioteker. Udfordringen omkring afhængigheder er beskrevet i afsnit 4.4.



Figur 4.2: *Oprettelse af isolate kræver koden ligger fysisk på disken.*

4.3.1 Eventuel løsning med mirrors

En mulig løsning på problemet med *streamSpawnFunction* og *spawnFunction* er med standardbiblioteket *dart:mirrors* der gør det muligt, at undersøge den kørende kode under kørsel. Ved at gøre brug af dette, er det sandsynligt at kunne få fat i et *mirror* system der repræsenterer en konkret metode, og konvertere denne til en tekstrepræsentation der kan sendes. Tekstrepræsentationen kan senere pakkes ud til et *mirror* system på det andet system og køre metoden.

Der er dog en række udfordringer omkring, hvor stort et scope af programkoden der er nødvendig, at sende videre over på et andet system, for at kunne køre metoden. Derudover er der også problematikken omkring tilgang

af tilstandsvariabler, om disse skal være kopieret fra det originale system.

Dart biblioteket *dart:mirrors* er desuden ikke færdigudviklet, og mangler stadig at implementere en del af de features der findes i dokumentationen. Der ligger spor i dokumentationen at det eventuelt bliver muligt, at få adgang til kildekoden til forskellige dele af et Dart program, igennem en række metoder, samt andre features der kunne være til gavn for *distributed_dart*. Disse er ved projektets afslutning ikke implementeret, og der er heller ikke noget bud på hvornår disse features bliver implementeret [2, 10].

4.3.2 Opsummering

Det er muligt i Dart, at indlæse ny kode dynamisk i form af oprettelse af en ny isolate med *spawnUri* funktionen, hvor en parameter angiver stien, til koden der skal danne grundlag for den nye isolate. Metoden har dog en begrænsning ved at koden der peges på, skal ligge fysisk på systemet i forvejen, før den kan bruges, hvilket er en problemstilling der skal tages højde for ved implementeringen af *distributed_dart*.

4.4 Håndtering af afhængigheder

I forrige afsnit blev der konstateret at det er muligt, at bruge *spawnUri* funktionen til, at oprette en ny isolate, baseret på en sti til en fil. Der er ikke mange programmer der kun kræver en enkelt fil, hvilket gør at næste udfordring, er problematikken omkring afhængigheder til et givent Dart program der ønskes oprettet som en isolate. Afhængighederne kan opdeles i to hovedgrupper:

Direkte afhængigheder defineres som værende afhængigheder som programmet direkte refererer til i form af imports, og som derfor er nødvendige for at kunne køre programmet. Disse imports kan både være til programbiblioteker, og til filer der deler programmet op.

Kendetegnet for de direkte afhængigheder er, at det er muligt, at analysere kildekoden, og finde frem til hvilke filer et givent program er afhængig af.

Indirekte afhængigheder er andre nødvendige afhængigheder, som programmet har brug for som fx adgang til specifikke filer, men kan også være adgang til f.eks. en database. Fælles for disse afhængigheder er, at de ikke direkte kan analyseres frem til før programkørsel, fordi de typisk ikke er direkte angivet i programmet.

Disse afhængigheder er mere kompliceret at genkende. Det vil sandsynligt være muligt at analysere programmet, og finde frem til hvilke filer koden teoretisk vil kunne forsøge at få adgang til. Dette er dog en krævende process der ikke er garanteret et præcist resultat. En anden løsning er derfor, at lade programmøren angive hvilke afhængigheder programmet har, i en form der er muligt at analysere før programkørsel.

De næste to afsnit vil omhandle om hver type afhængighed og komme ind på forskellige løsninger hvordan disse kan analyseres frem før programkørsel.

4.4.1 Direkte afhængigheder

Disse afhængigheder er direkte angivet som en del af grammatikken for Dart, og er specificeret i sprogspecifikationen for Dart [11]. Ved at se nærmere på sprogspecifikationen, der står skrevet i Extended Backus–Naur Form (EBNF), er det muligt at specificere hvordan Dart programmer angiver afhængigheder. Definitionen for et Dart program ser således ud:

$$\begin{aligned}\langle \textit{libraryDefinition} \rangle \quad ::= \; & [\langle \textit{scriptTag} \rangle]\ [\langle \textit{libraryName} \rangle] \\ & \{\langle \textit{importOrExport} \rangle\}\ \{\langle \textit{partDirective} \rangle\} \\ & \{\langle \textit{topLevelDefinition} \rangle\}\end{aligned}$$

En vigtig detalje omkring *libraryDefinition* er at alle Dart programmer er et programbibliotek, enten eksplicit (brug af *libraryName*) eller implicit (ingen angivelse af *libraryName*). Endnu vigtigere er det at alle muligheder for, at angive afhængigheder er defineret før *topLevelDefinition*, der angiver klasser, metoder, variabler og lignende:

$$\begin{aligned}\langle \textit{topLevelDefinition} \rangle \quad ::= \; & \langle \textit{classDefinition} \rangle \\ \mid \; & \langle \textit{typeAlias} \rangle\end{aligned}$$

| **external** ⟨*functionSignature*⟩ ';'
| **external** ⟨*getterSignature*⟩ ';'
| **external** ⟨*setterSignature*⟩ ';'
| ⟨*functionSignature*⟩ ⟨*functionBody*⟩
| [⟨*returnType*⟩] ⟨*getOrSet*⟩ ⟨*identifier*⟩
⟨*formalParameterList*⟩ ⟨*functionBody*⟩
| (**final** | **const**) [⟨*type*⟩]
⟨*staticFinalDeclarationList*⟩ ';'
| ⟨*variableDeclaration*⟩ ';'

Det er en fordel ved især større programmer da det kun er nødvendigt at scanne samt analysere den første del af et givent Dart program for at få informationen omkring direkte afhængigheder.

Forskellige direkte afhængigheder

Der er en række sprogkonstruktioner i Dart for, at angive en direkte afhængighed for et Dart program til en anden fil. Forskellen på de forskellige konstruktioner, er primært hvorledes Dart skal opfatte de forskellige afhængigheder under kørsel af programmet. Derimod har det ikke den store betydning for *distributed_dart*, eftersom opgaven her er, at finde alle afhængigheder, og sørge for at de er tilgængelige på den node programmet skal køre på. Følgende er grammatikken for de måder det er muligt, at angive en direkte afhængighed i Dart (*import*, *export* og *part*):

⟨*identifierList*⟩ ::= ⟨*identifier*⟩ [',' ⟨*identifier*⟩]

⟨*combinator*⟩ ::= **show** ⟨*identifierList*⟩
| **hide** ⟨*identifierList*⟩

⟨*libraryExport*⟩ ::= ⟨*metadata*⟩ **export** ⟨*uri*⟩ [⟨*combinator*⟩] ';'

⟨*libraryImport*⟩ ::= ⟨*metadata*⟩ **import** ⟨*uri*⟩ (**as** ⟨*identifier*⟩)
[⟨*combinator*⟩] ';'

⟨*importOrExport*⟩ ::= ⟨*libraryImport*⟩
| ⟨*libraryExport*⟩

⟨*partDirective*⟩ ::= ⟨*metadata*⟩ **part** ⟨*stringLiteral*⟩ ';'

Det er derved muligt, at finde alle direkte afhængigheder til en given Dart fil ved, at scanne efter *import*, *export* og *part* og opsamle de URI's der står efterfølgende disse nøgleord. Samtidig kan der spares tid ved, at stoppe scanneren, så snart scanneren støder ind i den første *topLevelDefinition*, da der herefter ikke kan være flere afhængigheder.

I Dart er der flere måder at angive afhængigheder på, alt efter hvor den kommer fra, og som ikke er en del af grammatikken. Det er derimod forskellige måder *URI'en* til afhængigheden er opbygget:

import "dart:io";
: Importerer et programbibliotek fra Dart standard API. Disse biblioteker er tilgængelige på alle maskiner der kan køre Dart programmer fordi bibliotekerne kommer sammen med Dart miljøet.

import "package:crypto/crypto.dart";
: Importerer en Dart fil fra en pakke. Pakker som et givent Dart program er afhængige af, er defineret i filen *pubspec.yaml* og installeres før start af programmet med kommandoen *pub install*. Alle pakker der hentes ned, er gemt i mappen *packages* der ligger i samme mappe som *pubspec.yaml*.

part "src/network/network.dart";
: Angiver at en given fil skal være en del af et programbibliotek, og importeres efterfølgende ud fra den relative sti.

Ved scanning efter afhængigheder, kan almindelige Dart biblioteker blot ignoreres, da disse altid vil være repræsenteret på alle systemer, der kan køre programmet. Omkring pakker er det muligt at finde frem til disse afhængigheder udelukkende ved, at tjekke *pubspec.yaml*. Problemet med denne løsning er at det stadig, er nødvendigt, at scanne Dart programmet efter andre typer af afhængigheder. En fordel ved, at bruge resultatet fra scanningen, er samtidig at det kun er Dart filer der faktisk benyttes som afhængigheder, der bliver registreret som direkte afhængigheder.

4.4.2 Indirekte afhængigheder

Som tidligere beskrevet, har disse afhængigheder det tilfælles, at de ikke indgår som en del af grammatikken for Dart. Det er derimod afhængigheder, der afgøres mere eller mindre dynamisk, ved kørsel af programmet. På grund af dette, er det ikke realistisk, at kunne afgøre disse afhængigheder automatisk, ud fra et givent Dart program, uden nogen vejledning fra udvikleren. Der er under projektet blevet diskuteret følgende forslag, der alle er realistiske måder at løse problemet på:

- Angive afhængigheder i særligt formateret kommentarer i kildekoden af programmet.
- Angive afhængigheder i en ekstern fil med en defineret syntaks.
  - En samlet fil for hele Dart programmet.
  - En fil for hver fil et Dart program består af, således afhængigheder også kan definere yderligere indirekte afhængigheder.

Fordelen ved sidstnævnte løsning, er at den gør det nemt, at scanne efter indirekte afhængigheder, da disse er isoleret til specifikke filer, samtidig med det er muligt at lave en større træstruktur af indirekte afhængigheder, da inkluderet Dart biblioteker derved også har mulighed for, at angive deres egne indirekte afhængigheder.

4.5 Kommunikation mellem noder

Dart's IO api (*dart:io*) tilbyder netværkskommunikation via TCP sockets, som er en stream orienteret forbindelse, der garanterer at pakker ankommer i samme rækkefølge som de blev sendt. Dart understøtter tre forskellige typer af socket forbindelser, som hver især også findes i en "Secure" udgave med understøttelse for SSL kryptering.

1. *RawSocket*
2. *Socket*
3. *WebSocket*

4.5.1 RawSocket og Socket

Forskellen mellem *RawSocket* og *Socket*, er graden af manuel kontrol over udlæsning af data som ankommer til en socket server. Sockets implementerer *Stream* og *IOSink* interfaces for at læse og skrive data til dem. For *Socket* kan man læse data ved at abonnere på en stream. Det samme er tilfældet med *RawSocket*, forskellen ligger i at der for *RawSocket* ikke kan trækkes data ud ved at abonnere på stream interfacet. Dette giver i stedet et *RawSocketEvent*, som fortæller at der er ulæst data på bufferen. Man kan så selv vælge hvordan disse data skal læses.

Socket API'et benytter streams, både ved afsendelse og modtagelse, og der er ikke indbygget nogen abstraktioner til at genkende datatyper ved modtagelse. Sockets opererer udelukkende med datatypen *UInt8List*, som er en liste af 8 bit integers, og det er således op til applikationsprogrammøren at implementere logik til at håndtere serialisering og deserialisering af de datastrukturer som der ønskes at sende via netværket.

Endvidere så er der ikke nogen måder at separere forskellige data strukturer, som bliver sendt på den samme *Socket* instans. Det har den betydning at skrives der flere forskellige objekter til den samme *Socket*, inden for et kort tidsrum, vil de på modtagersiden blive opfattet som en sammenhængende datamængde, idet der at der ikke kommunikeres nogen information omkring hvordan den egentlige datastruktur ser ud.

```
1  import 'dart:io';
2  // print anything received on the socket,
3  void incommingConnection(Socket incomming) {
4    incomming
5      .transform(new StringDecoder())
6      .listen((s) => print("Received: $s"));
7  }
8
9  Future startserver(){
10   return ServerSocket
11     .bind("0.0.0.0",1234)
12     .then((ss) => ss.listen(incommingConnection));
13 }
14
15 void runclient(){
16   Socket
17     .connect("127.0.0.1",1234)
18     .then((Socket socket) {
19         // send 25 messages on the socket
```

```
20          for(int i = 0; i < 25; i++){
21            String data = "${i}~";
22            socket.add(data.codeUnits);
23            print("Sent Message: $data");
24          }
25          });
26 }
27 main() =>  startserver().then((_) => runclient());
```

Kodeeksempel 4.1: *Socket stream eksempel.*

Kodeeksempel 4.1, viser en simpel implementering med kommunikation mellem en *Socket* og en *ServerSocket*. Efter at serveren er startet, opretter klienten en *Socket* forbindelse.

På denne *Socket*, sendes der 25 separate beskeder. På serveren er der registreret en callback funktion som udskriver modtaget data. Denne funktion bliver kaldt hver gang der registeres et nyt event på *Streamen*. På serveren bliver data gjort tilgængelig via stream interfacet på socket forbindelsen, og den registrerede callback funktion bliver kaldt, hver gang streamen signalerer at der er nyt data.

I kodeeksempel 4.2, vises outputtet der genereres ved at eksekvere koden fra fra kodeeksempel 4.1. Her ses det, at på trods af at *Socket.add* er kaldt 25 gange, resulterer det i kun 2 events hos modtageren, hvor de separate strenge opfattes som et enkelt data objekt.

```
1 Sent Message: 0~
2 Sent Message: 1~
3 Sent Message: 2~
4 Sent Message: 3~
5   .... etc (4~ - 23~)
6 Sent Message: 24~
7 Received: 0~1~2~3~4~5~6~7~8~9~10~11~12~13~14~15~16~
8 Received: 17~18~19~20~21~22~23~24~
```

Kodeeksempel 4.2: *Output fra Kodeeksempel 4.1.*

4.5.2 Besked orienteret kommunikation

Problemet med at transformere en stream orienteret protokol om til en besked orienteret, således at der kan skelnes mellem objekter kan løses på flere

forskellige måder. Følgende opstiller 3 metoder:

Hvert objekt bliver sendt på en separat socket
En mulig løsning er at oprette en separat socket for hvert objekt der skal sendes. Det vil forhindre at data bliver blandet sammen hos modtageren, idet hver besked vil blive behandlet af en separat instans af meddelelseshåndteringen. Denne fremgangsmåde, betyder dog at der vil blive oprettet uforholdsmæssigt mange forbindelser. Kodeeksempel 4.1, ville f.eks. skulle oprette 25 separate forbindelser.

Der sendes en separator mellem hvert objekt
En anden mulig løsning er at sende et specielt tegn imellem hvert objekt, og lade modtageren benytte dette til at splitte beskeden.Denne metode er dog også problematisk, hvis den data der sender også indeholder separationstegnet. Dette betyder at afsender siden skal implementere en parser som kan escape instanser af separationstegnet, og omvendt skal modtager siden implementere en unescaper.

Objekter præfixes med en header
En tredje løsning er at præfixe samtlige beskeder med en header som angiver længden af beskeden som sendes. Denne metode har den fordel at den ikke kræver nogen parsing af beskeden, hverken på afsender eller modtager siden.

4.5.3 WebSockets

Et alternativ til *Sockets*, er *WebSockets*, som tilbyder et højere abstraktions niveau end almindelige *Sockets*. Kodeeksempel 4.3 viser et simpelt klient / server eksempel, tilsvarende kodeeksempel 4.1. Til forskel fra rå *Sockets*, sørger *WebSockets* selv for at enkapsulere data, således at der kommer det samme antal beskeder på server siden, som der er sendt. Kommunikationen er således beskeds-orienteret frem for stream orienteret.

```
import 'dart:io';
// print anything received on the socket,
incommingConnection(WebSocket incomming){
  incomming
    .listen((s) => print("Received: $s"));
}

Future startserver(){
  return HttpServer
    .bind("0.0.0.0",1234)
    .then((hs){
        hs
        .transform(new WebSocketTransformer())
        .listen(incommingConnection);
        });
}

void runclient(){
  WebSocket
    .connect("ws://127.0.0.1:1234")
    .then((WebSocket ws) {
        // send 25 messages on the socket
        for(int i = 0; i < 25; i++){
          String data = "${i}~";
          ws.add(data);
          print("Sent Message: $data");
        }
        });
}
main() => startserver().then((_) => runclient());
```

Kodeeksempel 4.3: *Eksempel på simpel websocket server og klient.*

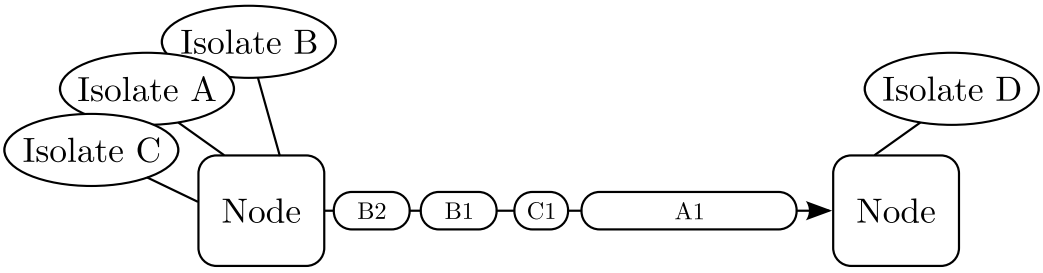
4.5.4 Deling af sockets

Hvis flere uafhængige isolates på en *Node* har behov for at kommunikere med isolates på en anden *Node*, vil der være mulighed for at de kan dele den samme *Socket*.

Her skal man dog være opmærksom på at der kan opstå problemer med at data intensive isolates kan forårsage en service forringelse for øvrige isolates som ønsker at kommunikere med samme node. Figur 4.3 viser tre isolates som deles om en enkelt socket. De tre isolates har forskellige netværks profiler, Isolate A sender store beskeder, B sender mange små beskeder, og C sender

sporadisk en besked.

Problemet ved at dele en enkelt Socket, er således at en isolate som deler socket med en data intensiv isolate, vil få en forøget latency.



Figur 4.3: *Tre isolates delers om en socket.*

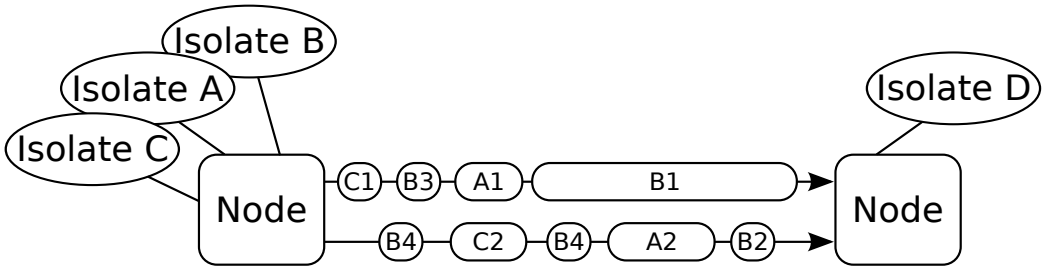
For at modvirke dette problem, men samtidig holde antallet af åbne sockets nede, kan man lave en socket pool, hvor der skiftes imellem forskellige socket. Figur 4.4 viser et eksempel hvor tre isolates deler to sockets.

Ser man nærmere på figuren vil man dog opdage et problem, som potentielt kan bryde med semantikken for isolates i Dart. For isolates gælder det at beskeder skrevet til samme sendport, vil blive modtaget i den rækkefølge de er sendt. Når man ser på figur 4.4 er dette ikke tilfældet. Her vil modtagelsesrækkefølgen i stedet være B2, A2, B4, B1, A1, C2, B3, B4, C1, Fordi den store besked B1, tager længere tid at overføre, imens den anden socket kan overføre en række mindre beskeder.

```
1 class SocketPool {
2   List<Socket> pool = [];
3   int _index = 0;
4
5   add(List<int> data){
6     _index = (_socketIdx+1 % pool.length)
7     return pool[_socketIdx].add(data);
8   }
9 }
```

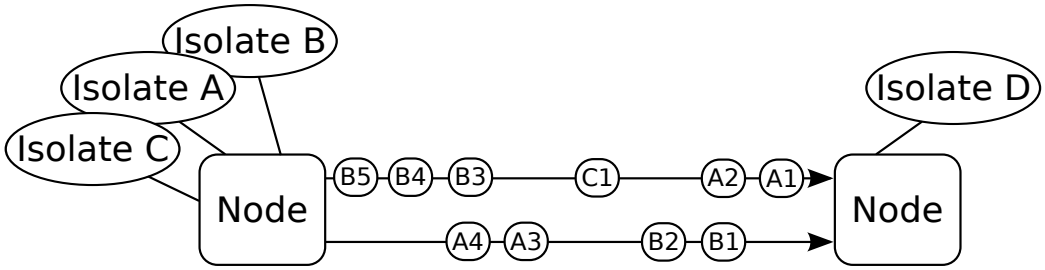
Kodeeksempel 4.4: *Naiv SocketPool implementation som benytter en simpel Round Robin skedulerings algoritme.*

Dette problem kan opstå hvis implementeringen af en socketpool ikke tager højde for detaljer, som afsender, besked størrelse og socket trafik, men i stedet benytter en naiv algoritme, som den vist i kodeeksempel 4.4 som er en simpel round robin algoritme som blot skifter socket ved hver besked.



Figur 4.4: *Isolates sender beskeder på en socket pool med round robin skedulering.*

En bedre løsning vil være at socket pool algoritmen var lidt mere intelligent. Figur 4.5, er et eksempel på en socket pool som sørger for at data fra den samme afsender bliver sendt af sted på samme socket som den forrige besked den sendte.



Figur 4.5: *Tre isolates benytter en socket pool.*

Kodeeksempel 4.5, viser hvordan sådan en algoritme kan implementeres. Hver afsender har tilknyttet en *ClientSocket*, som husker index på den sidst benyttede socket i en SocketPool. Efter et givet tidsrum, her 30 sekunder, hvor denne afsender ikke har sendt nogen data, glemmes det valgte index, og næste besked kan således sendes på en hvilken som helst ledig socket.

Det er så op til *SocketPool* singelton objektet at oprette og lukke sockets, alt efter behov, imens *ClientSocket* objekterne sørger for at beskeder ikke kommer til at overhale hinanden grundet kø-dannelser, som dem vist i figur 4.3.

```
class ClientSocket{
  int _index;
  Timer _timeout;

  add(List<int> data){
    if(_index == null)
      _index = SocketPool.getSocket(_index);

    try {
      SocketPool.socket(_index).add(data);
    } on UnknownSocketException catch (e) {
      _timeout.cancel();
      _index = null;
      add(data):
      return;
    }

    if(_timeout is Timer) _timeout.cancel();
    var t = new Duration(seconds:10);
    _timeout = new Timer(t,() => _index = null);
}
```

Kodeeksempel 4.5: *SocketPool implementering som husker om en afsender har benyttet en specifik Socket inden for et givent interval.*

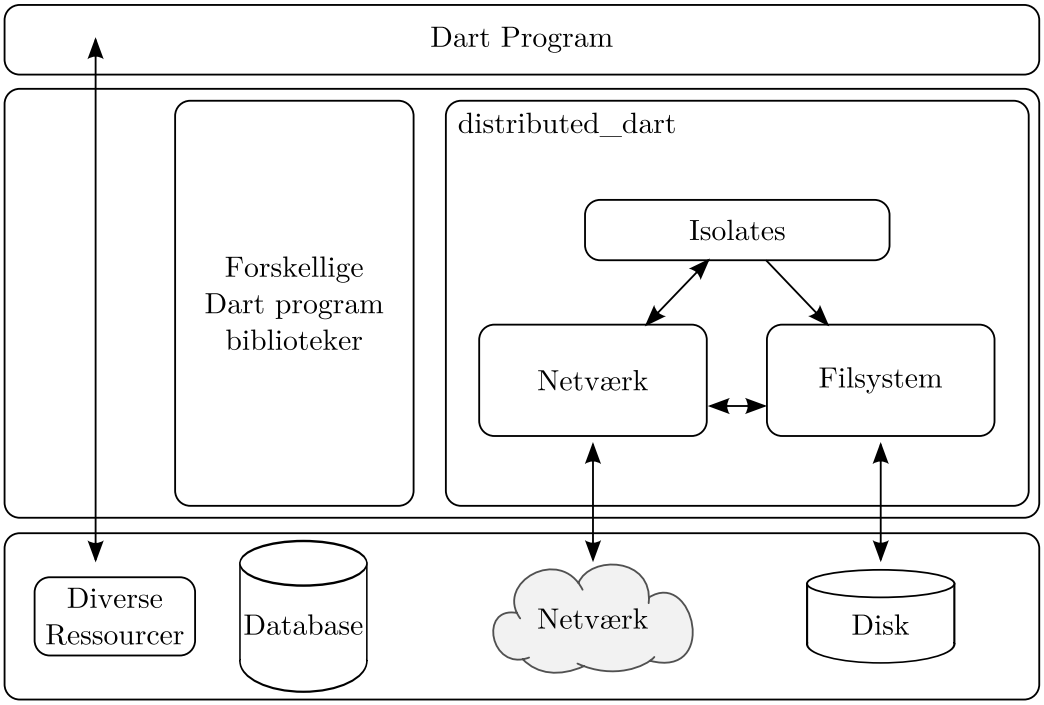
Kapitel 5

Arkitektur

På baggrund af de informationer der er fremlagt i analysen, kan vi nu definere en overordnet arkitektur for *distributed_dart* biblioteket.

Figur 5.1 viser et program, som benytter *distributed_dart*, til at understøtte distribuering. Selve biblioteket er vist i midten af figuren, nemlig den som består af 3 moduler, *Netværk*, *Filsystem*, og *Isolates*.

For at kunne benytte biblioteket, skal hver node i et netværk som man ønsker at arbejde med, importere biblioteket, og kalde funktionen



Figur 5.1: *Overordnet model over arkitekturen ved et distribueret system der arbejdes med i dette projekt.*

distributed_dart biblioteket er designet til at benytte sig af den samme semantik som regulære isolates, hvorfor den også benytter samme API, til

kommunikation: en *SendPort*. *streamSpawnFunction* fra *dart:isolate* biblioteket understøtter også et *IsolateSink* interface til streams, men der er ikke en tilsvarende stream udgave for *spawnUri* funktionen, så det er der heller ikke til *distributed_dart*. Ønsker man alligevel at benytte streams, er det trivielt at wrappe en *SendPort* i et objekt der benytter en *IsolateSink*, men man kan ikke sende *Sinks* mellem isolates.

Det eksterne API som skal benyttes af programmet som skal distribueres, er ganske simpelt, da det kun består af 2 funktioner, og 1 klasse, nemlig disse:

class IsolateNode(String host, {int port})
: En klasse som benyttes til indentificere noder, på netværket. Benyttes også til initialisering i funktionen *registerNode*.

void registerNode(IsolateNode node)
: Denne funktion skal kaldes før en Node kan deltage i netværket. Formålet med funktionen er at starte netværks serveren op for den aktuelle node, samt at garantere at der er initialiseret en *IsolateNode*.

SendPort spawnUriRemote(String path, IsolateNode node)
: Denne funktion benyttes til at starte en ny isolate på en navngiven *IsolateNode* på netværket. Path strengen angiver stien til en lokal dart fil, som er main fil for den isolate der ønskes startet. Det er altså ikke påkrævet at filerne ligger på noden som skal køre programmet, inden den *spawnUriRemote* kaldes. Funktionen returnere en almindelig *SendPort* fra *dart:isolate* biblioteket.

Et minimalt eksempel på brug af biblioteket kan ses i kodeeksempel 5.1 og 5.2. De to programmer kører på hver deres node. Begge noder registrer sig hos biblioteket, og den ene spawner således en isolate på den anden, hvorefter sendes en besked, og afventes et resultat.

```
1 import 'package:distributed_dart/distributed_dart.dart';
2 main() =>  registerNode(new IsolateNode("192.168.1.100"))
```

Kodeeksempel 5.1: *Hello World, beregnings node*

```
1 import 'package:distributed_dart/distributed_dart.dart';
2 main(){
3     registerNode(new IsolateNode("192.168.1.50"))
4     IsolateNode remote = new IsolateNode("192.168.1.100");
5     SendPort sp = spawnUriRemote("/path/to/myprogram.dart", remote);
6     sp.call("hello remote world").then(print);
7 }
```

Kodeeksempel 5.2: *Hello World, hoved program*

De følgende 3 kapitler, omhandler yderligere detaljer om de enkelte moduler i biblioteket.

Kapitel 6

Netværks Modulet

Dette kaptitel omhandler detaljer for implementering af Netværks Modulet.

6.1 Data Kodning

Som tidligere beskrevet i afsnit 4.5, vil der opstå problemer hvis man forsøger at sende flere forskellige beskeder via en socket forbindelse.

Det er dog let at implementere en løsning som tilføjer en header til beskeder, inden de bliver sendt ud på en socket. I kodeeksempel 6.1, ses hvordan en input stream, kan transformeres inden dens indhold bliver sendt ud på en socket. Dette gøres ved at benytte 2 indbyggede transformation, til at klargøre data, samt en metode til at vedhæfte en header.

Processen er som følger:

- Der skrives data til en sink
- Data JSON kodes af en en *JsonEncoder*
- Data transformeres til et byte array
- Størrelsen af bytearrayet bliver præfixet som en header

Af disse punkter, skal kun den sidste implementeres manuelt, *JsonEncoder*, og *StringEncoder*, er en del af darts SDK. Kodeeksempel 6.2, viser hvordan man kan lave en transformation ved at nedarve fra StreamEventTransformer klassen. Headeren består af en *Uint64List*, som angiver længden af beskeden.

```
  stream
   .transform(new JsonEncoder())
   .transform(new StringEncoder())
   .transform(new ByteListEncoder())
   .listen(socket.add);
```

Kodeeksempel 6.1: *Tilføj en header til en meddelelse*

```
// Prefixes a the bytelist object with the size of the json string
// size prefixe is a 64 bit integer, encoded as Uint8List(8)
class ByteListEncoder extends StreamEventTransformer {
    void handleData (Uint8List data, EventSink sink) {
      var size = new Uint8List(8);
      new ByteData.view(size.buffer).setUint64(0,data.length);
      sink.add(size);
      sink.add(data);
    }
}
```

Kodeeksempel 6.2: *Tilføj en header til en meddelelsen*

Kodeeksempel 6.3, viser hvordan afkod er implementeret.

Den har grundlæggende to modes. Enten er den istand til at hente en hel besked, som vil blive sendt videre på streamen med det samme, eller også har den kun modtaget en del af en længere transmission, og er derfor nød til at gemme på en den modtagne del i en intern buffer, indtil den kan læse resten.

```
class ByteListDecoder extends StreamEventTransformer {
  int remaining = -1;
  List<int> obj = [];

  void handleData(Uint8List data,EventSink<List<int>> sink){
    var idx = 0;
    var dataView = new ByteData.view(data.buffer);
    _log("received ${data.length} bytes");

    while (idx < data.length) {
      // prepare to read a new object
      if(remaining == -1 ){
        remaining = dataView.getUint64(idx);
        idx+=8;
        obj = new List();
      }

      // try to get the entire object first
      // if the data is incomplete, add data to buffer,
```

```
20        // and ajust remaining
21        List objpart = [];
22        try {
23          objpart = data.sublist(idx,idx+remaining);
24        }
25        catch (e) {
26          // sublist from index to end
27          objpart = data.sublist(idx);
28        }
29        finally {
30          remaining -= objpart.length;
31          idx += objpart.length;
32          obj.addAll(objpart);
33          _log(" > read ${objpart.length} bytes");
34        }
35
36        // object is assembled, write to sink,
37        // and mark that we are ready to read the next object.
38        if( remaining == 0) {
39          sink.add(obj);
40          remaining = -1;
41          _log(" > done: total ${obj.length} bytes");
42        }
43      }
44    }
45  }
```

Kodeeksempel 6.3: *Parse header*

6.2 Detektere SendPort

En af parameterene for *distributed_dart*, er at der ikke skal være nogen forskel om en isolate startes med *spawnUri*, eller *spawnUriRemote*. For at understøtte fuld interoperabilitet imellem almindelige isolates og distribuerede isolates, er der behov for at kunne identificere om de beskeder der sendes indeholder en *SendPort*, og i så fald udskifte dem med *RemoteSendPorte*, før beskeden bliver sendt på netværket.

Kodeeksempel 6.4 viser implementeringen af en rekursiv parser som benyttes til at udskifte *SendPort* objekter fra forskellige objekter som f.eks. *List* og *Map* objekter, registre dem i isolate modulet, for derefter at udskifte dem med en tilsvarende *RemoteSendPort* objekter.

```
class ReplaceSendPorts {
  List<Object> seen = new List<Object>();

  void checkCycle(final object) {
    seen.forEach((Object o) {
      if (identical(o, object)) {
        throw new JsonCyclicError(object.toString());
      }
    });
    seen.add(object);
  }

  Object scanAndReplaceObject(final object) {
    if (object is num) {
      return object;
    } else if (object is bool) {
      return object;
    } else if (object == null) {
      return object;
    } else if (object is String) {
      return object;
    } else if (object is List) {
      checkCycle(object);
      List a = object;
      if (a.length > 0) {
        a = a.map((Object o) {
          return scanAndReplaceObject(o);
        }).toList(gteringrowable:false);
      }
      seen.remove(object);
      return object;
    } else if (object is Map) {
      checkCycle(object);
      Map<String, Object> m = object;

      m.keys.forEach((String key) {
        if (key is String) {
          m[key] = scanAndReplaceObject(m[key]);
        } else {
          String msg = "Key Must be String";
          throw new NotSerializableObjectException(msg);
        }
      });
      seen.remove(object);
      return object;
    } else if (object is SendPort) {
          return new LocalIsolate
                        .fromSendPort(object)
```

```
49                              .toRemoteSendPort();
50       } else {
51         checkCycle(object);
52         var r = scanAndReplaceObject(object.toJson());
53         seen.remove(object);
54         return r;
55       }
56     }
57 }
```

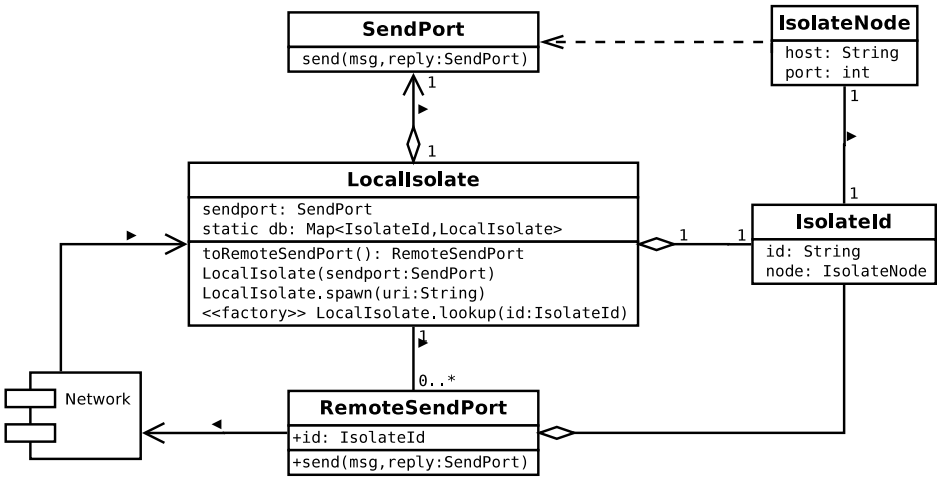
Kodeeksempel 6.4: *Rekursiv Object parser, som kan søger efter SendPorte, og urskifter dem med RemoteSendPorte*

Kapitel 7

Isolate Modulet

Isolate modulet har til formål at sørge for at der kan oprettes isolates på andre noder i det distribuerede netværk, og gøre det muligt at kommunikere med dem ved brug af samme interface som eksisterer for lokale isolates.

Figur 7.1 viser et klassediagram for modulet.



Figur 7.1: *Klasse diagram for isolate modulet*

Hver isolate der skal kunne tilgås fra en anden node, har tilnkyttet et *IsolateId*. IsolateId har to formål.

1. Den skal være en unik identifikation af en isolate
2. Den skal indeholde information om hvor isolaten er placeret

Kodeeksempel 7.1 viser implemetationen af *IsolateId*. Et *IsolateId* genereres når en *SendPort* tilknyttes til et *LocalIsolate* objekt. Et *IsolateId*, er blot et data objekt som indeholder et fortløbende id, samt en reference til den

node den er oprettet på. Fordi der skal køres en netværks service på noden, som lytter på en given port, så kan det ikke lade sig gøre at genere to ens *IsolateId*. Det ville nemlig være betinget af at to forskellige isolates kunne genere det samme *IsolateId*, Hvilket ikke kan lade sig gøre, da *IsolateNode* objektet er kædet sammen med netværks serveren, og der kan ikke køre 2 SocetServer på samme port.

```
1 class IsolateId {
2     static int id = 0;
3     final IsolateNode node;
4     final int id;
5     IsolateId():
6         node = IsolateNode.localhost;
7         id = _nextid++;
8 }
```

Kodeeksempel 7.1: *En IsolateId gøres unik ved at være baseret på noden den er oprettet på, og et fortløbende nummer*

LocalIsolate har til formål at holde styr på de *SendPorts* som skal kunne kontaktes. Den fungerer som en proxy klasse, som vedligeholder en mapping imellem en *SendPort*, og en *RemoteSendPort*.

En *RemoteSendPort* kan betragtes som en netværks pointer. Den er istand til at benytte netværks modulet, og den indeholder information om hvilken node *LocalIsolate* instansen findes på.

Kapitel 8

Filsystem modul

Formålet med filsystem-modulet, er at håndtere alt aktivitet der sker på harddisken ved brug af *distributed_dart*, og gælder både læse og skrive operationer alt efter hvordan biblioteket benyttes. Modulet indeholder ingen funktioner eller klasser som brugeren af *distributed_dart* vil kunne få adgang til. Opbygningen af filsystem-modulet er delt op i følgende tre afsnit:

Scanning af program med afhængigheder.
Beskriver hvorledes en given Dart fil indlæses og efterfølgende scannes for yderligere afhængigheder. Resultatet af denne scanning er en træstruktur, der repræsenterer Dart filen samt alle afhængigheder og afhængigheders afhængigheder.

Overførsel af kildekode over netværk.
Træstrukturen der oprettes ved forrige fase omformes, og laves om til JSON som kan sendes med netværks-modulet. Afsnittet gennemgår desuden protokollen for overførsel af filer igennem netværks-modulet, med det formål at mindske mængden af genereret netværkstrafik.

Oprettelse af miljø ved modtager.
Ved modtagelse af filerne fra forrige fase skal disse placeres på disken i en struktur der gør det muligt at kalde *spawnUri* på Dart filen hvorefter Dart skal kunne finde alle afhængigheder.

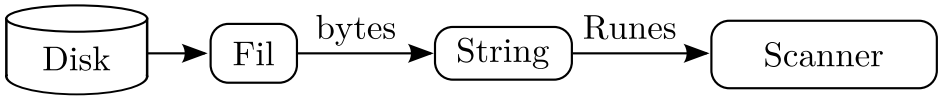
Fælles for alle punkter, er der gøres brug af de resultater, der blev fundet frem ved analysen i kapitel 4, med særlig vægt på afsnittene 4.3 og 4.4.

8.1 Scanning af program med afhængigheder

Første fase starter ved at brugeren af *distributed_dart* har angivet en sti til et program, der ønskes blive kørt på en node i netværket. Før dette kan lade

sig gøre, skal programmet, samt alle afhængigheder til programmet, flyttes over på noden i netværket.

For at analysere hvilke afhængigheder en given fil har, er der lavet en scanner i *distributed_dart*, der tager imod en fil som illustreret i figur 8.1. *Runes* klassen gør det muligt at læse et tegn af gangen uden, at skulle tage højde for at tegnet kan bestå af flere tegn. Problematikken opstår fordi UTF-8 tegnsættet tillader at nogle tegn, kan bestå af et par af tegn og derved vil visse tegn bestå af to fysiske tegn hvis der blot indlæses ud fra strømmen af *bytes*.



Figur 8.1: *Processen fra fil til scanner. Filen indlæses som bytes og konverteres til en Streng for derefter at læse strengen en rune af gangen.*

Scanner klassens opbygning er baseret på designet fra en scanner der bruges internt i Dart til, at scanne Dart kode med det formål, at oversætte Dart til JavaScript[1]. Ved at modificere denne kraftigt er det muligt, at hente oplysninger omkring afhængigheder baseret på grammatikken beskrevet i 4.4. Scanneren returnere en liste over alle afhængigheder som Dart filen har. Et eksempel kan ses i Kodeeksempel 8.2 der viser afhængigheder baseret på et Dart program med den header der kan ses i 8.1.

```
1 import "package:distributed_dart/distributed_dart.dart";
2 import "package:crypto/crypto.dart";
3 import "dart:io";
4 import "dart:json" as json;
```

Kodeeksempel 8.1: *Eksempel på import udtryk fra et Dart program.*

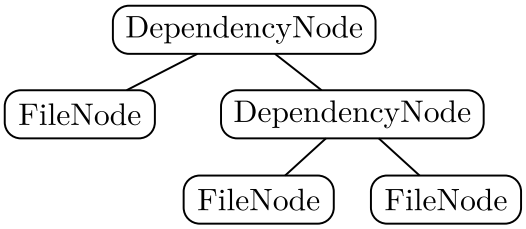
```
1 package:distributed_dart/distributed_dart.dart
2 package:crypto/crypto.dart
3 dart:io
4 dart:json
```

Kodeeksempel 8.2: *Eksempel på output fra scanneren efter et input som 8.1.*

Som det kan ses i Kodeeksempel 8.2 er der to afhængigheder til standardbiblioteker i Dart (*dart:io* og *dart:json*). Disse kan blot ignoreres eftersom disse vil være tilgængelige på alle systemer. De resterende to afhængigheder er filer

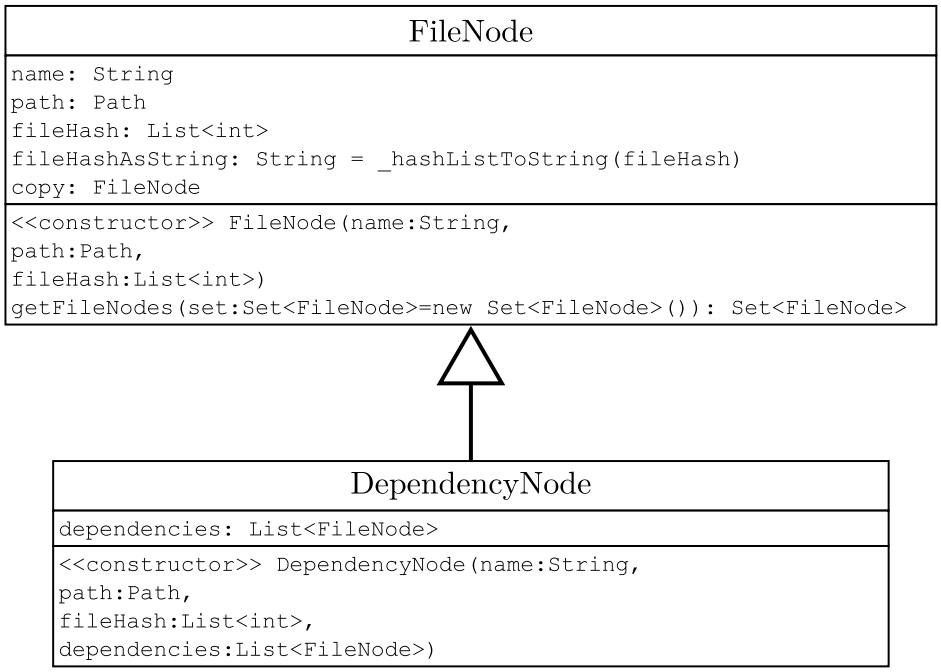
[1]Scanner.dart fra Dart: http://goo.gl/Ut5w7

der er placeret i pakker og har også mulighed for at indeholde afhængigheder til andre filer. Derfor kaldes scanneren rekursivt på samtlige filer der måtte blive opdaget. Resultatet der returneres er en træstruktur som vist i figur 8.2.



Figur 8.2: *Træstruktur der repræsenterer afhængigheder for en Dart fil.*

Træstrukturen består af to klasser og benyttes alt efter om filen der repræsenteres har afhængigheder. Klasserne kan ses i klassediagrammet figur 8.3.

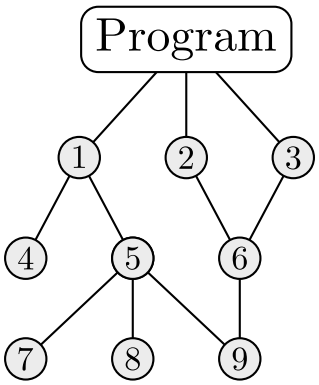


Figur 8.3: *Klassediagram over FileNode og DependencyNode.*

8.1.1 Gentagende filer

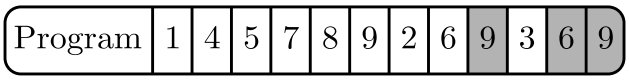
En problemstilling der kan opstå ved oprettelse af træstrukturen er at nogle noder i træet går igen fordi flere filer peger på det samme bibliotek. Problemet

opstår senere i systemet når der forsøges at oprette filer igen på disken og disse filer allerede eksisterer på systemet. Samtidig er det unødvendigt at sende en hel træstruktur der består af flere ens noder. Problemet er illustreret i figur 8.4.



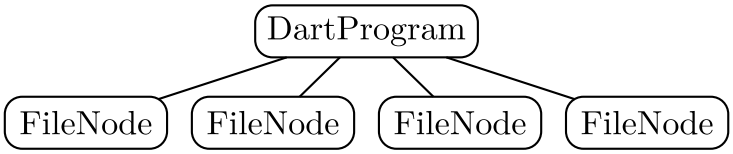
Figur 8.4: *Træ af afhængigheder hvoraf flere af disse er afhængig af samme fil.*

Ved en gennemløbning af træstrukturen er resultat en samlet liste af afhængigheder til Dart programmet som vist i figur 8.5. De grå elementer i figuren angiver afhængigheder der er i listen flere gange.



Figur 8.5: *Liste over afhængigheder bygget ud fra figur 8.4. De grå felter angiver afhængigheder der optræder mere end en gang i listen.*

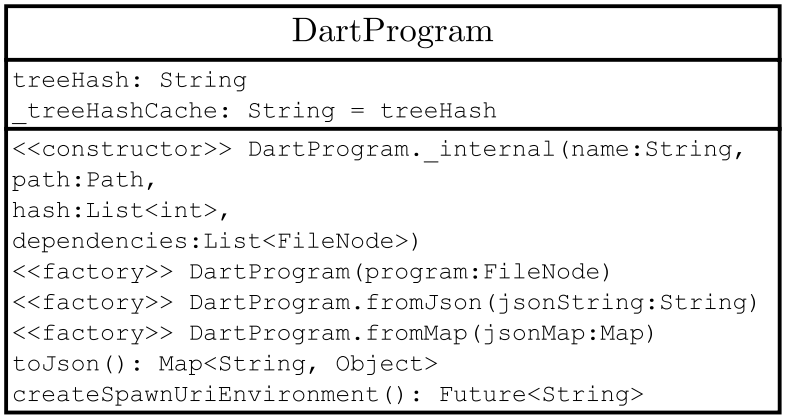
For at undgå dette og samtidig gøre det nemmere at sende en liste over afhængigheder igennem netværket bliver træstrukturen i figur 8.2 senere konverteret over til en træstruktur som vist i figur 8.6. I strukturen benyttes klassen *DartProgram* og formålet med denne er at repræsentere det program brugeren ønsker at køre på en node på netværket.



Figur 8.6: *Simplificeret træstruktur over afhængigheder uden dubletter.*

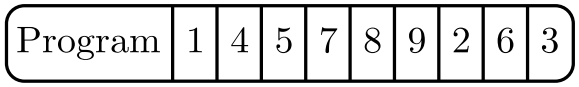
Et *DartProgram* objekt oprettes ud fra en eksisterende *FileNode* eller *DependencyNode*. Under konstruktionen af objektet traverseres hele træet af

afhængigheder og gemmes i et *Set* således ens objekter bliver fjernet. Resultatet er herefter en instans af *DartProgram* som beskrevet i figur 8.7.



Figur 8.7: *Klassediagram over klassen DartProgram som nedarver fra DependencyNode.*

Ved efterfølgende at tjekke afhængighederne til *DartProgram* objektet fås en liste som vist i figur 8.8 hvor der ikke længere optræder dubletter af afhængigheder.



Figur 8.8: *Liste over afhængigheder bygget ud fra figur 8.4 som efterfølgende er konverteret til et DartProgram objekt. Der er ikke længere nogen dubletter i forhold til 8.5.*

DartProgram objektet repræsenterer nu det program der ønskes oprettet og indeholder samtidig information omkring alle afhængigheder for programmet. En vigtig detalje omkring *FileNode* objektet er at selve objektet ikke indeholder kildekoden men derimod kun information omkring placering og checksum. Begrundelsen for dette designvalg er beskrevet i næste afsnit.

8.2 Overførsel af kildekode over netværk

Efter oprettelse af *DartProgram* objektet sendes det over netværket til den node hvor programmet ønskes at blive kørt på. Dette er her illustreret i figur

8.9 hvor *Klient* er det system der står med et program der skal startes på *Server*.

Figur 8.9: *Trin 1 - DartProgram sendes til Server.*

For dette er muligt er det nødvendigt at konvertere *DartProgram* objektet om til en tekststreng der kan sendes over netværket. I Dart anbefales det at gøre dette ved brug af JSON formatet og ved brug af standardbiblioteket *dart:json*. Biblioteket indeholder en global funktion *stringify* der konverterer følgende objekter til JSON:

- *num*, *String*, *bool*, *null*.
- *List* objekter hvis alle elementer i listen kan konverteres til JSON.
- *Map* objekter hvis alle nøgler er af typen *String* og alle værdier kan konverteres til JSON.
- Objekt der implementerer *toJson* metoden. Metoden kaldes automatisk og resultatet skal være muligt at konvertere til JSON (kan f.eks. godt returnere et andet objekt der også har en *toJson* metode).

Som det kan ses i klassediagrammet i figur 8.7 så implementerer *DartProgram* netop *toJson* metoden og gør det derfor muligt at repræsentere objektet som en JSON streng. Ligeledes er der lavet en *factory constructor* med navnet *fromJson* der tager imod en JSON streng for at genopbygge objektet.

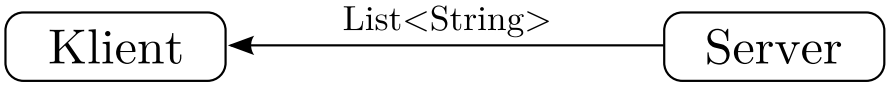
Et eksempel på en JSON streng oprettet ud fra et *DartProgram* objekt kan ses i kodeeksempel 8.3.

```
1  {
2    hash: [85, 120, 2, 22, 179, 96, 78,
3      33, 221, 31, 191, 215, 237, 250, 66,
4      247, 113, 12, 5, 124
5    ],
6    name: helloworld.dart,
7    path: helloworld.dart,
8    dependencies: [{
9        hash: [57, 74, 242, 255, 127, 201,
10         23, 193, 159, 103, 208, 113,
```

```
11          240, 143, 67, 249, 41, 219, 104,
12          188
13        ],
14        name: sun.dart,
15        path: lib / sun.dart
16      }
17    ]
18 }
```

Kodeeksempel 8.3: *Eksempel på hvordan Dart behandler bool-værdier i forhold til JavaScript.*

Som det kan ses i JSON strengen så modtager *Server* ikke nogen filer men derimod kun informationer omkring filerne. Årsagen til dette valg er at det teoretisk er muligt at *Server* skulle ligge inde med filerne i tilfælde af den før har været brugt til at køre det samme kode. I tilfælde af en eller flere filer mangler sendes der en liste af *String* tilbage til *Klient* med checksum for de filer der mangler. Dette er illustreret i figur 8.10.



Figur 8.10: *Overordnet model over arkitekturen ved et distribueret system der arbejdes med i dette projekt.*

Klient har samtidig en cache over hvilke filer der tidligere har været indlæst og kan slå op i denne ud fra en checksum. Herefter returneres der et *Map* hvor nøglen er checksummen og værdien er en *List* af *int* der repræsenterer indholdet af filen som vist i figur 8.11.

Figur 8.11: *Overordnet model over arkitekturen ved et distribueret system der arbejdes med i dette projekt.*

8.3 Oprettelse af miljø ved modtager

Den sidste del af processen er at gemme data fysisk på disken ved *Server* således det er muligt at køre *spawnUri* metoden og oprette en *isolate* ud fra koden. Kriterierne for filstrukturen er mulighed for at kunne cache forrige

tilfælde af kode der skal køres således der ikke skal flyttes det samme indhold gentagende gange.

Mappestrukturen der er valgt er følgende og er som standard gemt i mappen ".distributed_dart_data" der ligger i arbejdskataloget for det kørende Dart program der tager imod *requests*:

.distributed_dart_data / hashes
: Mappen indeholder en fil for hver eneste fil der er blevet hentet ned. Filnavnet for hver fil svarer til SHA1 checksum der er konverteret til et filnavn som f.eks: 1858880cf575c21482e7a1b3ebb4165950c2f7f3.dart

  Fordelen ved denne metode er at det er hurtigt at slå op i mappen og finde ud af om der findes en fil med den checksum der efterspørges. Samtidig er det ligegyldigt at det originale navn er smidt væk fordi navnet står i det *request* der modtages sammen med hvilken checksum filen har.

.distributed_dart_data / isolates
: Her oprettes et miljø af filer der er placeret således det er muligt at bruge *spawnUri* metoden på den fil der indeholder *main*. Selve filstrukturen minder om "hashes" mappen ved at der oprettes en mappe for hver unik sammensætning af filer:

  - 0a27e3473cb67eee79f002d5ac7d09387750fb50 / helloworld.dart
  - 0a27e3473cb67eee79f002d5ac7d09387750fb50 / lib / sun.dart

  SHA1 summen den bruges som mappenavn er genereret ud fra en række parametre med blandt andet summen af de forskellige checksums fra hver fil. Dette er gjort for at sikre så vidt muligt at der dannes en ny mappe i tilfælde af en af filerne er ændret. Omvendt, hvis et miljø er komplet ens som et tidligere tilfælde er der intet i vejen for at genbruge dette.

For at undgå at spilde unødvendig lagerplads med mange kopier af de samme data er der benyttet links oprette filerne i "isolates" mappen således filerne ikke kræver det dobbelte i plads. For at oprette disse links er der benyttet den indbygget *Link* klasse fra "dart:io".

En vigtig detalje omkring *Link* klassen er den har forskellige funktionalitet alt efter hvilket operativsystem der kører Dart programmet. Ved oprettelse af et link står der følgende i dokumentationen for Windows:

"On the Windows platform, this will only work with directories, and the target directory must exist. The link will be created as a Junction. Only absolute links will be created, and relative paths to the target will be converted to absolute paths.[9]"

Det er altså ikke muligt at bruge *Link* klassen på Windows til at oprette links mellem individuelle filer som der netop er behov for i *distributed_dart* for at kunne implementere den føromtalte filstruktur. Derimod er der ingen problemer på Unix baseret systemer da symlinks godt kan pege på individuelle filer:

"On other platforms, the posix symlink() call is used to make a symbolic link containing the string target. If target is a relative path, it will be interpreted relative to the directory containing the link.[9]"

Forklaringen skal findes at det ikke er tilladt i Windows at oprette symlinks mellem individuelle filer uden ekstra rettigheder på systemet [5]. Derimod er det muligt at oprette symlinks mellem mapper (junctions) men også tilladt at oprette *hardlinks* mellem filer uden yderligere rettigheder (men virker ikke på Windows XP).

Oprettelse af *hardlinks* kan ikke lade sig gøre igennem Dart standard API så derfor er der udviklet metoden som ses i kodeeksempel 8.4. Metoden fungerer ved et simpelt tjek hvilket OS der benyttes og hvis der benyttes Windows vil der forsøges at oprettes en ny proces der kører *mklink* kommandoen passende parametre.

Efter oprettelse af de forskellige links i "isolates" mappen kan der returneres en URI der peger på den fil der skal startes med *spawnUri* metoden.

```
static Future createLink(Path source, Path destination) {
  Completer c = new Completer();

  if (Platform.operatingSystem == "windows") {
    List<String> arguments = ["/C",
                              "mklink",
                              "/H",
                              destination.toNativePath(),
                              source.toNativePath()];

    Process.run("cmd", arguments).then((ProcessResult result) {
      if (result.stderr.isEmpty) {
        _log("Link succesfully created.");
        c.complete();
      } else {
        String error = "Could not create link.";
        OSError osError = new OSError(result.stderr,
                                      result.exitCode);
        throw new FileIOException(error, osError);
      }
    });

  } else {
    StringBuffer sb = new StringBuffer();
    for (int a = 1; a < destination.segments().length; a++) {
      sb.write("../");
    }

    Path releativeSource = new Path(sb.toString() +
                                    source.toString());

    Link link = new Link.fromPath(destination);
    link.create(releativeSource.toString()).then((_) {
      c.complete();
    });
  }

 return c.future;
}
```

Kodeeksempel 8.4: *Metode der bruges til at oprette links mellem to Path objekter. Metoden bruger hardlinks på Windows platformen mens der bruges symlinks ved alle andre platforme.*

Kapitel 9

Konklusion

Dette kapitel indeholder konklusionen på projektet *distribueret_dart* og tager udgangspunkt i problemformuleringen kapitel 2.

Projektet startede ud med en problemstilling der blev opdaget ved forundersøgelsen til specialet og som omhandlede mangel på muligheder for at oprette isolate distribueret. Ud fra denne problemstilling er der blevet undersøgt forskellige muligheder for at implementere distribueret isolates i Dart.

Produktet der er udviklet under projektet er navngivet *distribueret_dart* og består af et samlet programbibliotek til Dart der skal gøre det muligt at kunne oprette isolates i et distribueret miljø. Selvom produktet ikke er færdigudviklet og fortsat mangler større dele af netværkslaget er det stadig muligt at besvare de spørgsmål der blev opstillet i problemformuleringen ud fra den analyse der er blevet foretaget under projektet.

Til projektet blev der opstillet tre konkrete spørgsmål med det formål at besvare dem i løbet af projektet. Svaret på spørgsmålene er følgende:

Kan det lade sig gøre, at udvikle et programbibliotek til sproget Dart, der understøtter en distribueret programmeringsmodel, uden at ændre i Dart VM?

Ja, det er muligt at lave en distribueret programmeringsmodel uden at foretage ændringer i Dart VM'en. Selvom *distribueret_dart* ikke på nuværende tidspunkt er færdigudviklet så er der intet der peger i retningen af det ikke kan lade sig gøre at gøre færdigt. De dele af *distribueret_dart* der har skabt flest problemer har været omkring scanning efter afhængigheder til et givent Dart program samt flytning af programdata over til det miljø hvorfra koden skal køres hvilket er implementeret på nuværende tidspunkt i *distribueret_dart*.

Samtidig skal det tilføjes at der under projektet er fundet frem til følgende problemstillinger der ville kræve ændringer i Dart VM for at kunne blive løst:

Status på isolates er ikke mulige og kræver at hver isolate selv implementere en form for statusmeddelelse. Problemet opstår når der skal forsøges at registrere om en given isolate er lukket ned. Eftersom det ikke er muligt at spørge en given *SendPort* om der er hul igennem er det ikke muligt at vide om en given isolate blot arbejder på noget kompliceret og derfor ikke har tid til at svare eller om den er lukket ned.

Desuden er det kun muligt at lukke en isolate ned hvis det gøres inde fra selvsamme isolate. Dette gør det ekstra besværligt at håndtere mængden af isolates der kører i baggrunden på et givent system eftersom det ikke er muligt at tvinge en isolate ned.

Opret isolate ud fra *String* vil kunne gøre det væsentligt nemmere at sende koden til en isolate over netværket. Problemet som det er lige nu er at det skal gøres fra filer og disse skal være tilgængelige for operativsystemet. En mulighed er at bruge netværksdrev men det er stadig ikke ret fleksibelt.

Disse problemstillinger er ret så væsentlige og især problematikken omkring manglende status på isolates kan gøre det besværligt at bruge i en distribueret sammenhæng.

Er det muligt at overholde semantikken for den måde Dart i forvejen understøtter flertrådet programmering?

Ja, fordi den semantik har for flertrådet programmering er brug af *message-passing* samt asynkrone kald til en lang række operationer. Fordi semantikken i forvejen understøtter at operationer kan køre i baggrunden og efterfølgende kalde en *callback*-metode eller smide exceptions.

Udviklingen af *distribueret_dart* har derfor været at udvikle en løsning der passer direkte ind i den måde Dart i dag bruger isolates og gøre netværksdelen gennemsigtig og enkel med henblik på at kunne kommunikere med *SendPort* objekter uden udvikleren nødvendigvis tænker over det foregår over et netværk.

På trods af netværksdelen endnu ikke er færdigudviklet er der ikke fundet noget i analysefasen der skulle forhindre at udvikle *distribueret_dart* til at overholde Dart semantikken for flertrådet programmering.

Hvilke begrænsninger opstår, når Dart semantikken for flertrådet programmering, skal overholdes?

Den største begrænsning ved semantikken for flertrådet programmering i Dart er der på ingen måde tages højde for datatab fordi det ikke er realistisk når isolates alligevel kører på samme maskine og CPU. Problemet gælder også i visse tilfælde omkring fejl. Især ved *spawnUri* er det problematisk at det ikke er muligt at modtage information ved ikke behandlet exceptions eller om en given isolate i det hele taget er lukket ned.

I samme ånd er det heller ikke muligt at angive timeouts ved afventning af svar ved isolates. Hvis en isolate laver en operation så kan den ikke stoppes på nogen måde med mindre operationen er udviklet til ikke at køre konstant men køre i en række små events. Kombineret med manglen på tvunget nedlukning af isolates er der en række problemer der bør løses omkring isolates før kan betragtes til at fungere på et passende niveau for et distribueret miljø.

En anden begrænsning er det ikke er muligt at sende clojures til en isolate. Under projektet er der fundet frem til mange fordele og ulemper omkring dette og årsagen til det ikke kan lade sig gøre er højst sandsynligt det store scope der kan risikeres at følge med en clojure.

Afrunding

Med undtagelse af visse problemstillinger samt manglende implementering af netværkslaget konkluderes der at *distribueret_dart* vil kunne lade sig gøre og fremtiden for et lignende projekt ser lovende ud. Programbiblioteket *distribueret_dart* skal ikke ses som en endelig løsning der gør det muligt at løse alle problemer indenfor distribueret programmering. Derimod skal biblioteket ses som et fundament der gør det relativt simpelt at oprette en isolate på en node i et netværk baseret på kode der kommer fra en anden node. Efter oprettelsen er det enkelt at kommunikere med den oprettede isolate og udvikleren skal ikke koncentrere sig om der ligger et netværkslag imellem.

Projektet blev udviklet med den begrænsning at løsningen skulle så vidt muligt kunne laves i ren Dart hvilket kan konkluderes at være muligt.

Litteratur

[1] Bo Hedegaard Andersen, Brian Astrup Mikkelsen, and Jacob Bang. Evaluating the dart programming language. Technical report, Department of Computer Science, Aalborg Ø, 2012-2013. URL http://goo.gl/3J92M.

[2] Gilad Bracha. Reflection in dart with mirrors: An introduction, Maj 2013. URL http://www.dartlang.org/articles/reflection-with-mirrors/.

[3] Seth Ladd Kathy Walrath. *Dart: Up and Running*. O'Reilly Media, 2012. ISBN 978-1449330897.

[4] Seth Ladd. Dynamically load code with dart, April 2013. URL http://blog.sethladd.com/2013/04/dynamically-load-code-with-dart.html.

[5] Microsoft. Mklink, April 2012. URL http://technet.microsoft.com/en-us/library/cc753194%28WS.10%29.aspx.

[6] Bob Nystrom. Dart style guide - members, Februar 2013. URL http://www.dartlang.org/articles/style-guide/#members.

[7] Dart project. dart:async library, Juni 2013. URL http://api.dartlang.org/docs/releases/latest/dart_async.html.

[8] Dart project. dart:isolate library, Juni 2013. URL http://api.dartlang.org/docs/releases/latest/dart_isolate.html.

[9] Dart project. dart:io - link, Juni 2013. URL http://api.dartlang.org/docs/releases/latest/dart_io/Link.html.

[10] Dart project. dart:mirrors library, Juni 2013. URL http://api.dartlang.org/docs/releases/latest/dart_mirrors.html.

[11] The Dart Team. The dart programming language specification, June 2013. URL http://www.dartlang.org/docs/spec/latest/dart-language-specification.pdf.

[12] L. Thamsen, A. Gulenko, M. Perscheid, R. Krahn, R. Hirschfeld, and D.A. Thomas. Orca: A single-language web framework for collaborative

development. In *Creating, Connecting and Collaborating through Computing (C5), 2012 10th International Conference on*, pages 45–52, 2012. doi: 10.1109/C5.2012.9.

[13] Kathy Walrath and Seth Ladd. Dart: Up and running - built-in types - booleans, Juni 2013. URL `http://goo.gl/Ug596`.

[14] Kathy Walrath and Seth Ladd. Dart: Up and running - dart:isolate - concurrency with isolates, Juni 2013. URL `http://goo.gl/QUbsV`.

Akronymer

EBNF Extended Backus–Naur Form

GWT Google Web Toolkit

IDL Interface Definition Language

WSDL Web Service Definition Language

IDL Interface Description Language