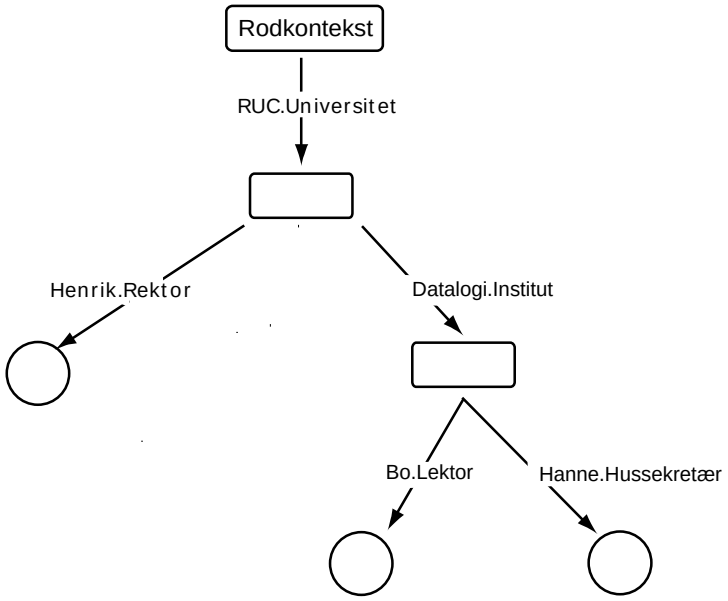
Roskilde Universitetscenter - Datalogi OB

Implementation af en CORBA-navneservice

(Implementation of a CORBA-naming service)



En 2. moduls projektrapport af:

Benjamin Beyer, Kasper G. Christensen og Lars Schjøth

gruppe 29, Hus 42

Vejleder:

Henrik Høltzer

Foråret 2002

Abstract

Dette projekt omhandler implementering af OMG's CORBA-navneservice. Første halvdel af rapporten beskriver CORBA og navneservice teori, mens sidste halvdel beskriver og diskuterer en specifik implementering af en CORBA-navneservice. Implementeringen er efterstræbt at opfylde CORBA-navneservice v. 1.1 specifikationen. Vores problemformulering lyder derfor således:

At implementere en persistent CORBA-navneservice, der opfylder kravene i CORBA-navneservice 1.1 specifikationen [OMG; 2001ns] .

Funktionaliteten af vores implementation testes, og afslutningsvis konkluderer vi, at vi har implementeret en persistent CORBA-navneservice, der opfylder kravene i specifikationen på alle punkter undtaget implementation af metoden `to_url` i `NamingContextExt`-interfacet.

Vores målgruppe er personer med interesse for programmering af distribuerede systemer. Specielt med henblik på personer med interesse for navneservicer og CORBA.

English abstract

This project is about the implementation of OMG's CORBA-naming service. First part of this thesis describes CORBA and naming service theory, while the last part describes and discusses a specific implementation of a CORBA-naming service. In making the implementation we have tried to live up to the CORBA-naming service v. 1.1 specification. In short our problem sounds like this:

To implement a persistent CORBA-naming service that lives up to the CORBA-naming service v. 1.1 specification [OMG; 2001ns].

The functionality of our implementation is then tested and finally we conclude that we have succeeded in implementing a persistent CORBA-naming service. This implementation lives up to the specification in all but one account, which is the implementation of the method `to_url` from the interface `NamingContextExt`.

This thesis is meant for people who have an interest for programming distributed systems. It is especially for people with an interest for naming services and CORBA.

Indholdsfortegnelse

1 Indledning

Denne rapport beskriver en specifik implementering af OMG's CORBA-navneservice. Den er skrevet som en introduktion til CORBA og dens navneservice. Første halvdel af rapporten omhandler således både CORBA og navneservice teori, mens sidste halvdel af rapporten beskriver og problematiserer en specifik implementering af en CORBA-navneservice. Implementeringen er efterstræbt at opfylde CORBA-navneservice v. 1.1 specifikationen. De designproblemer, denne implementering fremviser, beskrives rapporten igennem. Vores problemformulering lyder:

At implementere en persistent CORBA-navneservice, der opfylder kravene i CORBA-navneservice 1.1 specifikationen [OMG; 2001ns] .

Kort fortalt er CORBA en standard for, hvad der kaldes middleware. Middleware er et softwarelag, der tilbyder programmører af distribuerede systemer en abstraktion fra underliggende netværk. En programmør kan benytte middleware til kommunikation mellem delene af et distribueret system. Middleware er således et softwarelag, der implementerer kommunikation på et højere abstraktionsniveau for derved at gøre implementation af distribuerede systemer lettere.

CORBA er en standard defineret af Object Management Group [OMG; 2002], og CORBA-standarden er specificeret således, at den tillader forskellige implementationer at kommunikere sammen. Modsat det meste andet middleware er CORBA sproguafhængigt. Dette muliggøres ved, at CORBA benytter sit eget sprog, kaldet IDL, til at definere grænsefladerne mellem de enkelte dele af en CORBA-applikation. Dette og mange andre begreber vil blive beskrevet i rapporten.

Det andet spændende koncept, denne rapport behandler, er navneservicer. En navneservice tilbyder brugeren et redskab til at katalogisere og navngive ressourcer som e-mail adresser, websider eller fjernobjekter. Dette system gør det muligt at lokalisere en ressource i et distribueret system udfra et menneskeligt let forståeligt navn.

Implementering af en navneservice byder på en række interessante problemer, som vil blive diskuteret rapporten igennem. Blandt andet kan nævnes datarepræsentation, samtidighedskontrol og skalerbarhed.

Vores målgruppe er personer med interesse for programmering af distribuerede systemer. Specielt med henblik på personer med interesse for navneservicer og CORBA.

2 Teori

I de følgende afsnit beskrives den grundlæggende teori, som vi finder nødvendig, for at kunne forstå og implementere en CORBA-navneservice.

2.1 CORBA-arkitektur

CORBA står for Common Object Request Broker Architecture og er en standard for distribuerede, objektorienterede systemer. CORBA-Standarden beskriver en fælles arkitektur, der gør det muligt, for objekter i forskellige sprog og på forskellige maskiner, at kalde metoder på hinanden og derved at indgå i et distribueret system. CORBA-standarden udvikles af OMG (Object Management Group).

I de følgende afsnit vil grundlæggende CORBA-teknologi blive beskrevet. Først beskrives den overordnede arkitektur, hvorpå der gås i dybden med udvalgte dele af denne.

2.1.1 Overordnet arkitektur

CORBA-standarden består af flere dele, herunder:

- CORBA/IIOP
- IDL og dertilhørende Language mappings
- CORBAServices
- CORBAFacilities
- CORBADomains

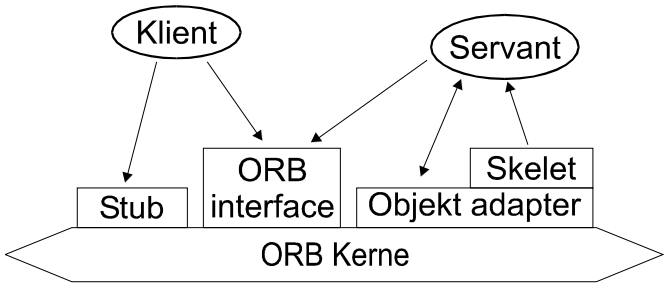
I dette afsnit beskæftiger vi os først og fremmest med den grundlæggende CORBA/IIOP specifikation [OMG; 2001a], der er den centrale del af CORBA-standarden, som de øvrige dele bygger på. CORBA/IIOP har til opgave at gøre metodekald mellem objekter, implementeret i forskellige sprog og på forskellige maskiner, muligt. En central del af dette er ORB'en (Object Request Broker). En ORB kan opfattes som en databus, der transporterer metodekald, samt disses returværdier, mellem objekter. Et CORBA-Objekt tilknyttes en bestemt ORB, som den benytter til at kommunikere med andre CORBA-Objekter. For at gøre det muligt at kommunikere med objekter tilknyttet andre ORB'er, findes der i CORBA-standarden protokoller, der benyttes til kommunikation mellem ORB'er. En af disse er IIOP, hvilket står for Internet Inter ORB Protokol, som er en standardiseret protokol, der benyttes til kommunikation mellem forskellige ORB'er over Internettet. Kommunikation mellem ORB'er vender vi tilbage til i afsnit: **2.1.4 Interoperabilitet**. Bemærk, at en ORB er en abstraktion, og altså ikke nødvendigvis en fysisk entitet, der kan implementeres på mange måder (f.eks. oven på forskellige kommunikationsprotokoller, som en særskilt proces eller som biblioteks-kode). Disse implementeringsspørgsmål vil vi ikke beskæftige os med i dette projekt. Vi vil i stedet koncentrere os om den overordnede arkitektur, og dermed de interfaces ORB'en stiller til rådighed.

For at opnå sprog-uafhængighed, er det nødvendigt med et fælles sprog til definering af objekt-interfaces. Til dette formål benyttes CORBA-IDL (Interface Definition Language), for hvilket der er defineret en lang række "mappings" til konkrete programmeringssprog. En mapping er således en angivelse af, hvordan de enkelte dele af IDL svarer til sprog-konstruktionerne i et givent programmeringssprog. IDL's rolle i CORBA-arkitekturen vil vi vende tilbage til i dette afsnit, og udvalgte dele af IDL-semantik og -syntaks vil blive beskrevet i afsnit **2.1.2 IDL**.

Udover den grundlæggende CORBA/IIOP specifikation, findes der standardiserede interfaces og tilhørende semantik-beskrivelser, for en række generelt anvendelige CORBA-Objekter. Disse er grupperet i Services, Facilities og Domains. Services er generelle objekt-services, der ofte vil indgå i applikationsudvikling, som for eksempel en navneservice og en transaktionsservice. Facilities er mere specifikke objekter, dog stadig med en generelt anvendelig funktionalitet, dette kunne for eksempel være objekter til afsendelse af e-mail. Domains er domæne-specifikke standarder, som for eksempel en standard for finansielle transaktioner og en til medicinalindustrien. I dette projekt beskæftiger vi os ikke med hverken Domains eller Facilities, og vi beskæftiger os udelukkende med CORBA NameService og ikke med nogle af de øvrige Services.

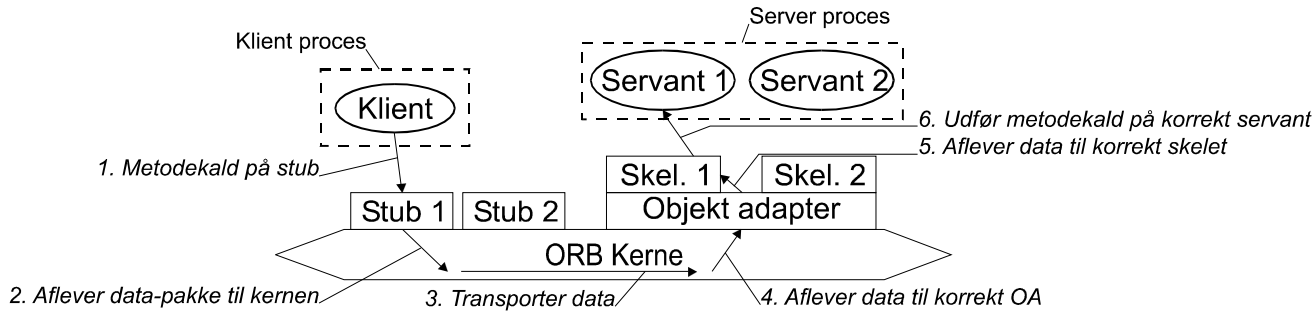
I CORBA-standarden findes beskrivelser af dynamiske interfaces. Disse interfaces er beregnet til applikationer, hvor det er ønskeligt at kunne finde objekter, hvis typer ikke var kendt på kompileringstidspunktet, og kalde metoder på disse objekter. Dette betegnes "dynamisk CORBA". Vi har valgt ikke at beskæftige os med dette i denne rapport, da det ikke direkte berører navneservicen og for at undgå unødig kompleksitet. Derfor vil dynamisk CORBA ikke blive omtalt yderligere.

På **Figur 2.1** ses en overordnet illustration, af den grundlæggende CORBA ORB-Arkitektur.



Figur 2.1: Grundlæggende CORBA ORB-arkitektur

Servanten er en konkret implementation af et CORBA-Objekt. Klienten kalder metoder på dette objekt gennem ORB'en. ORB'en fungerer ved hjælp af stubbe på klientsiden, gennem hvilke klienten foretager metodekald. Metodekaldene transformeres af stubben til et format, der, via kernen, kan transporteres til den korrekte objekt-adapter (der kan eventuelt være flere objekt-adaptere). Objekt-adapteren fungerer som dispatcher og videregiver altså metode-kaldet til det relevante skelet, der udpakker transport-pakken og kalder implementationen på serversiden (servanten). Dette dataflow er illustreret på **Figur 2.2.**



Figur 2.2: Udførsel af et metodekald på en CORBA-ORB, hvor der er to tilgængelige CORBA-Objekter. Returnering af eventuel returværdi illustreres ikke, men foregår på samme måde som metodekaldet (klienten og serveren bytter groft sagt roller).

Stubben er et objekt implementeret i det sprog, som klienten er implementeret i. Stubben implementeres ikke af klient-programmøren, men genereres af en kompiler – dette beskrives senere i afsnittet. Stubbens opgave er at modtage metodekald fra klienten, og videreformidle disse til en konkret implementation af det objekt stubben repræsenterer. Klienten benytter stubben som ethvert andet objekt. Dermed opnås lokations-gennemsigtighed for klienten, der ikke umiddelbart kan se, om det er et fjernt, eller et lokalt objekt den benytter – da den blot kalder metoden på stubben. Yderligere muliggør brugen af stubbe implementeret i klient-sproget, at CORBA-Objekter i et givent sprog benyttes på samme måde som alle andre objekter i sproget. Eneste forskel for klienten er, at et metodekald på et CORBA-Objekt kan kaste nogle CORBA-relaterede undtagelser, for eksempel ved kommunikationsfejl eller hvis det fjerne objekt er fjernet fra ORB'en. Når stubben modtager et metodekald, transformerer den dette til en datapakke, som kan transporteres via ORB-kernen. Derpå overlader den det til kernen, at aflevere pakken til den objekt-implementation, der skal udføre metodekaldet.

Objekt-adapteren har til opgave at fungere som dispatcher, det vil sige videregive metodekald til de rette skeletter (og dermed objekt-implementationer). Enhver servant skal således registreres hos en objekt-adapter, før en klient kan kalde metoder på den. Der kan være flere aktive objekt-adaptere på samme tid. Kernen sørger for at transportere en datapakke, afsendt af en stub, til den objekt-adapter, servanten er registreret på. I CORBA-Standarden findes en beskrivelse af interfacet til en objekt-adapter kaldet POA (Portable Object Adapter). Brug af en POA sikrer, at koden kan flyttes til en hvilken som helst ORB, der har implementeret POA-standarden (koden er portabel). En POA kan, udover at fungere som dispatcher, også håndtere forskellige strategier for aktivering af objekter. Således behøver et objekt ikke altid at være til stede i hukommelsen, men kan gemme på disk, og kan da aktiveres af POA'en, når en metode på objektet forespørges. Yderligere uddybning af POA'ens funktionalitet, findes i afsnit **2.1.2 IDL**.

Skelettet er et objekt på serversiden, skrevet i samme sprog som servanten, og modsvarer stubben på klient-siden. Skelettet implementeres ikke af servant-programmøren, men genereres, ligesom stubben, af en kompiler (dette beskrives senere i afsnittet). Skelettets opgave er at modtage data fra objekt-adapteren for derpå at pakke datapakken ud og foretage metodekaldet på den servant, som skelettet repræsenterer. Derved kommer metodekaldet for servanten til at

se ud, nøjagtigt som hvis det var foretaget lokalt. Skelettet er altså en form for bindeled mellem den generelle objekt-adapter og den specifikke servant.

ORB interfacet (illustreret på **Figur 2.1**) er et pseudoobjekt, som giver både klient og servant adgang til en række metoder til for eksempel konvertering mellem forskellige repræsentationer af objektreferencer, samt bootstrapping og initialisering af ORB'en. Bootstrapping er en proces med det formål at skaffe en applikation de første objekt-referencer til CORBA-Objekter for dermed at gøre det muligt for applikationen at foretage de første CORBA-kald. Bootstrapping og initialisering beskrives nærmere i afsnit **2.1.5 Bootstrapping og initialisering af ORB'en**

Stubbene og skeletterne skal samarbejde tæt med henholdsvis klient og servant, der implementeres af en applikations-programmør i et valgt sprog. Da klienten og servanten skal kunne implementeres i forskellige sprog, kræves det, at det skal være muligt at fremskaffe for eksempel en stub til en given servant i et valgt sprog. For at gøre dette muligt defineres alle servanter's interfaces i CORBA-IDL. Ud fra en sådan IDL-specifikation kan en, til ORB'en hørende, IDL-Compiler generere de relevante stubbe og skeletter i de ønskede sprog. Dette betyder naturligvis, at en ORB må levere en eller flere compilere, der understøtter de sprog, som ORB'en kan benyttes med. CORBA-IDL er en vigtig del af CORBA-Standarden, da det er hjørnestenen i sprog-uafhængigheden.

I de følgende afsnit gås i dybden med udvalgte dele af CORBA-arkitekturen.

2.1.2 IDL

I denne del af rapporten vil IDL (Interface Definition Language) kort blive beskrevet. Kort fordi der ikke vil blive gået mere i dybden med det, end at det er muligt at forstå den IDL der benyttes til at implementere en navneservice.

På det overordnede plan er IDL opbygget af to dele, moduler og interfaces. Et modul er til for at lave en logisk modularisering af koden. Den kode man samler i modulerne er, blandt andet, men væsentligst, interfaces (de relevante dele forklares senere i kapitlet). Interfaces er, som navnet hentyder til, interfaces til objekter og er derfor grænseflader til funktionel kode. IDL-interfaces kan indeholde definitioner af nye typer. [IDL; 2002] Et eksempel på IDL-syntaks kan ses i nedenstående stykke eksempel-kode.

```
module Auction {
      interface Trade {
            void Sell(in Person seller, in Person buyer, in short price, inout Item soldItem)
                  raises(NotAPerson, InvalidPrice, NotForSale);
      };
};
```

IDL-syntaksen for interfaces ligner i vid udstrækning andre programmeringssprog (som f.eks. Java eller C++). I IDL skal det defineres, om en parameter er `in`, `out` eller `inout`. Disse tre parametre til parametre definerer, om en metode-parameter enten benyttes som input til metoden, som output fra metoden eller som både input og output, se afsnit **2.1.2.1 Java Language mapping detaljer**. En anden egenskab er, at man på interface-niveau, skal definere hvilke undtagelser, en metode kan forventes at kaste. At dette er vigtigt at gøre skyldes, at det skal være muligt, via CORBA, at fange disse undtagelser, og sende dem videre til den metode som forsøgte at benytte den metode som fejlede. Det er muligt i IDL selv at definere specifikke undtagelser. [IDL; 2002]

Det er muligt at definere nye typer i IDL – til brug i moduler, eller blot indenfor et enkelt interface. Dette kan foretages via en `typedef`. [IDL; 2002] Et eksempel på dette kan ses herunder.

```
typedef string nameString;
```

Det er også muligt at definere sammensatte typer. Nogle sammensatte typer, som indgår i en navneservice og derfor er relevante, er `struct`, `enum` og `sequence`. Den førstnævnte, structen, er en samling af oplysninger af vilkårlig type.

```
struct PersonStruct {
    boolean       isAMan,
    short         age,
    nameString    name,
    long          ageInSeconds
};
```

En `enum` definerer en type, der kun kan indeholde én ud af et sæt af allerede definerede værdier med samme type. [IDL; 2002]

```
enum GenderEnum {
    Male,
    Female
};
```

Sequences er dynamiske lister af værdier af samme type. Den kan dog også indeholde værdier, der nedarver fra den valgte type. [IDL; 2002]

```
sequence<Object> AuctionInventory;
```

2.1.2.1 Java Language mapping detaljer

Når IDL-interfaces kompileres, gøres det i forhold til et givent sprog, som programmøren ønsker at implementere servanterne i. Der kompileres altså i forhold til en bestemt såkaldt language mapping, som oversætter IDL-typer – som `enum`, `struct`, `short`, og så videre, til tilsvarende typer i det sprog man ønsker at arbejde i. Disse language-mappings er, lige som resten af CORBA-standarden, fastlagt af OMG.

Java language-mapping angiver for eksempel, at sequences skal repræsenteres som Java-arrays, og at structs skal repræsenteres som klasser med public-felter, der svarer til felterne i `struct`'en, en konstruktor, der ikke tager nogle parametre, samt en konstruktor der tager alle felterne som parametre.

Som beskrevet i afsnit **2.1.2 IDL**, så kan man i IDL-interfaces sætte de tre parametre (til metode-parametre) `in`, `out` og `inout`. I Java er alle parametre in-parametre, og det er derfor nødvendigt, at have en måde hvorpå man kan repræsentere `out` og `inout` parametre – det vil sige parametre der, efter det metodekald hvori de blev brugt, indeholder nye oplysninger.

For at kunne opnå dette bliver der, når IDL-interfaces kompileres, genereret de såkaldte "holders", som skal bruges til at overføre oplysningerne. En `holder` er et objekt, som indeholder et felt af samme type som `out`-parameteren i IDL-interfacet. Man benytter så en `holder` som parameter i stedet for et objekt af den ønskede type. At dette er nødvendigt skyldes, at der ikke sker nogen ændring, hvis man forsøger at ændre den objekt-reference, der sendes med som parameter. I stedet kan man ændre på værdierne i det refererede objekt og altså også på feltet i holderen. I praksis benyttes denne konstruktion ved, at man først opretter et `holder`-objekt, som man så sender med, som parameter til den metode man benytter. Metoden kan da opbygge en passende returværdi i en lokal variabel (evt. objekt-variabel), og før den returnerer, lægges denne returværdi ned i feltet i `holderen`. Når metoden så er færdig, er det muligt at hente oplysningerne ud af holderen. Herunder ses et eksempel på benyttelse af `out`-parametre.

```
BindingListHolder bl = new BindingListHolder;
BindingIteratorHolder blIt = new BindingIteratorHolder;
ctx1.list(10, bl, blIt);
Binding[] bindings = bl.value;
```

Først oprettes to tomme holdere, en til en `BindingList` og en til en `BindingIterator`. Herefter kaldes metoden `ctx1.list` med disse som parametre. Efter kaldet til `list`, indeholder hver af holdernes `value`-felter en værdi genereret af metoden.

2.1.3 POA

POA står, som før nævnt, for "Portable Object Adapter" og er en konkret objekt-adapter, hvis interface og funktionalitet er specificeret i CORBA-Standarden [OMG; 2001b]. Da dens interface er fastlagt, kan kode, skrevet til at benytte POA'en på én ORB, direkte flyttes til en anden ORB uden nogen ændringer – koden er altså portabel, heraf "Portable".

I forbindelse med benyttelse af POA'en må man skelne imellem et CORBA-objekt, der er et abstrakt begreb, og en servant, der er en konkret entitet, der varetager metodekald på et CORBA-objekt. Det enkelte CORBA-objekt identificeres ved et objektID, der skal være entydigt på den enkelte POA. Dette objektID kan autogenereres af POA'en, når et ny objekt registreres, eller det kan angives af programmøren. Det er objektID'et, der afgør et CORBA-objekts identitet på POA'en, og det er dermed objektID'et, der benyttes af POA'en, når den rigtige servant skal vælges til at udføre et metodekald. POA'en tillægger ikke ObjektID'et nogen yderligere betydning og repræsenterer det som et array af bytes.

Da POA'en er en objekt-adapter, er dens første opgave at holde styr på hvilke CORBA-objekter, den har tilknyttet, og formidle metodekald til disse objekter. POA'en indeholder derfor en tabel over aktive objekter (identificeret ved deres objektID) samt disses tilknyttede servanter. POA'en har dermed indbygget et begreb om aktive og inaktive CORBA-objekter. Aktive CORBA-objekter er objekter, der er opført i tabellen over aktive objekter, og dermed har en aktiv servant tilknyttet. En servant er aktiv, når den eksisterer i hukommelsen og er klar til at modtage metodekald samt er tilknyttet et CORBA-objekt i tabellen over aktive objekter. Udover at fungere som objekt-adapter (altså videreformidle kald til servanter via skeletter), kan en POA konfigureres til at sørge for at starte servant'er op, såfremt der ankommer et kald til et objekt, der ikke p.t. er aktivt. POA'en starter servanter op ved hjælp af en ServantManager, der er et objekt, der tilknyttes POA'en. Det er muligt at deaktivere objekter på POA'en, hvilket medfører, at den tilknyttede servant kan nedlægges.

Det samme CORBA-objekt kan, i løbet af dets levetid, have forskellige servanter, der udfører dets metodekald. Dette kan for eksempel forekomme som resultat af, at objektet er blevet deaktiveret på POA'en, som så på et senere tidspunkt aktiverer objektet, fordi objektet da er blevet forespurgt.

Et objekt aktiveres på POA'en, ved kald af metoden:

```
ObjectId activate_object(in Servant p_servant)
```

Denne metode tager en reference til en servant som parameter, aktiverer servanten med et genereret objektID og returnerer dette ID. Hvis POA'en er konfigureret til at kunne benytte bruger-definerede objektID'er, kan et objekt aktiveres med et angivet objektID (der skal være unikt) ved at benytte metoden:

```
void activate_object_with_id(in ObjectId oid, in Servant p_servant)
```

Det faktum, at en POA kan konfigureres til at opføre sig på forskellig vis, gør det relevant at have mere end én aktiv POA. Der kan således være en POA, der aktiverer objekter, hvis de ikke er aktive, når de bliver kaldt. Objekter, der ønsker at implementere denne opførsel, kan så registrere sig på denne POA. Samtidig kan der være en anden POA, der kun benytter tabellen over aktive objekter, til at formidle metodekald. Derved forsøger denne POA ikke at aktivere et inaktivt objekt, hvis der kaldes metoder på det – i stedet kastes en `OBJECT_NOT_EXISTS` undtagelse. En ny POA oprettes ved hjælp af metodekaldet `create_poa` på en allerede eksisterende POA, og den nye POA er da "barn" til den POA, som create-kaldet blev foretaget på. Derved danner POA'erne et træ af objekt-adaptere. Når en POA nedlægges, nedlægges alle dens "børn" også. Den første POA, som alle andre POA'er nødvendigvis må udspringe fra enten direkte eller via mellemliggende "børn", oprettes af ORB-implementationen, og en reference til den skaffes via metodekaldet:

```
orb.resolve_initial_references("RootPOA")
```

En POA er altid tilknyttet en POA-manager. POA-managerens rolle er, at styre hvad der skal ske med indkomne forespørgsler til de POA'er, den er knyttet til. Den kan således enten videresende kaldene til POA'en, så de kan blive udført, den kan sætte kaldene i kø, eller den kan afvise dem. En POA-manager kan indstilles til at starte ikke-aktive POA'er, når de forespørges, og kan altså her udføre en tilsvarende rolle i forhold til POA'er, som POA'en kan udføre i forhold til servanter. Da en POA-manager derved kan have forskellige indstillinger og tilstande, kan det også være relevant at have flere managere.

Når CORBA-klienter ønsker at kontakte et objekt, sker dette gennem en CORBA-reference. En CORBA-reference kan, internt i ORB'en, repræsenteres implementationsspecifikt, men skal eksternt kunne repræsenteres som en IOR. En CORBA-reference må, blandt andet, indeholde identifikation, af den POA objektet befinder sig på samt objektID'et, for at muliggøre at metodekald bliver sendt til den rette servant. Da det i nogle implementationssituationer, kan være nødvendigt at konvertere mellem CORBA-referencer (der refererer til CORBA-objekter), servanter (dvs. referencer i implementeringssproget til de konkrete servant-objekter) samt objektID'er, findes følgende metoder på POA'en:

```
ObjectId servant_to_id(in Servant p_servant)
```

- Returnerer objektID'et for det CORBA-objekt den givne servant udfører metodekald for.

```
Servant id_to_servant(in ObjectId oid)
```

- Returnerer den servant, der er knyttet til det CORBA-objekt, der har det angivne objektID.

```
Object servant_to_reference(in Servant p_servant)
```

- Returnerer en CORBA-reference til det CORBA-objekt, som den angivne servant udfører metodekald for.

```
Servant reference_to_servant(in Object reference)
```

- Returnerer den servant, der er tilknyttet det angivne CORBA-objekt (dvs. der udfører dets metodekald)

```
ObjectId reference_to_id(in Object reference)
```

- Returnerer objektID'et for det angivne CORBA-objekt (angivet vha. en CORBA-reference)

```
Object id_to_reference(in ObjectId oid)
```

- Returnerer en CORBA-reference til det CORBA-objekt der har det angivne ObjektID

Disse metoder må nødvendigvis tage udgangspunkt i den POA, de kaldes på, og findes det angivne objektID/Servant/CORBA-Objekt ikke på denne POA, kan metoden ikke udføres meningsfuldt (og må derfor kaste en undtagelse).

2.1.4 Interoperabilitet

Da formålet med CORBA-standarden blandt andet er at gøre det muligt for forskellige programmører at skrive objekter i forskellige sprog og på forskellige platforme og samtidig give mulighed for, at disse objekter kan kalde metoder på hinanden, må CORBA tage højde for, at der også benyttes forskellige ORB'er. Dels kan man ikke regne med, at

forskellige programmører vælger et ORB-produkt fra den samme producent, og dels skrives ORB'er generelt ikke til at understøtte mere end nogle få sprog (idet der kun følger IDL-compilere med til disse udvalgte sprog). Derfor er det nødvendigt at specificere en protokol, der gør det muligt for forskellige CORBA-ORB'er at kommunikere med hinanden og derved at kalde metoder på hinandens objekter. [OMG; 2001c]

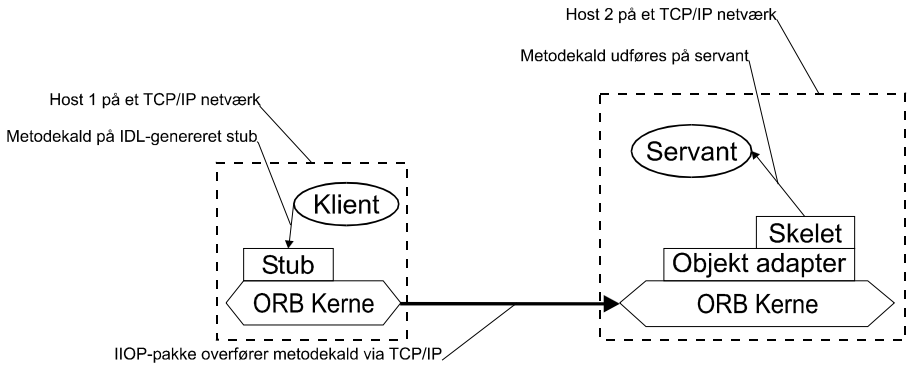
I CORBA-standarden er specificeret en generel protokol GIOP (General Inter ORB Protocol), der kan implementeres oven på et hvilket som helst netværks-transportlag og som muliggør, at ORB'er kan kalde hinandens objekter. Desuden specificeres det, at en CORBA-ORB skal implementere GIOP over internetprotokollerne TCP/IP. Denne konkrete implementation hedder IIOP (Internet Inter ORB Protocol) og muliggør, at alle CORBA-ORB'er kan udveksle objekt-referencer og kalde metoder på hinandens objekter via Internettet (eller et lokalt TCP/IP netværk).

En vigtig forudsætning for, at ORB'er kan kommunikere med hinanden, er at de data, der sendes via den fælles protokol repræsenteres på en standardiseret måde, således at alle ORB'er kan "forstå" indbyrdes beskeder. Denne fælles datarepræsentation kaldes CDR (Common Data Representation) og er en vigtig del af CORBA's interoperabilitets-arkitektur. [Mowbray; 1997]

En anden vigtig brik i interoperabilitets-arkitekturen er en fælles repræsentation af objekt-referencer. Hvis det skal være muligt for en ORB at videregive referencer til en anden ORB, må de to ORB'er nødvendigvis have en fælles standard for repræsentation af objekt-referencer. Denne problematik svarer nøjagtigt til behovet for den fælles standard for udveksling af andre data (CDR).

CORBA's standardiserede objekt-reference repræsentation hedder IOR (Interoperable Object Reference). Endvidere specificerer CORBA, hvorledes en IOR kan repræsenters som en streng. I ORB-interfacet findes en metode, der kan omsætte en objektreference til en sådan IOR-streng (`orb.object_to_string`), samt en anden metode der kan omsætte en IOR-streng til en objektreference (`orb.string_to_object`). [OMG; 2001d] Disse metoder gør det muligt for en server, via sin ORB, at omsætte en reference til en tekststreng, der for eksempel kan skrives i en fil. Denne fil kan derpå overføres til en klient, der kan læse filen og gendanne objektreferencen. Derpå kan klienten kalde metoder på server-objektet. En IOR kan naturligvis også udveksles på andre måder, for eksempel kan en metode i et objekt, returnere en IOR til et andet objekt. En IOR indeholder blandt andet adressen på den computer, på hvilken det refererede objekt befinder sig samt en entydig identifikation af objektet. En IOR identificerer et objekt entydigt på tværs af alle computere i verden, idet computerens adresse er entydig, og den medfølgende identifikation af objektet skal være entydig på den enkelte computer. I de fleste tilfælde genererer ORB'en en identifikation af objektet ud fra procesID og systemtid, men i nogle specielle tilfælde, kan det være nyttigt for programmøren at styre genereringen af denne identifikation, for eksempel for at opnå at et givent server-objekt får det samme ID, selvom serveren har været gået ned, og er blevet genstartet. Dette kaldes en persistent IOR, da referencen ikke ændrer sig, selvom servanten der håndterer metodekald på objektet, skiftes ud (f.eks. på grund af server-genstart).

Ved hjælp af IIOP og IOR er det altså muligt for to CORBA-ORB'er, at kommunikere med hinanden og dermed kalde metoder på hinandens objekter. **Figur 2.3** illustrerer et metodekald fra en klient på en ORB til en servant på en anden ORB.



Figur 2.3: Metodekald på tværs af ORB'er via IIOP. "Klient-ORB'en" kender adressen på "Server-ORB'en" idet denne er angivet i den CORBA-reference til server-objektet som klienten holder, og som den kalder metoden på.

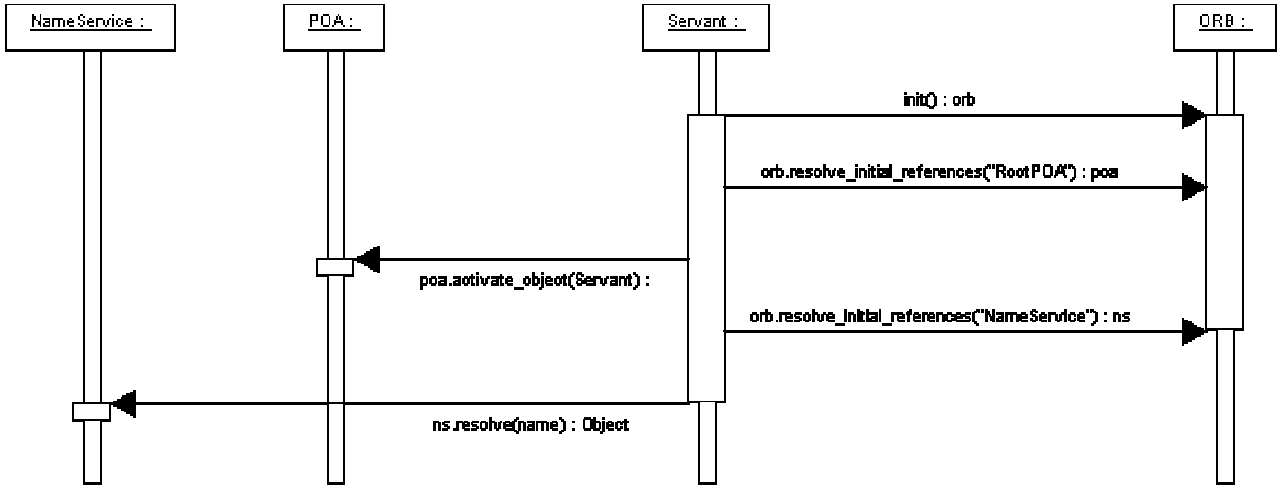
2.1.5 Bootstrapping og initialisering af ORB'en

Når en servant (altså en implementation af et CORBA-Objekt) oprettes, skal den initialiseres på en ORB for at være tilgængelig for CORBA-Klienter. Først skal servanten skaffe sig en reference til ORB'en. Dette gøres via den statiske metode `ORB.init`. `Init`-metoden initialiserer en ORB og kan tage en parameter, der navngiver den ORB, servanten

ønsker at benytte. Hvis ikke den navngivne ORB eksisterer, bliver den oprettet før `init`-kaldet returnerer. [OMG; 2001d] Nogle ORB'er understøtter kun oprettelse af én ORB (der skal altid angives et bestemt ORB-navn). `init`-kaldet returnerer en reference til ORB-interfacet (et pseudoobjekt). Derpå skal servanten registrere sig på en objekt-adapter (f.eks. en POA), hvorpå den er tilgængelig for omverdenen.

For at kunne registrere sig på objekt-adapteren, må servanten naturligvis have en reference til denne. Hvis POA er implementeret, kan en sådan reference fås ved at kalde `orb.resolve_intial_references("RootPOA")`. Her er "`orb`" den reference, til den initialiserede ORB, som `ORB.init` returnerer. [OMG; 2001e]

Metoden `resolve_initial_references` er en metode i ORB-interfacet og har til funktion at returnere referencer til vitale services. Metoden benyttes til bootstrapping, det vil sige til at få fat i de første referencer til CORBA-Objekter. Herefter kan metodekald på disse objekter returnere referencer til yderligere CORBA-Objekter. `resolve_initial_references` tager en streng som parameter. Denne streng angiver hvilket objekt, der ønskes en reference til (typisk en CORBA-Service eller POA'en). Udover "`RootPOA`" er en række andre strenge reserverede i denne sammenhæng. Dette gælder blandt andet "`NameService`", der forespørger en reference til en navneservice. [OMG; 2001d] Initialiserings- og bootstrappingprocessen illustreres på **Figur 2.4**



Figur 2.4: Metodekald illustreres som pile, hvor kaldet angives efterfulgt af et kolon samt en eventuel returværdi. Figuren viser en servant, der initialiserer sig på ORB'en, hvorpå den skaffer en reference til POA'en og aktiverer sig selv på denne (dvs. registrerer sig som tilgængelig for metodekald). Derpå skaffes en reference til en navneservice, hvori den slår en objektreference til endnu et CORBA-Objekt op (vha. metoden `resolve`). Metodekaldet `init` er statisk, hvorimod alle øvrige metodekald foregår på et instantieret objekt. Kaldene af `resolve_initial_references` er en del af bootstrapping-processen, der skaffer de første referencer. Som illustreret med navneservicen kan kald af metoder på disse referencer resultere i, at der returneres yderligere objekt-referencer.

Nu kan man så med rette stille spørgsmålet: Hvorfra kender `resolve_initial_references` referencerne til de grundlæggende services? I tilfældet "`RootPOA`" er der tale om et objekt leveret af den forhåndenværende ORB-implementation, og metoden ved derfor hvilket objekt, den skal returnere en reference til. I tilfældet NameService (og for øvrigt andre services, som f.eks. TransactionService), må ORB'en konfigureres til, at kende en reference til den service der ønskes benyttet. Dette kan gøres på en standardiseret måde, via parameteren `-ORBInitRef`, der gives til ORB'en, når denne initialiseres (parameteren sendes med i `init`-kaldet). Ved hjælp af denne parameter kan en objektreference, der peger på en bestemt service, angives. Objektreferencen angives som en IOR på strengform. Et eksempel kunne være:

```
-ORBInitRef NameService=IOR:<her står en IOR-streng>
```

Imidlertid er IOR-strenge ofte forholdsvist lange og består af hexadecimale cifre, som et menneske vanskeligt kan finde nogen mening med. Derfor er disse strenge ikke velegnede til indtastning via en prompt eller lignende. I stedet kan man naturligvis gemme IOR-strengen i en fil og indlæse den herfra. For at denne metode er nyttig, kræver det dog, enten at servanten distribuerer en fil med dens IOR-streng, hver gang den startes op, og at klienten kan tilgå denne fil (f.eks. via ftp), eller at serveren benytter en persistent IOR, således at denne ikke skal ændres, når servanten genstartes (medmindre servanten flyttes til en anden maskine – i hvilket tilfælde ny adresse-information nødvendigvis må udveksles). Den, i filen gemte IOR-streng, kan benyttes i forbindelse med `-ORBInitRef` og efterfølgende kald af `resolve_initial_references`, men kan også indlæses direkte af klienten og benyttes i et kald af `orb.string_to_object` for at skaffe en objekt-reference.

Da IOR-strenge er vanskelige at håndtere for mennesker, har OMG indført nogle URL-formater i CORBA-standarden, der kan benyttes i forbindelse med IIOP (og altså TCP/IP protokollerne), og som minder om http URL'er, som vi

kender dem. Disse URL'er er lettere for mennesker at udveksle og indtaste og kan benyttes som alternativ til IOR-strenge, hvilket specielt er nyttigt i forbindelse med bootstrapping. En CORBA-ORB, der understøtter disse URL'er (som først blev indført med CORBA/IIOP v. 2.4 [OMG; 2000]), kan benytte en sådan URL i forbindelse med `-ORBInitRef` eller `orb.string_to_object` og derved omsætte den til en objekt-reference.

Et af de ovenfor omtalte URL formater kaldes Corbaloc, og en Corbaloc-URL kan for eksempel se således ud:

```
corbaloc:iiop:1.1@host.com:2060/NameService
```

i ovenstående eksempel kan protokollen og protokol-versionen udelades, dermed får vi URL'en:

```
corbaloc::host.com:2060/NameService
```

2060 angiver på vanlig vis, hvilken port serveren lytter efter Corbaloc-forespørgsler på. Teksten efter "/" identificerer objektet entydigt på serveren. Corbaloc er ikke så generel som IOR, da den kun kan benyttes på Internettet, men til gengæld vil den, for internet-brugere, virke naturlig at benytte og kan dermed være med til at lette bootstrapping, da det bliver lettere at kommunikere objektreferencer mellem mennesker.

2.2 Navneservice

Dette afsnit indeholder to underafsnit. Først en overordnet beskrivelse af navneservicer, der skal læses som en introduktion til begrebet, og hvis formål er at lægge op til næste underafsnit, som er en detaljeret beskrivelse af CORBA's navneservice.

2.2.1 Navneservicer generelt

Formålet med dette afsnit er at beskrive navneservicer udfra et overordnet perspektiv. Der vil, teksten igennem, blive draget paralleller til Internets navneservice – Domain Name Service (DNS). Formål med og ønskede egenskaber ved en navneservice ridses op.

En navneservice har følgende formål:

- at knytte computerspecifikke adresser til menneskeligt forståelige navne.
- at gøre en række ressourcer tilgængelige for dets klienter.
- at gøre klienten uafhængig af ressourcernes lokalitet.

Derudover kan en navneservice tage sig af replikerede servere og belastningsfordeling. [Coulouris; 2001]

Et klassisk eksempel på en navneservice der opfylder disse mål, er den nok bedste kendte og største navneservice: Domain Name Service. Denne navneservice bruges til at lokalisere ressourcer på Internettet udfra et, menneskeligt set, letlæseligt syntaktisk navn kaldet en Uniform Ressource Locator (URL). En bruger kan med en browser søge efter en ressource – som for eksempel en hjemmeside – ud fra en given URL. Denne søgning foregår, ved at browseren kontakter en DNS-server med en forespørgsel om URL'en. DNS-serveren returnerer derefter, om muligt, ressourcens adresse i form af et IP-nummer. IP-nummeret bruges efterfølgende til at lokalisere og udnytte ressourcen – kommunikation mellem DNS og browser er normalt ikke synligt for brugeren.

Brugeren behøver ikke at kende IP-nummeret – altså adressen – på den specifikke computer, som udbyder den ønskede ressource. Brugeren skal kun kende det let forståelige domæne-navn, som ressourcen er tildelt.

Foruden således at gøre en række ressourcer tilgængelige for dets klienter, ud fra et lettere forståeligt navngivningssystem, tilbyder DNS understøttelse af replikering og belastningsfordeling. Eksempelvis ved at belastede hjemmesider deles ud på replikerede servere, således at DNS kan henvise klienter til de mindst belastede af disse.

Dette er også et eksempel på, hvorledes klienten kan udnytte en ressource uden forhåndskendskab til dens lokalisering. Hvad klienten skal kende er en adresse til navneservicen og et navn på ressourcen.

Efter således at have ridset nogle af de servicer op som en navneservice tilbyder, vil de egenskaber, der skal til at implementere en stabil og funktionel navneservice, blive skitseret.

Ønskede egenskaber ved en navneservice:

- Skal kunne skaleres op til et stort antal brugere
- Fejlsikker
    - Fejltolerant overfor lokale fejl og klient fejl.
    - Fejltolerant overfor servernedbrud.

Med hensyn til at skalere til et stort antal brugere, findes der forskellige tiltag, der gør en navneservice i stand til dette. Tre faktorer gør sig gældende:

- Den tid det tager at behandle den enkelte klient.
- Antal klienter per server.
- Den mængde data der skal holdes og behandles.

Tiden det tager at behandle navneservicens klienter kan forbedres ved at benytte en cache til at holde de seneste opslag. Navneservice-serverne kan replikeres og belastningen uddeles – således er der færre klienter per server. Mængden af data hos den enkelte server – og dermed behandlingstiden – kan mindskes ved at data deles mellem flere servere. Dette fungerer på flere måder: En klient kan have adresser til flere servere, til hvilke den sender sin forespørgsel en efter en, indtil den får et svar fra den server, der kan besvare forespørgslen. Eller klientens navneservice-server kan foretage forespørgslen hos de andre servere for den. En anden mulighed er, at serveren sender de forespørgsler, for hvilke den ikke har komplet data, videre til en anden server, som den ved har, og at den nye server ligeledes sender videre, hvis den heller ikke har. Dette kræver at data er repræsenteret på en sådan måde, at den ved hvilken server, den med rette bør sende forespørgslen videre til.

Skal en navneservice være pålidelig, er det vigtigt, at servicen er stabil og til rådighed for dens brugere til ethvert tidspunkt. Serveren, hvorpå servicen kører, skal først og fremmest være fejlsikker, men servicen selv skal også være sikker. En pålidelig navneservice skal være tolerant over for fejlbehæftede klienter og fejlagtige brugerhandlinger. Fejltolerance kan opnås ved at begrænse bruger og klients virkefelt ved at tilbyde et veldefineret interface til servicen – altså en indkapsling af servicen, der ikke tillader fejlagtige handlinger. Derudover skal navneservicen om muligt være uafhængig af dets klienter (tilstandsløs). Hvis en service ikke holder nogen oplysninger om de klienter, der benytter sig af den, betyder et klientsammenbrud ikke, at servicen holder unyttig data.

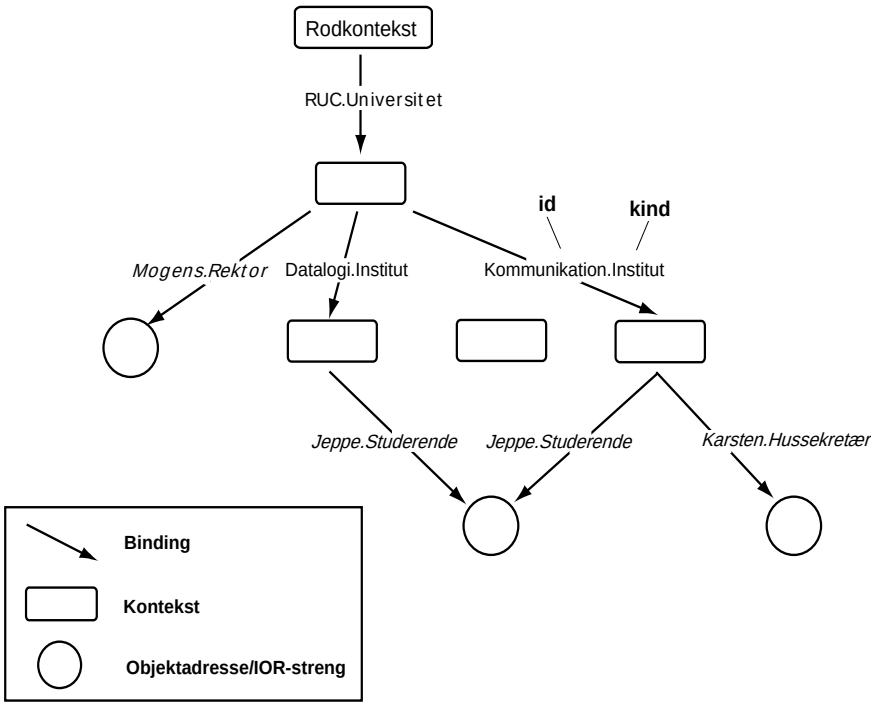
En navneservice kan desuden gøres mere stabil ved replikering, som det ses ved DNS. Sker der et servernedbrud, kan en replikeret server benyttes. Dette kræver, at klienten har adressen på de replikerede servere. [Coulouris; 2001]

2.2.2 CORBA's navneservice

OMG definerer CORBA's navneservice i en specifikation. Denne specifikation beskriver et interface til servicen, og den funktionalitet en implementering af dette skal yde. Dette interface og de funktioner, som det skal implementere, vil blive beskrevet i dette afsnit. Funktionaliteten, interfacet skal yde, er relativt veldefineret, men den implementerende programmør skal stadig tage stilling til en del designvalg, der har betydning for implementationens hastighed og skalerbarhed. Disse designvalg vil endvidere blive pointeret, men ikke diskuteret, i dette afsnit.

Som forklaret i afsnit **2.1.1 Overordnet arkitektur** er det CORBA's opgave at muliggøre kommunikation mellem forskellige programmeringssprog på forskellige maskinarkitekturer gennem metodekald mellem fjernobjekter. Denne kommunikation startes, ved at en objektreference til et fjernobjekt initialiseres af klienten. Måden, hvorpå denne og andre objektreferencer skaffes, kan som sagt være ved, at referencerne, som strenge i filer, transporteres fra server til klient. Denne potentielt meget besværlige proces – afhængig af antallet af benyttede fjernobjekter – kan afhjælpes af CORBA's navneservice.

CORBA's navneservice tilbyder organisering og navngivning af fjernobjektreferencer (IOR). Fjernobjektreferencerne organiseres i en navnegraf, som er opbygget af kontekster. I en kontekst kan fjernobjektreferencer til andre kontekster samt til objekter bindes. Alle bindinger i en kontekst er navngivet med en navnkomponent bestående af to tekststrenge, kaldet `id` og `kind`, hvis indhold er op til navneservicens brugere. **Figur 2.5** illustrerer et eksempel på en navnegraf. Kontekster fungerer som holdere af navne til fjernobjektreferencer og andre kontekster, som igen kan fungere som holdere. [OMG; 2001ns] Der kan drages en pædagogisk parallel til gængse filsystemer; kontekster kan forstås som mapper og objektnavne som filer, med den forskel at forskellige objektnavne kan referere til samme fjernobjekt, og forskellige kontekster kan holde den samme kontekst, og derudover at navnegrafen kan være cyklisk, eksempelvis ved at kontekst 1 henviser til kontekst 2, som igen henviser til kontekst 1. På illustrationen står `id` og `kind` over bindingerne adskilt af punktum.



Figur 2.5: Eksempel på en CORBA-navneservice navnegraf

Foruden således at give brugeren mulighed for at organisere og navngive fjernobjekter i et logisk system, gør CORBA-navneservice klienten uafhængig af fjernobjekternes fysiske lokalitet. Klienten behøver kun at have en reference til navneservicen, og navnene på de fjerneobjekter den skal bruge, for at skaffe IOR-adresser til de ønskede fjernobjekter.

2.2.2.1 OMG's CORBA-navneservice specifikation

Interfacet til CORBA's navneservice er som sagt defineret i IDL (som er beskrevet i afsnit **2.1.2 IDL**). Dette interface og den funktionalitet en implementation af dette kræver, ifølge OMG's specifikation [OMG; 2001ns] vil blive forklaret i dette underafsnit.

En forenklet udgave af navneservice-interfacet er vist **Figur 2.6**. Interfacet er forenklet, idet visse af de undtagelser de enkelte metoder skal kaste ikke er medtaget – hvor de er vigtige, vil de være forklaret i selve teksten. Det komplette interface findes i **10 Appendiks B – Navneservice IDL**.

```
struct NameComponent{ Istring id; Istring kind; };

typedef sequence<NameComponent> Name;

Interface NamingContext{

    void bind(in Name n, in Object obj)

    void rebind(in Name n, in Object obj)

    void bind_context(in Name n, in NamingContext nc)

    void rebind_context(in Name n, in NamingContext nc)

    Object resolve(in Name n)

    void unbind(in Name n)

    NamingContext new_context

    NamingContext bind_new_context(in Name n)

    void destroy
```

```
    list(in unsigned long how_many, out BindingList bl, out
    BindingIterator bi)

};
```

Figur 2.6: IDL-interface til CORBA navneservice.

2.2.2.1.1 NameComponent og Name

Et navne-komponent (`NameComponent`) består som sagt af to tekststrenge – `kind` og `id`. Disse to strenges formål er at give brugeren et katalogiseringsværktøj til sine objekter. Strengens længde, og de karakterer som er tilladt i dem, er implementationsafhængigt. Altså op til den implementerende part. På **Figur 2.5** ses navnekomponenter illustreret som to tekststrenge adskilt af punktum. `id` først og `kind` bagefter som i eksemplet `Kommunikation.Institut`.

`Name` er en sekvens af navne-komponenter, der tilsammen identificerer en objektreference eller en kontekst i en navnegraf. Fra eksemplet **Figur 2.5** er den objektreference, der er bundet til `Karsten.Hussekretær` fra rodkonteksten, identificeret som:

```
Name = {NameComponent(RUC.Universitet), NameComponent(Kommunikation.Institut,
NameComponent(Karsten.Hussekretær)}
```

Et sådant navn, der består af flere navnekomponenter, kaldes et sammensat navn.

2.2.2.1.2 bind og rebind

Første metode i interfacet `void bind(in Name n, in Object obj)` har til formål at binde et objekt til et navn. Dette gøres udfra en given kontekst. Brugeren skal levere en objektreference og et `Name`. Objektreferencen skal være til et CORBA-objekt og `Name` en sekvens af navnekomponenter som beskrevet i ovenstående afsnit.

Det er muligt at binde samme objekt under forskellige navne og i forskellige kontekster, og ligeledes er det muligt at bruge samme navn i forskellige kontekster men ikke samme navn i samme kontekst. Prøver brugeren at binde et allerede bundet navn med `bind`, kastes en undtagelse.

`rebind` har samme funktion som `bind`, med den forskel at samme navn kan bruges i samme kontekst. Gøres dette, overskrives den forrige binding, og den sidst bundne objektreference vil gøre sig gældende ved `resolve` – forklaret senere i afsnittet.

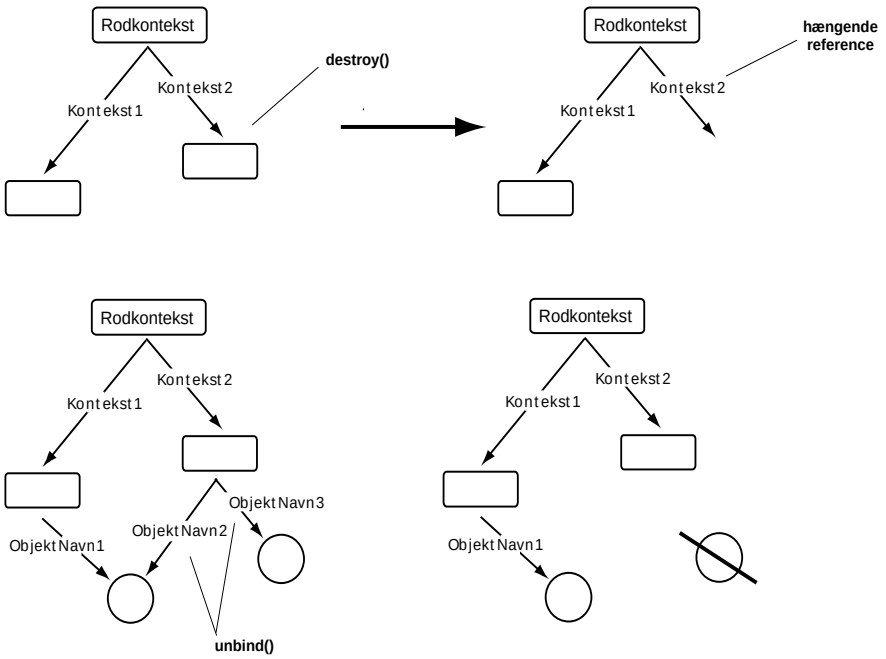
2.2.2.1.3 bind_context og rebind_context

Disse to metoder binder eksisterende kontekster til navne-komponenter. De virker som `bind` og `rebind` bortset fra, at objektreferencerne skal referere til kontekstobjekter, og at kontekster bundet ved disse metoder, bliver nye knuder i navnegrafen. Disse knuders funktionalitet benyttes ved fremtidige opslag i navnegrafen. Som ved `bind` og `rebind` navngives bindingen af en navnekomponent. Det er desuden muligt at binde en kontekst fra en anden navneservice. En navnegraf kan således være repræsenteret i en række navneservicer, og idet en CORBA-objektreference er en generel henvisning, der kan referere til et objekt på en fremmed maskine, kan dele af navnegrafen befinde sig på forskellige maskiner i et distribueret system. Se eventuelt **Figur 2.8**.

`rebind` giver den implementerende part et problem. Når `rebind_context` knytter en ny kontekst til et navn, hvad skal der så gøres med den gamle kontekst? Den kan ikke uden videre fjernes, da navneservicen ikke ved, hvorvidt dens klienter eller andre navneservicer stadig har referencer til konteksten. Det samme problem opstår ved `new_context`.

2.2.2.1.4 destroy og unbind

På **Figur 2.7** kan det ses hvordan de to metoder `destroy` og `unbind` virker.



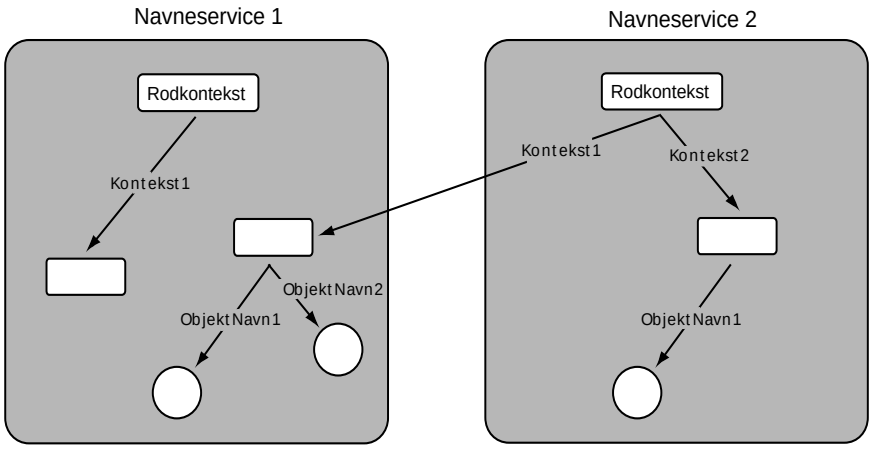
Figur 2.7: Øverst ses hvordan metoden `destroy` benyttes til at nedlægge et kontekst-objekt – bemærk at metoden efterlader en hængende reference, som ikke længere peger på noget. Nederst ses metoden `unbind` i aktion. `Unbind` fjerner en binding af en objektreference eller en kontekst. Bemærk at det er objektreferencen der fjernes og ikke selve objektet.

`destroy` benyttes til at fjerne kontekster. Disse skal være tomme og må altså ikke indeholde nogle bindinger hverken til objekter eller andre kontekster. `destroy` efterlader hængende referencer, hvilket er uundgåeligt af flere grunde; for det første kan navneservicen ikke vide, om andre navneservicer har bindinger til den pågældende kontekst, og for det andet kan andre klienter have referencer til konteksten igen uden dens viden. Disse vil derfor have ”døde” referencer til den nedlagte kontekst.

`unbind` fjerner bindinger og kan derfor medføre forældreløse kontekster, det vil sige kontekster, der ikke er bundet under andre kontekster i samme navneservice. Er en kontekst forældreløs, kan det være fornuftigt at nedlægge den, således at systemressourcerne ikke spildes. Hvorledes dette gøres, og om det overhovedet gøres, er op til den implementerende part.

2.2.2.1.5 new_context og bind_new_context

`new_context` skaber en ny kontekst, men uden at binde den til et navn. Det betyder, at klienten kan have en kontekst, som kun han kan benytte. Klienten kan binde nye kontekster og objekter til konteksten, men hvis ikke klienten binder konteksten til et navn, vil han ikke kunne få adgang til konteksten igen, hvis han mister sin reference til den. Hvad dette også betyder, er at en navneservice kan belastes af forældreløse kontekster, som ingen kan få fat i. For navneservicen kan ikke uden videre rydde op og slette ubundne kontekster, da den ikke kan vide, om klienter eller andre navneservicer har referencer til dem. **Figur 2.8** illustrerer netop denne problemstilling. Navneservice 1 har ikke mulighed for at vide, om den ubundne kontekst er bundet i andre navnekontekster eller er refereret til af en dens klienter.



Figur 2.8: Illustrerer en, i værtsnavneservicen (navneservice 1), ubunden kontekst. Konteksten er skabt med `new_context` i navneservice 1 og bundet til rodkonteksten i navneservice 2.

`bind_new_context` har samme funktion, men binder dog samtidig konteksten til et navn og undgår derved ovenstående problem.

2.2.2.1.6 resolve

Denne metode returnerer CORBA-objektreferencer udfra en given sekvens af navnekomponenter. Typen på objektreferencen er `Object`, så det er nødvendigt for klienten at typekonvertere objektreferencen. `resolve` kan, som de andre metoder, kaldes i en hvilken som helst kontekst.

2.2.2.1.7 list

`list` returnerer en liste over bindinger i den kontekst, på hvilken den blev kaldt. Den returnerede liste indeholder bindingernes navne og deres typer, hvor typen enten er kontekst eller objekt. `how_many` er et heltal, der angiver hvor mange bindinger, der skal returneres i listen. De resterende bindinger returneres via en `BindingIterator`. Denne iterator kan gennemløbes med metoden `next`, der returner næste binding. Derudover har iteratoren metoden `next_n`, der returnerer de næste n (hvor n er et antal) bindinger i en liste.

Der opstår igen et problem ved brugen af iteratorer, hvilket er, at navneservicen ikke kan være sikker på, at brugeren er færdig med at benytte dem, og derfor er der mulighed for overflow, hvis klienten undlader at nedlægge sine iteratorer. Dermed begrænses skalerbarheden. Specifikation tillader derfor en implementationsafhængig nedlæggelse af iteratorer.

2.2.2.1.8 afrunding

Som det fremgår af interfacebeskrivelsen, har designeren af en navneservice, flere valg der skal tages. Der er således et par problemer vedrørende dataspild, der skal tages hånd om. Disse opstår ved `rebind_context`, `new_context` og ved `list`.

`rebind_context` og `new_context` kan, hvis klienten ikke rydder op efter sig, medføre forældreløse kontekster. Navneservicen har dog ikke mulighed for at vide, hvorvidt de refereres til eller ej, da de kan være bundet i andre kontekster eller refereret til af navneservicens klienter.

Specifikationen siger intet om hvordan, endsige om, dette problem skal løses. Det må derfor være op til den implementerende part at dokumentere sine designvalg nøje.

`list`'s iteratorer kan, efter specifikationerne, nedlægges efter forgodtbefindende, og en klient skal derfor være opmærksom på at en iterator til et givent tidspunkt, måske ikke vil være til rådighed. Designet bør igen dokumenteres.

2.2.2.2 OMG's udvidede CORBA-navneservice specifikation

Der eksisterer et udvidet interface til CORBA-navneservicen, som nedarver fra `NamingContext`. Dette IDL-interface indeholder hjælpemetoder, der kan bruges til konvertering mellem strenge og sekvenser af navnekomponenter. Disse er `to_string` og `to_name`. Specifikationen angiver en fast syntaks, for hvorledes et navn repræsenteres på strengform. Derudover har interfacet en `resolve_str` metode, der tager et navn som streng i stedet for som en sekvens af

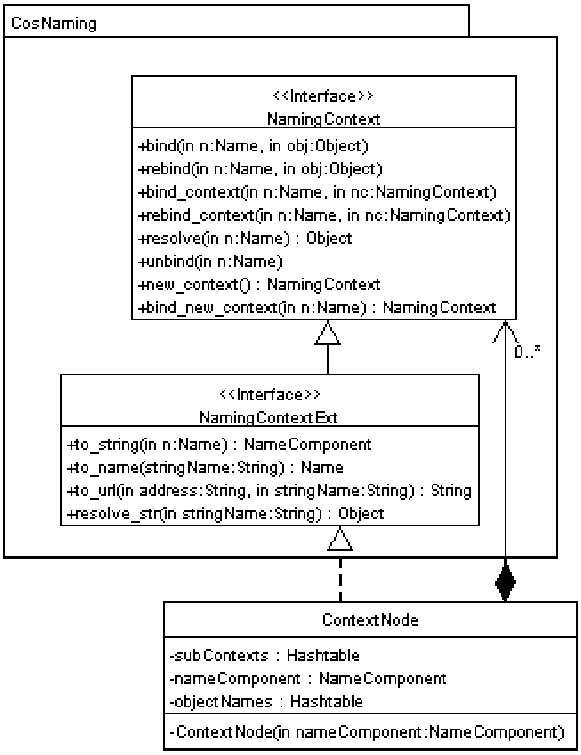
navnekomponenter og slutteligt en metode `to_url`, hvis funktion er at sammensætte en Corbaloc-adresse og et navn på strengform til en gyldig URL og samtidig undersøge dem for syntaksfejl. Disse metoder skal implementeres, for at navneservicen opfylder CORBA-navneservice 1.1 specifikationen [OMG; 2001ns] .

3 Implementation af en CORBA-Navneservice

Vi har, i dette projekt, valgt at implementere en CORBA-navneservice, der skal leve op til CORBA-navneservicespecifikationen version 1.1 [OMG; 2001ns] . Ydermere har vi valgt at gøre servicen persistent – det vil sige gemme navnegrafen i permanent lager. I de følgende afsnit vil dele af denne implementation blive diskuteret og beskrevet. Koden til implementationen findes i **11 Appendiks C – Navneservice kildekode**.

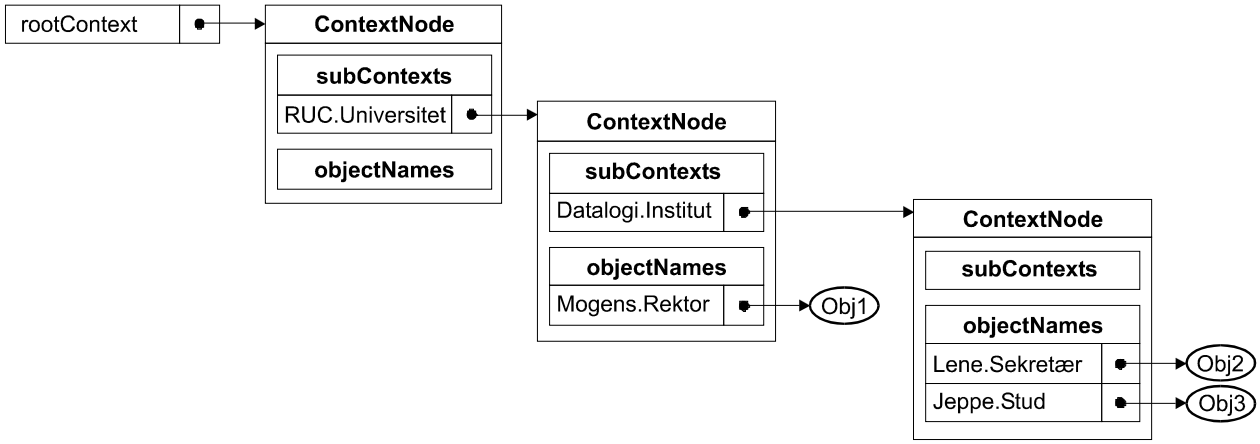
3.1 Navnegrafen

Hovedklassen i den implementerede navneservice er klassen `ContextNode`, der repræsenterer en navnekontekst med de i standarden specificerede metoder. En `ContextNode` indeholder en tabel over de objekter, der er bundet i den kontekst, den repræsenterer, samt en tabel over de sub-kontekster, der er bundet i den kontekst, den repræsenterer. Klassediagrammet på **Figur 3.1** illustrerer den grundlæggende struktur omkring `ContextNode`.



Figur 3.1: Den grundlæggende struktur omkring `ContextNode`.

Som det fremgår af klassediagrammet, så har en `ContextNode` referencer til en række objekter, der implementerer interfacet `NamingContext` (nemlig dens sub-kontekster). Det eneste objekt, i vores design, der implementerer dette interface er `ContextNode`, og der er derfor tale om en rekursiv datastruktur. Denne datastruktur repræsenterer navnegrafen. **Figur 3.2** viser et eksempel på en navnegraf, opbygget af `ContextNodes` og bundne objekter.



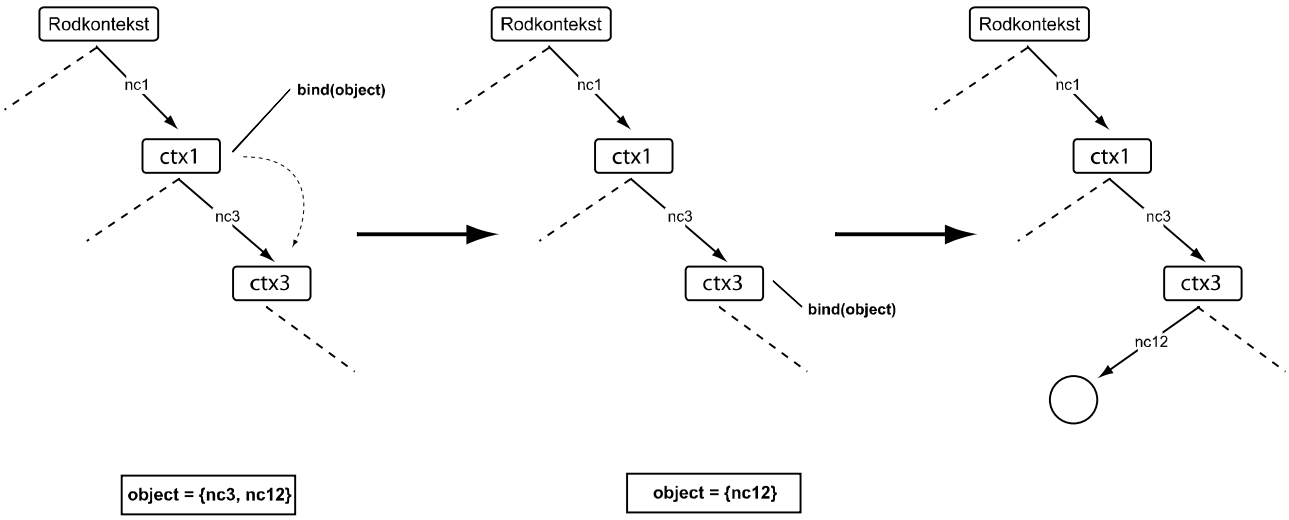
Figur 3.2: Eksempel på en navnegraf der illustrerer, hvorledes grafen er opbygget af `ContextNode`-objekter. "rootContext" er en reference, der angiver rodkonteksten. Hvorledes denne registreres, så klienter kan få fat i den, beskrives i afsnit 3.1.2 Registrering af rodkonteksten. Hver `ContextNode` indeholder to tabeller. Én over de kontekster og én over de objekter der er bundet til den.

Tabellerne over bindinger i den enkelte kontekst er implementeret som Java `HashTable`, da disse er hurtige at slå op i. Som nøgle benyttes navnekomponenten, der navngiver bindingen, og som værdi benyttes en CORBA-reference til en `NamingContext`. Imidlertid er der to krav til en klasse, der skal benyttes som nøgle i en `HashTable`: Den skal implementere `equals`- og `hashCode`-metoden. Dette kan `NameComponent` klassen ikke leve op til, og da denne genereres af IDL-kompileren, ønsker vi ikke manuelt at ændre i denne. I stedet implementeres en "wrapper" klasse med navnet `NCWrapper`. Denne klasse har et felt, der indeholder en `NameComponent`. Feltet sættes i konstruktoren og kan læses ved hjælp af en get-metode. `NCWrapper`-klassen implementerer `equals` og `hashCode` og kan dermed benyttes som nøgle i hash-tabellen.

3.1.1 Implementation af bindingsmetoderne

Implementationen af metoderne `bind`, `rebind`, `bind_context` og `rebind_context` minder meget om hinanden og er derfor bygget ind i en enkelt hjælpemetode kaldet `bindImpl`. Denne metode binder en CORBA-reference, der enten refererer til en kontekst eller til et CORBA-objekt. Via to boolske parametre defineres, om metoden skal binde eller genbinde, altså om den skal overskrive eksisterende referencer, og om det er en kontekst eller et objekt der bindes.

Metoden er implementeret rekursivt, hvilket betyder, at selve bindingen ikke behøver ske i den kontekst hvorpå metoden kaldes, men eventuelt videregives til et metodekald på en sub-kontekst via et nyt (rekursivt) kald af den relevante bindingsmetode. En illustration af dette princip, ses på **Figur 3.3**, hvor kaldet foretages på konteksten `ctx1`, men hvor objektet først bindes under subkonteksten `ctx3`.



Figur 3.3: Binding af objekt. Metoden `bind`, med navnet `{nc3, nc12}` som parameter, kaldes på `ctx1`, der rekursivt kalder `bind`, denne gang med navnet `{nc12}`, på subkonteksten `ctx3`, hvorunder objektet bindes.

3.1.2 Registrering af rodkonteksten

Under serveropstart bliver der instantieret et `ContextNode` objekt, der benyttes som rodkontekst. Dette objekt skal registreres på POA'en, og dette gøres via den private hjælpe-metode `activateNamingContextWithID`, som tager en `ContextNode` og et objektID og returnerer et objekt af typen `NamingContextExt`. Metoden aktiverer `ContextNode` objektet på POA'en og returnerer en CORBA-reference til det derved dannede CORBA-objekt. Metoden kaldes altid med det samme objektID for rodkonteksten, hvilket implicerer, at det altid er det samme CORBA-objekt, der er rodkontekst (på trods af at der instantieres en ny servant ved servergenstart).

For at gøre det muligt at nå rodkonteksten via en Corbaloc URL, registreres objektet under navnet "NamingService", ved hjælp af den ORB-specifikke metode `register_initial_references`. Derpå er det muligt at kontakte navneservicen via Corbaloc og dermed skaffe en reference til rodkonteksten. Ved opstart udskriver serveren også en IOR-streng, der repræsenterer en reference til rodkonteksten, til standard output. Denne IOR-streng kan alternativt benyttes til at kontakte navneservicen.

3.2 Persistens

Da vi ønsker at gøre vores navneservice persistent, må vi gemme navnegrafen på harddisken, således at den kan gendannes efter et server-nedbrud.

For at sikre at den opdaterede navnegraf altid er gemt på disk, således at der ved et server-nedbrud ikke mistes data, som en klient tror er gemt, ønsker vi at skrive en ændring i navnegrafen til disk, før vi returnerer til klienten. Derfor vil vi normalt kun ønske at skrive en enkelt ændring (f.eks. oprettelse eller nedlægning af en enkelt binding) ad gangen til disken.

Der er umiddelbart mange måder at repræsentere navnegrafen på disken på. Nogle muligheder, vi har diskuteret i forbindelse med dette projekt, er:

- Serialiserede objekter – I Java kan objekter serialiseres og skrives til disk for senere at gendannes. Imidlertid vil serialisering af et objekt føre til, at de objekter der refereres til også serialiseres. Dette ville betyde, at en hel node i navnegrafen (inkl. alle bindinger – men dog ikke under-noderne, da disse ikke er refereret ved hjælp af Java-referencer, men ved hjælp af CORBA-referencer) skrives hver gang. Dette forekommer uhensigtsmæssigt, da vi ved, det kun er en enkelt binding, der er ændret.
- Logfil – Vi kunne skrive en logfil, der indeholder alle gennemførte operationer. Derved kan operationerne "genspilles" ved server-genstart. Her kan en enkelt ændring skrives ad gangen. En ulempe her er, at logfilen vil vokse i det uendelige og efterhånden indeholde en masse operationer, hvis effekt overskrives af senere operationer. For at undgå dette, kan man indføre en oprydningsmekanisme, der fjerner unødvendige operationer fra loggen, men det koster ressourcer, samt naturligvis udvikling af en passende "oprydnings-logik".
- Træstruktur – Navnegrafen kan repræsenteres som en filstruktur på harddisken, hvor mapper repræsenterer kontekster, og hvor der i hver mappe ligger en række filer, der repræsenterer de enkelte bindinger i konteksten. Afhængig af det nøjagtige valg af repræsentationsform, kan dette give nogle begrænsninger på navnelængde m.m., afhængig af det underliggende filsystem.
- Relationel database - Navnegrafen kunne gemmes i en passende designet relationel database. Dette ville give et ekstra softwarelag. Hvorvidt hastigheden bliver større eller lavere afhænger af implementationen af databasen og skal ikke afgøres i dette projekt. En database ville give mulighed for at bygge på allerede implementerede løsninger af samtidighedsproblemer m.m.

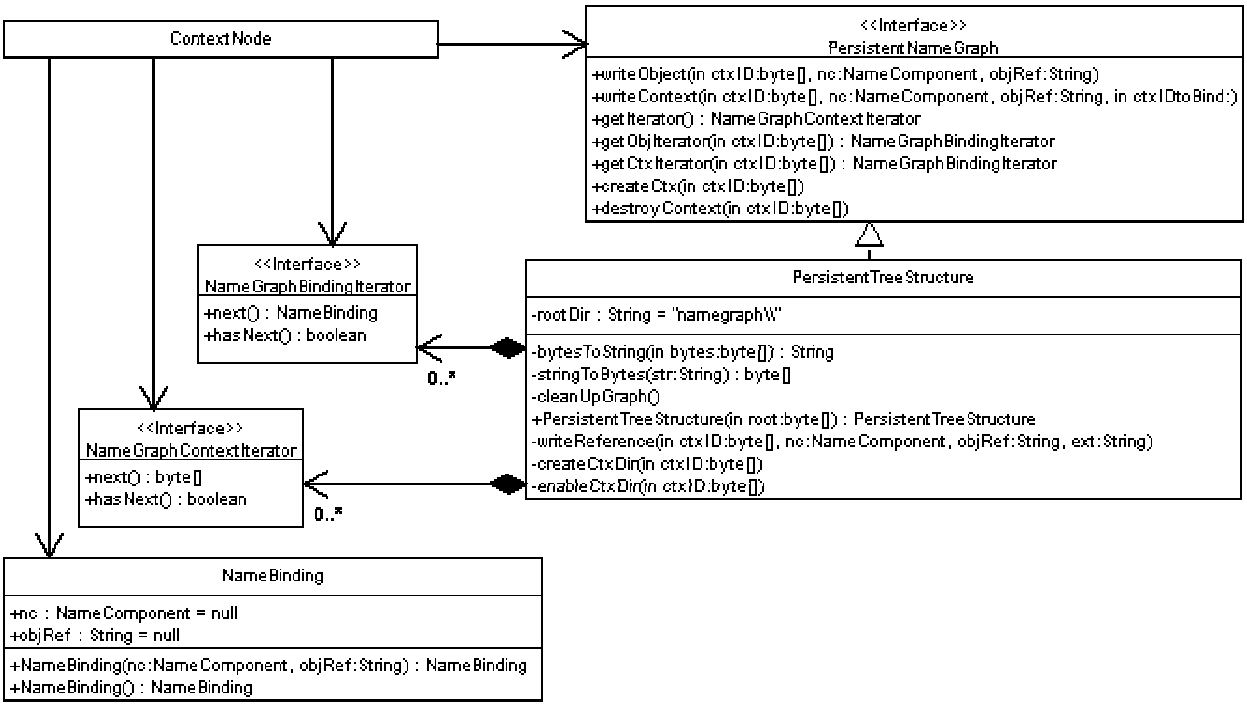
Da der således er mange muligheder for at repræsentere navnegrafen, har vi valgt at uddelegere skrivningen af navnegrafen til et interface, således at der kan implementeres og benyttes forskellige repræsentationer (designmønstret template). Dette interface kaldes "`PersistentNameGraph`". Vi vælger at holde fast i, at der skal kunne skrives én kontekst- eller objekt-binding ad gangen til disken (og har dermed udelukket muligheden for at bruge serialiserede objekter).

Det skal naturligvis være muligt at gendanne navnegrafen fra den skrevne repræsentation. Dette opnår vi, ved at lade `PersistentNameGraph` indeholde tre metoder, der returnerer iteratorer over henholdsvis navnene på alle skrevne kontekster, alle objektbindinger i en given kontekst samt alle kontekstbindinger i en given kontekst. Derved er det muligt at gennemløbe og oprette alle elementer i navnegrafen.

Når en binding, hvad enten det er af en kontekst eller et objekt, skrives til disken, skrives en IOR-streng som reference til det bundne objekt. Hvis objektet derefter skifter objektID på POA'en, vil denne reference ikke længere være gyldig. Dette er normalt ikke navneservicens problem, men i forbindelse med gendannelse af navnegrafen, vil det betyde, at alle kontekst-bindinger i den genstartede server-proces vil være ugyldige, medmindre det sikres, at en given kontekst har det samme objektID før og efter en genstart. Standardindstillingen for rod POA'en er, at den auto-genererer

objektID'et, men det er muligt at oprette en ny POA, og indstille denne til at benytte bruger-specificerede objektID'er. ObjektID'et for et kontekst-objekt kan da fungere som entydig identifikation af en kontekst, og dette benyttes af vores interface, hvor flere af metoderne tager et `ctxID` som identifikation af den kontekst, der skal arbejdes på. Et `ctxID` er af typen `byte[]` (byte-array), da det er denne type, CORBA-standarden benytter til objektID'er. Et `ctxID` skal altså identificere en kontekst entydigt, og vi benytter POA'ens objektID til dette formål. I visse situationer (i vores implementation ved oprydning af forældreløse kontekster) kan det være nyttigt ikke blot at have den refererede konteksts reference men også dens ctxID. Derfor sendes objektID'et, på den kontekst der bindes, med ved skrivning af en kontekst-binding.

Ovenstående overvejelser vedrørende persistens kommer til udtryk i klassediagrammet på **Figur 3.4**.



Figur 3.4: Interfaces og klasser der benyttes til at gøre navnegrafen persistent.

Metoderne i interfacet PersistentNameGraph skal implementeres med følgende funktionalitet:

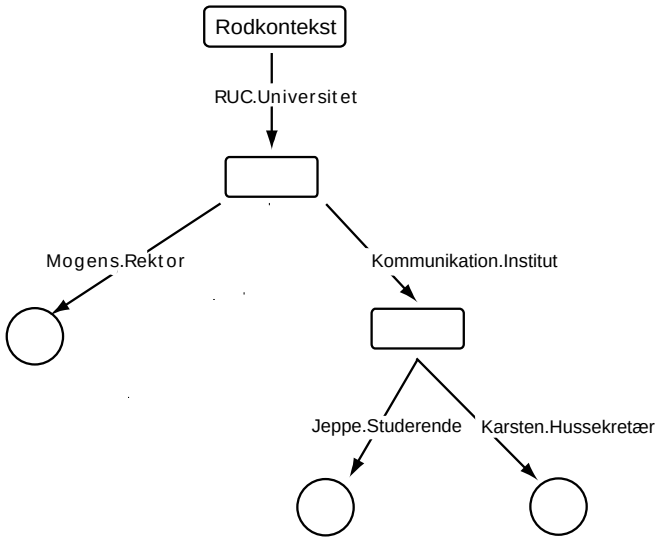
- `createCtx` – Opret en ny kontekst, identificeret ved det angivne `ctxID`, i det persistente lager.
- `destroyContext` – Fjern konteksten, identificeret ved det angivne `ctxID`, fra det persistente lager.
- `writeObject` – Skriv en objektbinding til det persistente lager. Objektet skal bindes i konteksten identificeret ved `ctxID` og er i denne kontekst navngivet ved navnekomponenten `nc`. Strengen `objRef` er en IOR-streng, der angiver en reference til objektet. Det kan forudsættes, at konteksten allerede er oprettet vha. `createCtx`
- `writeContext` - skriv en kontekstbinding til det persistente lager. Konteksten skal bindes i den kontekst, der er identificeret ved `ctxID` og er i denne kontekst navngivet ved navnekomponenten `nc`. Strengen `objRef` er en IOR-streng, der angiver en reference til den kontekst, der skal bindes. Hvis denne konteksts `ctxID` kendes, angives det i `ctxIDtoBind`. I modsat fald er `ctxIDtoBind` sat til `null`.
- `getIterator` – returnerer en `NameGraphContextIterator` over alle kontekster skrevet til persistent lager. Iteratoren returnerer `ctxID`'er for konteksterne ved gentagne kald af dens `next`-metode.
- `getObjIterator` – returnerer en `NameGraphBindingIterator` over alle objekt-bindinger i konteksten identificeret ved `ctxID`. Iteratoren returnerer objektbindingerne en ad gangen, via klassen `NameBinding`, ved gentagne kald af dens `next`-metode.
- `getCtxIterator` – returnerer en `NameGraphBindingIterator` over alle kontekstbindinger i konteksten identificeret ved `ctxID`. Iteratoren returnerer kontekstbindingerne en ad gangen, via klassen `NameBinding`, ved gentagne kald af dens `next`-metode.

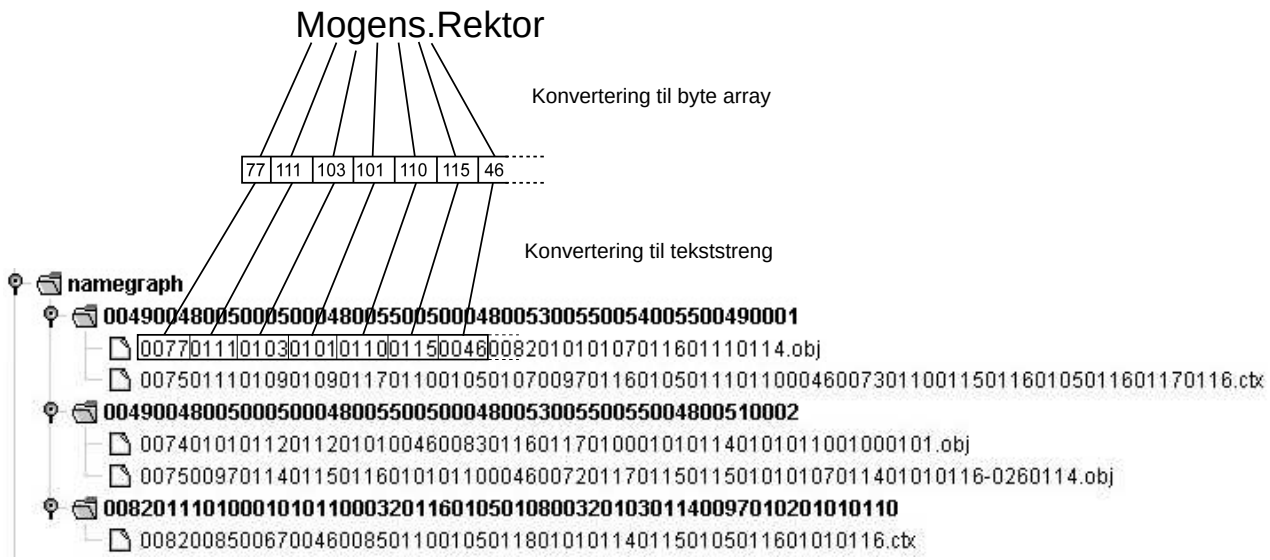
Klassen "`PersistentTreeStructure`" er vores implementation af `PersistentNameGraph`-interfacet. I denne klasse har vi valgt at repræsentere navnegrafen som en mappestruktur, da denne løsning forekom simplere end logfil eller relationel database. Yderligere argumentation for valget er, at en logfil vil medføre serialisering af opdateringer af

navnegrafen, da der kun kan være én bruger ad gangen, der skriver i filen. Med hensyn til at benytte en relationel database giver det et ekstra softwarelag og indfører en yderligere teknologi, som ikke direkte vedrører emnet for dette projekt, og som vi derfor ikke ønsker at ofre tid på.

Mappestrukturen opbygges med udgangspunkt i en ”rodmappe” (i vores konkrete testkørsler og eksempler kaldes den ”`namegraph`”), hvorunder alle data gemmes. I denne rod lægges en række mapper, hvis navne er `ctxID`’er for konteksterne i navnegrafen. I hver af disse mapper ligger filer, der repræsenterer bindinger i konteksten. Disse filers navne er den navnekomponent, som angiver bindingens navn, og indholdet af filen er en IOR-streng, der repræsenterer den reference, som navnet er bundet til. Filer med endelsen ”.obj” repræsenterer objekt-bindinger, og filer med endelsen ”.ctx” repræsenterer kontekst-bindinger. I ctx-filerne vælger vi desuden at skrive `ctxID`’et for den bundne fil, hvis dette gives med i metodekaldet (dvs. hvis parameteren `ctxIdToBind` ikke er `null`).

Da et `ctxID` er et `byte[]`, er det en forenkling at påstå, at mapperne har disse som navne. I stedet vælger vi at lave et `byte[]` til en streng, ved at lade hver byte repræsenteres ved fire karakterer, hvor første karakter er ”0” for positive tal og ”-” for negative tal, de tre næste karakterer er cifre, der angiver bytens værdi (se **Figur 3.5**). Ligesom vi ikke direkte kan bruge `ctxID`’er som mappenavne, kan vi heller ikke direkte bruge navnekomponenter (der jo repræsenteres som objekter) som filnavne, og da der er en række regler for, hvilke navne der er tilladte for filer, og da disse regler ikke tillader for eksempel ”\”, er det heller ikke muligt blot at konvertere navnekomponenterne til strenge via `NameContextExt.to_string` – metoden, da denne bruger ”\” som escape-tegn. I stedet har vi valgt at konvertere navnekomponenten til en streng via ”`to_string`”, hvorpå vi konverterer denne streng til et byte-array via Java-metoden `String.getBytes`. Dette byte-array kan da konverteres til en streng, på samme måde som netop beskrevet for `ctxID`’er. På **Figur 3.5** ses en illustration af en navnegraf samt dennes repræsentation på disken.





Figur 3.5: Illustration af en navnegraf samt navnegrafens repræsentation på disken i en mappestruktur. Mapperne er de tre kontekster (roden + to nye). ctx-filerne er kontekstbindinger (en i roden og en i den ene kontekst). obj-filerne er objektbindinger (en i den ene kontekst og to i den anden). Som eksempel er det vist, hvordan objektbindingen `Mogens.Rektor` er repræsenteret som en fil, hvis navn er dannet ved konvertering af navnekomponenten til en streng bestående af tal.

3.2.1 Regenerering af navnegrafen

Ved servergenstart skal navnegrafen kunne regenereres. Dette er, som nævnt, muligt via de to iterator-interfaces `NameGraphBindingIterator` og `NameGraphContextIterator`, der er illustreret på **Figur 3.4**. Referencer til objekter der implementerer disse interfaces (konstrueret som indre klasser), fås via metoderne `PersistentNameGraph.getIterator` `PersistentNameGraph.getCtxIterator(ctxID: byte[])` og `PersistentNameGraph.getObjIterator(ctxID: byte[])`. `getIterator`-metoden returnerer en `NameGraphContextIterator`, der benyttes til at gennemløbe alle kontekster – hvilket i praksis vil sige alle mapper under vores "rodmappe". Iteratorens `next`-metode returnerer `ctxID`'er for konteksterne en ad gangen, og derved bliver det muligt at instantiere `ContextNode`-objekter, og registrere disse på POA'en med de samme objektID'er som før genstarten. De to andre get-metoder returnerer referencer, til en implementation af `NameGraphBindingIterator` for en given kontekst, og muliggør dermed, at alle bindinger i hver kontekst genoprettes. `NameGraphBindingIterator`-interfacets `next`-metode returnerer en `NameBinding`, der indeholder en navnekomponent (der navngiver bindingen i den givne kontekst) og en IOR-streng – der kan omsættes til en CORBA-reference. Dermed kan `bind` og `bind_context` metoderne kaldes, for at genoprette bindingerne. I pseudokode ser det hele således ud:

```
Opret rodkontekst rootCtx //da denne altid skal oprettes – uanset om der er skrevet noget til disk
Registrer rodkonteksten på POA'en
Opret iterator ctxi over alle kontekster
Så længe der er flere kontekster i ctxi
      Hent kontekstID fra ctxi.next
      Hvis kontekstID er rodkontekstens ID
            Sæt ctxn til at pege på rodkonteksten
      Ellers
            Opret ny ContextNode ctxn
            Registrer ctxn på POA'en med objektID kontekstID
      Opret objekt-iterator for kontekstID
      Så længe der er flere objekt-bindinger
            Kald ctxn.bind med parametre fra objekt-bindingen
            Hent næste objekt-binding
      Gentag
      Opret context-iterator for kontekstID
      Så længe der er flere context-bindinger
            Kald ctxn.bind_context med parametre fra context-bindingen
            Hent næste kontekst-binding
      Gentag
Gentag
```

3.2.2 Binding af kontekster fra CORBA-navneservicer i andre processer

I forbindelse med indførslen af persistens kodes `ContextNode`-klassen, til at benytte `PersistentNameGraph`-interfacet til at skrive en kontekst-binding til disk, ved hjælp af metoden `writeContext`, når bindingen oprettes. Imidlertid tager `writeContext` objektID'et, for den kontekst der bindes, som sidste parameter. Formålet med denne parameter er at

kunne gemme ID'et på disk, da dette er nyttigt, i forbindelse med at afgøre hvilke kontekster der er bundne, og hvilke der er forældreløse (da konteksterne på disk, jo netop identificeres ved deres objektID frem for ved referencen). objektID'et for en kontekst findes med en metode på POA'en: `poa.reference_to_id` anvendt på referencen til den kontekst der skal bindes. Imidlertid fungerer denne metode ikke, hvis referencen peger på et objekt, der ikke findes på den POA, metoden anvendes på. Det betyder, at det ikke umiddelbart er muligt at binde en kontekst fra en anden navneservice i vores navneservice. Dette er naturligvis uacceptabelt – da vi ønsker at leve op til CORBA-standarden. Vi har altså behov for at afgøre, hvorvidt et givent `NameContext` CORBA-Objekt findes på "vores" POA. Der er ikke, jævnfør CORBA-specifiaktionen, en metode til dette på POA'en, men på Sun's ORB (som vi har implementeret vores navneservice på), resulterer et kald af `reference_to_id` med en objektreference til et objekt fra en anden POA som parameter, i en `ClassCastException`. Vi har derfor valgt at fange `ClassCastException`, og hvis denne kastes, konkluderer vi, at der er tale om en kontekst i en anden navneservice – hvor vi ikke har ansvaret for at holde den persistent. Derfor skriver vi konteksten til disk, med den sidste parameter sat til null – hvorved implementationen af `writeContext` undlader at skrive et objektID i filen. Dermed opnår vi, at en kontekst fra en anden navneservice (og dermed på en anden POA) kan refereres fra vores navneservice uden at give problemer ved skrivning til disk.

Ovenstående fremgangsmåde, hvor vi fanger en `ClassCastException`, kan ikke garanteres at virke på alle Java-ORB'er, da det ikke er standardiseret i CORBA-specifikationen, at netop denne undtagelse skal kastes i den givne situation. Derfor skal denne del af koden testes og eventuelt ændres, hvis koden flyttes til en anden ORB.

3.3 Hukommelsesstyring

I forbindelse med implementationen af navneservicen, er det nødvendigt at tage stilling til nogle problemstillinger vedrørende hukommelsesstyring. Det handler dels om navnekontekstens og `BindingIterator`'s `destroy`-metoder, hvis implementering ikke forekommer helt ligefrem i et Java-miljø med garbage-collection og ingen mulighed for eksplicit nedlæggelse af objekter, dels om nedlæggelse af `BindingIterator`-objekter i tilfælde af fejlbehæftede klienter og dels om håndtering af forældreløse kontekster.

3.3.1 Destroy-metoderne

I navneservicespecifikationen [OMG; 2001ns] angives at navnekonteksten skal implementere metoden `destroy`, og at `BindingIterator` ligeledes skal implementere en `destroy`-metode. I Java er det imidlertid ikke muligt direkte at nedlægge et objekt og fjerne det fra hukommelsen. Derimod fjerner Javas garbage-collector automatisk objektet fra hukommelsen, når der ikke længere er nogen, der refererer til objektet. Nu er spørgsmålet så, hvorledes det i Java er muligt at implementere `destroy`-metoden, så den opfører sig som angivet i specifikationen, og så hukommelsen frigives.

Princippet i løsningen på dette problem er ikke videre kompliceret: Sørg for at der kun findes én Java-reference til kontekstobjekterne og lad alle øvrige referencer være CORBA-referencer. Da CORBA-referencer ikke påvirker garbage-collectoren, da de ikke er Java-referencer, men blot objekter der repræsenterer en "adresse" på et CORBA-objekt, kan objekter, der kun refereres til via CORBA-referencer, garbage-collectes. Når så objektet ønskes nedlagt, skal den ene Java-reference til objektet blot fjernes.

Ved registrering på POA'en får denne en Java-reference til et `ContextNode-` eller `BindingIteratorImpl`-objekt, der fungerer som servant for CORBA-objektet. Ved afregistrering fra POA'en fjernes denne Java-reference, og hvis alle øvrige referencer er CORBA-referencer, kan objektet derefter garbage-collectes. `destroy`-metoden implementeres altså som en simpel afregistrering af objektet fra POA'en. Dette bevirker, at alle forsøg på at nå objektet via CORBA-referencer, efter konteksten er nedlagt, vil resultere i undtagelsen: `OBJECT_NOT_EXISTS` (da CORBA-objektet jo ikke længere eksisterer, når det ikke findes på POA'en, og da POA'en ikke er sat til selv at aktivere objekter). Dette stemmer overens med specifikationen og betyder, at vi kan have hængende CORBA-referencer til nedlagte navnekontekster eller iteratorer. Servant-objektet vil blive garbage-collectet når Javas garbage-collector kører (f.eks. ved mangel på systemressourcer).

3.3.2 Nedlæggelse af BindingIterator

`BindingIterator`-objekter oprettes af klienter, der kan benytte dem til et enkelt gennemløb af en konteksts bindinger, hvorpå klienten forventes at nedlægge iteratoren igen. Imidlertid kan det let tænkes, at en klient glemmer at nedlægge sin iterator, eller måske går klienten ned, inden den når det. Dette betyder, at der kommer til at ligge en masse iteratorer, som ingen kan få fat i referencer til, og som først forsvinder ved servergenstart. Disse "døde" iteratorer adskiller sig fra kontekster, som en fejlbehæftet klient mister sin reference til – idet kontekster oftest vil være bundet i navnegrafen, og derfor kan slås op igen (dette er dog ikke altid tilfældet, og undtagelsen – forældreløse kontekster – diskuteres i næste afsnit). Det er naturligvis spild af ressourcer, at have ubrugelige iteratorer liggende, og CORBA-standarden specificerer

derfor, at det er tilladt for navneservice-implementationen at indføre en implementationsspecifik strategi for, hvornår iteratorer nedlægges. Klienten må så tage højde for, at en iterator kan blive nedlagt, selvom den har en reference til den. [OMG; 2001ns]

Vi har valgt at indføre en tidsgrænse for iteratorers levetid, således at en iterator, der ikke har været brugt i et fastsat tidsrum, bliver nedlagt. Dette er implementeret ved, at iteratoren har tilknyttet en timer (der kører på en separat tråd), der nulstilles hver gang, der kaldes en metode på iteratoren. Derved vil iteratoren vedblive med at eksistere, så længe klienten benytter den regelmæssigt. Når klienten ikke benytter iteratoren i det fastsatte tidsrum vil timeren løbe ud og iteratoren blive fjernet. Derfor bør en klient naturligvis lave sit gennemløb, straks iteratoren er oprettet. Det er stadig at foretrække, at klienten selv husker at nedlægge iteratoren – da hukommelsen så kan blive frigivet tidligere, end hvis timeren først skal løbe ud. Hvor langt et tidsrum, en iterator skal have at leve i, er en afvejning mellem de tilgængelige systemressourcer og risikoen for at genere klienter, ved at nedlægge iteratorer, der stadig bruges. I denne forbindelse må man også huske, at klienternes kald af metoder på iteratoren kan blive forsinket af netværket. Derfor skal iteratorens levetid være større, end den tid det tager en besked at komme fra klient til server via netværket – da iteratoren ellers altid vil blive nedlagt, før klienten kan komme til at benytte den. Desuden skal der være en rimelig tid til, at klienten kan nå at foretage sig noget med en returværdi, før den kalder iteratoren igen.

Vi kunne også have valgt andre kriterier for nedlæggelse af iteratorer. For eksempel kunne vi vælge et kriterium baseret på et maksimalt antal samtidige iteratorer. Dette vil imidlertid betyde, at så længe der ikke er oprettet flere iteratorer end det fastsatte maksimum, vil eventuelle ”glemte iteratorer” ikke blive nedlagt og vil derfor spilde systemressourcer. En tredje strategi kunne være at bygge nedlæggelsen af iteratorer på behovet for systemressourcer. For eksempel kunne det måske være hensigtsmæssigt, at nedlægge iteratorer når garbage-collectoren starter (da dennes start kan skyldes, at der er mangel på ressourcer), men dette ville dog stadig kræve en regel for hvilke iteratorer, der kan nedlægges når behovet melder sig. Fordelen ved at basere nedlæggelse på mangel på systemressourcer er, at det giver mulighed for ikke at genere klienter ved at nedlægge deres iteratorer, medmindre der er behov for det, og samtidig bliver der ikke spildt ressourcer, når systemet har brug for dem til noget andet. Der er imidlertid implementationsmæssige problemer i at opstille relevante system-ressource kriterier, samt vedligeholde en passende regel for, hvilke iteratorer der må nedlægges, da det naturligvis ikke er rimeligt at nedlægge alle iteratorerer på grund af ressourcemangel, men i stedet må der føres en form for regnskab over, hvilke iteratorer der sandsynligvis ikke er i brug.

En kombineret løsning ville eventuelt være mulig. Man kunne for eksempel lade iteratorerne have en timer, men undlade at nedlægge dem selvom timeren løber ud. Hvis der så senere bliver kaldt en metode på iteratoren, kunne timeren startes igen, men hvis et givent kriterium (f.eks. systemressourcer eller antal oprettede iteratorer) bliver nået, nedlægges iteratorer, hvis timer er løbet ud.

Det mest hensigtsmæssige kriterium for hvornår iteratorer nedlægges vil nok være et kriterium baseret på systemressourcer – da formålet med automatisk nedlæggelse af iteratorer netop er at undgå at spilde ressourcer. Vi har imidlertid valgt ikke at tage implementeringsspørgsmålet: ”Hvorledes afgør vi mangel på systemressourcer” op i dette projekt, og holder os derfor til den tidsstyrede nedlægning.

3.3.3 Forældreløse kontekster

Som beskrevet i afsnit **2.2.2.1 OMG’s CORBA-navneservice specifikation** kan der opstå forældreløse kontekster ved brug af metoderne `bind_context`, `unbind` (med navnet på en kontekst som parameter) og `new_context`. Disse kontekster kan enten være refereret fra et andet sted (en anden navneservice, eller en navneservice-klient), men kan også være resultatet af uhensigtsmæssig brug af de nævnte metoder, eller måske af at en klient er gået ned, inden den har fået ryddet op efter sig (f.eks. ved at kalde destroy på kontekster den har oprettet til midlertidig brug, og som ikke længere er bundne). Derfor kan navneservicen ikke vide, om en forældreløs kontekst er en rest fra en fejlbehæftet klient, eller om den er refereret og bruges fra en anden CORBA-applikation. Forældreløse kontekster, der stammer fra fejlbehæftede klienter, kan aldrig ryddes op af navneservice-klienter, da disse ikke kan skaffe en ny reference til konteksten, da den jo ikke kan slås op via resolve eller findes via en iterator, når den er forældreløs. Derfor risikerer en navneservice at blive fyldt op af ubrugelige kontekster. Da navneservicen ikke kan afgøre, om en forældreløs kontekst faktisk er ubrugelig, eller om den refereres fra et andet sted, er det ikke trivielt at tage stilling til, hvad der skal gøres ved disse kontekster.

Den simple løsning er naturligvis at vedtage, at klienter skal opføre sig ”pænt”, og at navneservicen derfor går ud fra, at alle forældreløse kontekster er OK. Dette betyder imidlertid at servicen er sårbar overfor klienter, der går ned på uheldige tidspunkter, og navneservicen vil derfor, over tid, indeholde flere og flere ubrugelige kontekster, der aldrig nedlægges. Dette finder vi ikke acceptabelt, og vi vil derfor diskutere forskellige måder at rydde op på.

En løsning kunne være at udvikle et administrator-værktøj til navneservicen, der, for eksempel via den persistente repræsentation på harddisken (hvori de forældreløse kontekster vil kunne findes), kan vise en komplet navnegraf som

den ser ud i servicen, og så lade en menneskelig administrator slette unødige kontekster. Dette kræver naturligvis at administratoren har tilstrækkelig viden, til at træffe fornuftige beslutninger. En sådan løsning vil have den fordel, at menneskelig viden om navnegrafens semantik kan være med til at forme korrekte beslutninger, om hvad der skal slettes, og at værktøjet samtidig vil kunne benyttes til at slette andre kontekster end de forældreløse, der ikke længere benyttes. Denne sidste opgave (at slette ikke-benyttede, men dog bundne, kontekster) kan enhver CORBA-programmør imidlertid lave en navneservice-klient til, hvorimod navneservice-klienter, som nævnt, aldrig kan komme i kontakt med forældreløse kontekster, hvorfor det er disse, man som udvikler af navneservicen må tage stilling til. Ulemperne ved at benytte et administratorværktøj er naturligvis dels, at der skal være en menneskelig administrator, der bruger tid på at vedligeholde servicen, og dels at en sådan administrator kan begå fejl og dermed nedlægge benyttede dele af grafen. Endelig kræver et fornuftigt administratorværktøj (som gerne skal have en grafisk brugergrænseflade) en del udviklingsarbejde (hvilket er medvirkende årsag til, at det ikke bliver implementeret i dette projekt).

Som supplement eller alternativ til et administratorværktøj kan man forsøge at indføre automatisk oprydning. Uden administratorværktøjet er dette nødvendigt, og med et administratorværktøj vil automatisk oprydning kunne lette administratorens arbejde. En simpel form for oprydningsmetode er en metode, der simpelt hen nedlægger alle forældreløse kontekster. En sådan metode er forholdsvist simpel at implementere ud fra den persistente repræsentation af navnegrafen. Vi har således i `PersistentTreeStructure`-klassen (se evt. afsnit **3.2 Persistens**) implementeret en privat metode `cleanUpOrphaned`, der rydder op efter følgende fremgangsmåde (kontekster repræsenteres ved deres `ctxID`):

```
Udfør
      Opret liste ubundneCtxs over alle kontekster gemt på disk
      For hver kontekst ctx
            Opret iterator bundneCtxs over kontekster bundet i ctx
            For hvert element bundenCtx i bundneCtxs
                  Fjern bundenCtx fra ubundneCtxs hvis den findes
            Gentag
      Gentag
      Hvis ubundneCtxs ikke er tom
            Slet alle kontekster i listen ubundneCtxs fra disken
Gentag indtil ubundneCtxs er tom
```

Denne fremgangsmåde vil slette alle forældreløse kontekster samt alle disse konteksters sub-kontekster. Nu er spørgsmålet så, hvornår denne metode kan køres. Hvis den køres, mens navneservicen er i drift, risikerer vi at nedlægge kontekster, der refereres udefra, og som faktisk benyttes. I et meget uheldigt tilfælde kunne man sågar risikere, at en klient netop har afbundet en kontekst, med hensigten at binde den under et nyt navn i næste programlinie, men i mellemtiden er konteksten blevet opfattet som forældreløs og er derfor blevet slettet! Dette er naturligvis ikke acceptabelt, og et alternativ kunne derfor være at nøjes med at køre oprydningsmetoden ved servergenstart. Hvis serveren går/lukkes ned, er det alligevel overvejende sandsynligt, at klienter med referencer til forældreløse kontekster, forsøger at benytte disse, opdager at objektet ikke kan nås, og derfor opgiver før serveren startes igen. Derfor forekommer det ikke helt så urimeligt at nedlægge de forældreløse kontekster i denne situation. Løsningen er dog stadig ikke perfekt, da en klient for eksempel kunne have gemt referencen på disk, for at gøre arbejdet færdigt når serveren kører igen, eller en anden navneservice kunne indeholde en reference til konteksten, der således kan slås op på et senere tidspunkt. Imidlertid er den forældreløse kontekst på dette tidspunkt fjernet. Ved servernedbrud kan ovennævnte situation, hvor en kontekst fjernes fordi den er forældreløs et øjeblik, også stadig forekomme, hvis serveren bryder ned på et uheldigt tidspunkt. Brugere af navneservicen må derfor sørge for altid at have en kontekst bundet til et navn, hvis de vil være 100% sikre på ikke at miste konteksten ved en eventuelt uventet servergenstart. En ulempe ved kun at rydde op ved servergenstart er, at hvis serveren kører i lang tid (eller hvis den er replikeret – så den principielt altid kører), vil der ikke blive ryddet op. Endelig kunne der konstrueres et ”administrator-interface”, der muliggør, at et administratorværktøj kan implementeres og køre oprydningsmetoden på administratorens foranledning. Her må dog bemærkes, at en administrator oftest ikke kan være sikker på at undgå ovennævnte problemer med oprydning under kørsel – dog kan vedtagelse af oprydning på et bestemt tidspunkt, for eksempel om natten, til tider være en acceptabel løsning.

Et implementeringsmæssigt mere kompliceret alternativ ville være at indføre en slags ”papirkurv”, hvortil forældreløse kontekster flyttes ved oprydning. Hvis en klient så på et senere tidspunkt forespørger konteksten, kan den genoprettes på POA’en (her kan POA’ens aktiveringspolitik benyttes). Når en forældreløs kontekst ikke har været refereret i et længere tidsrum – eller eventuelt når pladsforbruget i ”papirkurven” bliver for stort, kan konteksterne slettes endeligt – med forholdsvis lille sandsynlighed for, at der er nogen, der benytter dem. Dette ville give en mere hensigtsmæssig opførsel, idet det ville tillade oprydning, mens serveren kører, samt gøre sandsynligheden for at refererede kontekster slettes mindre. Denne fremgangsmåde har vi, af tidsmæssige årsager, ikke implementeret i vores navneservice, men principielt er der ingen hindringer for at gøre det.

3.4 Samtidighedskontrol

Da en af pointerne ved en navneservice er, at den skal kunne servicere flere klienter, der benytter en samlet mængde distribuerede objekter, vil der på et tidspunkt ske det, at flere klienter vil forsøge at benytte den samme ressource. Dette er nødvendigt at forhindre, da det kan have uforudsete konsekvenser, som hvis der for eksempel er en klient, der forsøger at bruge en metode på en kontekst, en anden klient er ved at nedlægge.

I navneservicen er der følgende metoder, der kan komme i konflikt med hinanden:

- `bindImpl` – Binding og genbinding af både kontekst- og objekt-referencer
- `resolve` – Opslag i navneservicen
- `unbind` – Afbinding af referencer
- `destroy` – Nedlæggelse af kontekster

I Java findes der en måde til at undgå, at flere klienter tilgår den samme ressource på samme tid. Dette gøres ved at erklære områder af koden for synkroniseret ved at bruge `sychronized` med et argument, som er den lås, man vælger at bruge. Områder, der er synkroniseret med samme lås, kan ikke udføres af flere tråde på samme tid, hvorimod områder med forskellige låse ikke påvirker hinanden. Der er flere mulige løsninger, til hvordan man kunne synkronisere navneservicen; en løsning kunne være, kun at lade én bruger benytte navneservicen ad gangen, men navneservicen ville da overhovedet ikke være skalerbar. En anden kunne være kun at synkronisere på en sådan måde, så al anden tilgang til navnegrafen blev tilbageholdt, så snart en klient skal til at skrive til navnegrafen. Derved kunne man lade et vilkårligt antal brugere benytte opslagsmetoden `resolve` på samme tid. Det er en helt tredje synkroniserings-måde vi har valgt, og det skyldes, at det, i tilfælde af at man skal til at lave en opdatering eller nedlæggelse af en bestemt del af navnegrafen, ikke umiddelbart er muligt at få at vide, om der er nogen, der er ved at lave et opslag på samme del.

Vores implementation af navneservicen er blevet synkroniseret, så det kun er den del af koden, som står for opslagene i og skrivningen til samme kontekst, som kun kan tilgås af én tråd ad gangen. Idet vi kun har én kontekst låst ad gangen, kan der ikke opstå deadlocks. Deadlocks opstår, når to eller flere tråde gensidigt venter på hinanden. Hvis vi ikke slap synkroniseringslåsen, før det rekursive kald var færdigt, altså låste alle involverede kontekster, da kunne der opstå en situation, hvor to opslag venter på hinanden og derfor ikke kan gennemføres.

Nedenstående er fire stykker pseudokode, som skal beskrive, hvorledes de fire ovennævnte metoder er blevet synkroniseret. Først `bindImpl`,

```
hvis navnekomponent-arrayet kun er een lang
      hvis det er en kontekst
            synkroniser
                  hvis konteksten ikke eksisterer bindes denne
            stop synkronisering
      ellers
            synkroniser
                  hvis objektet ikke eksisterer bindes dette
            stop synkronisering
ellers
      synkroniser
            find ud af om næste bid er en kontekst
            hvis ja
                  få fat i en reference til denne kontekst
      stop synkronisering
      hvis næste bid af navnet var en kontekst
            send resten af navnet til denne kontekst så den kan binde det
      ellers
            kast undtagelse
```

dernæst `resolve`,

```
hvis navnekomponent-arrayet kun er een lang
      synkroniser
            hvis objektet findes returneres en reference til det
      stop synkronisering
ellers
      synkroniser
            find ud af om næste bid er en kontekst
            hvis ja
                  få fat i en reference til denne kontekst
      stop synkronisering
      hvis næste bid af navnet var en kontekst
            send resten af navnet til denne kontekst så den kan finde det
      ellers
            kast undtagelse
```

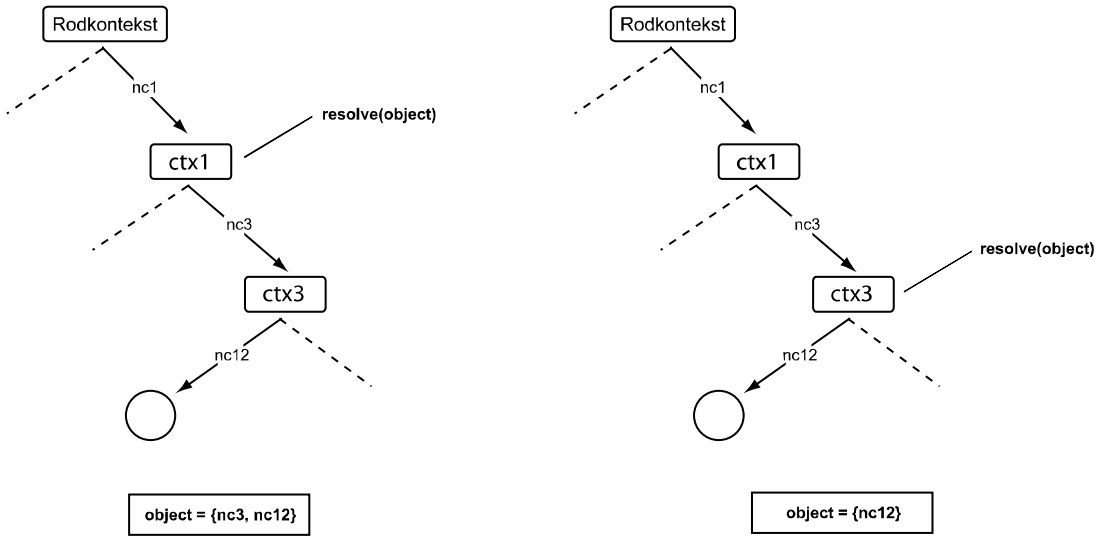
så `unbind`,

```
hvis navnekomponent-arrayet kun er een lang
      synkroniser
            hvis referencen findes fjernes den
      stop synkronisering
ellers
      synkroniser
            find ud af om næste bid er en kontekst
            hvis ja
                  få fat i en reference til denne kontekst
      stop synkronisering
      hvis næste bid af navnet var en kontekst
            send resten af navnet til denne kontekst så den kan afbinde det
      ellers
            kast undtagelse
```

og sidst `destroy`.

```
synkroniser
      prøv at nedlægge konteksten fra POA'en
stop synkronisering
```

Det der er det vigtige, ved denne pseudokode er, at det for det første ikke er hele metoder, der er synkroniseret (bortset fra i `destroy` – og det er pga., at dette er en metode, der kun påvirker den kontekst, den kaldes på), og for det andet, at de dele, der ikke er synkroniseret, er i det tilfælde, at navnekomponent-arrayet ikke kun er ét element stort. Da det, at der er flere komponenter, betyder, at det ikke er i denne kontekst, den ønskede reference befinder sig, undlader vi at låse læse-/skrive-operationerne her og udfører den ønskede metode, på den kontekst, der er den næste i navnekomponent-arrayet. En illustration af dette kan ses på **Figur 3.6**.



Figur 3.6: Til venstre er der lige blevet gjort et kald til metoden `resolve` på konteksten `ctx1`. Mens metoden undersøger det medsendte navnekomponent-array, låses konteksten, så der ikke er andre, der kan benytte læse- og skriveoperationer på netop denne kontekst. Da navnekomponent-arrayet indeholder mere end et objekt, sendes metodekaldet videre, og konteksten `ctx1` kan nu igen udføre læse- og skriveoperationer. På højre del af billedet er det nu `ctx3`, der er låst.

Denne form for synkronisering medfører, at der kun kan udføres en læse- eller skriveoperation ad gangen per kontekst. Det vil med andre ord sige at en helt flad graf, altså ikke indeholdende andre kontekster end rodkonteksten, kun vil kunne servicere én operation ad gangen.

4 Test

I dette afsnit vil vores implementation af navneservicen blive testet i henhold til, om den opfylder de funktionalitetskrav, der, i specifikationen, er til en CORBA-navneservice. Testen foretages ved hjælp af en test-klasse, der afprøver navneservicens funktionalitet, og som undervejs laver udskrifter, der viser, om en given funktionalitet er implementeret rigtigt. Udskrifterne er direkte kopieret fra den benyttede editors prompt og vises som specielt typograferet tekst. Den komplette kode til testklassen kan ses i 12 Appendiks D – Test-klassen kildekode. Desuden er udsnit af den kode, der genererer udskrifterne medtaget sammen med disse. Der er kun tale om kodeudsnit, og trivielle eller gentagne operationer er således udeladt. Hvor der er udeladt kode, angives dette med tre punktummer, og en kommentar, der fortæller, hvad den udeladte kode udfører. I kodeudsnittene er de linier, der forårsager de viste udskrifter, markeret med fed, og linienummereringen fra udskrifterne er angivet, således at kode og udskrifter direkte kan sammenholdes.

4.1 Grundlæggende bind/rebind/resolve

Det første, der testes, er om de grundlæggende metoder, som benyttes til registrering af nye objekter og kontekster, henholdsvist `bind` og `bind_context` og til at finde disse objekter igen via opslag i navneservicen, `resolve`, er funktionelle. Dette gøres, ved at oprette en navnegraf indeholdende både objekter, kontekster og underkontekster for derefter at lave opslag i denne. Efter kørsel af den del af test-klassen, som står for opbygningen af navnegrafen, får vi følgende udskrifter:

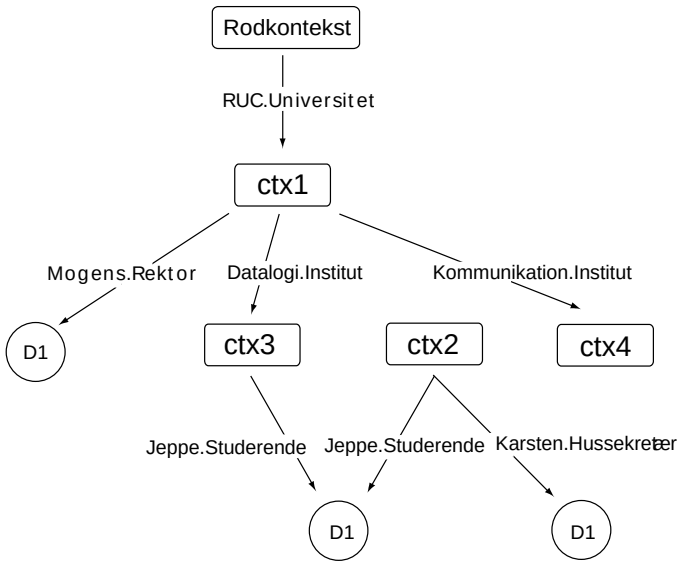
Udskrifter fra programmet

```
1 Context1: Ny kontekst bundet
2 Context3: Ny kontekst bundet
3 Context4: Ny kontekst bundet
4 object1: Bundet!
5 object2: Bundet!
6 object3: Bundet!
7 object3: Bundet!
```

Udsnit af kildekoden

```
  NameComponent nc1 = new NameComponent("RUC", "Universitet");
  NameComponent[] context1 = {nc1};
  NamingContext ctx1 = rootNC.bind_new_context(context1);
1 System.out.println("Context1: Ny kontekst bundet");

  NamingContext ctx2 = rootNC.new_context();

  NameComponent nc3 = new NameComponent("Datalogi", "Institut");
  NameComponent[] context3 = {nc1, nc3};
  NamingContext ctx3 = rootNC.bind_new_context(context3);
2 System.out.println("Context3: Ny kontekst bundet");
  .
3-5 .     binder ctx4 samt objekt1 og object2
  .
  NameComponent nc13 = new NameComponent("Jeppe", "Studerende");
  NameComponent[] object3 = {nc13};
  ctx2.bind(object3, dummyRef);
6 System.out.println("object3: Bundet!");

  ctx3.bind(object3, dummyRef);
7 System.out.println("object3: Bundet!");
```

Hver af disse udskrifter kommer, når en ny kontekst eller objektreference bindes i navneservicen. Navnegrafen ser på nuværende tidspunkt således ud, som det kan ses på **Figur 4.1**.



Figur 4.1: Navnegrafen lige efter oprettelse.

Nøjere beskrevet sker der følgende: Først (lige før udskriften "`Context1: ...`") oprettes et nyt navnekomponent-array indeholdende én enkelt komponent med værdien `RUC.Universitet`. Metoden `bind_new_context` udføres herefter på rootNC, altså rodkonteksten, og med array'et som argument. På forholdvist tilsvarende måde oprettes også de andre kontekster, som ses på figuren, blot med et par enkelte undtagelser; ved oprettelsen af konteksten `ctx2`, benyttes metoden `new_context`, og den er derfor ikke bundet under nogen anden kontekst, og konteksten `ctx3` blev bundet ved at benytte et sammensat navn, altså et navnekomponent-array med mere end en komponent. Efter samme princip bindes de tre objekter `Mogens.Rektor`, `Jeppe.Studerende` og `Karsten.Hussekretær`. Da man ikke kan lave en objekt-binding uden rent faktisk at have et objekt, benyttes en såkaldt dummy (på figuren har den betegnelsen `D1`).

Til at teste opslag i navneservicen og om det er muligt at genbinde en ny kontekst eller et nyt objekt til et allerede benyttet navn, benyttes henholdsvist metoderne `resolve`, `rebind_context` og `rebind`. Det skal her nævnes, at det ikke er muligt at genbinde en kontekst under et navn bundet til et objekt eller vice versa. Udskrifterne fra den del af test-klassen, der tager sig af at vise funktionaliteten af de justnævnte metoder, ses herunder.

Udskrifter fra programmet

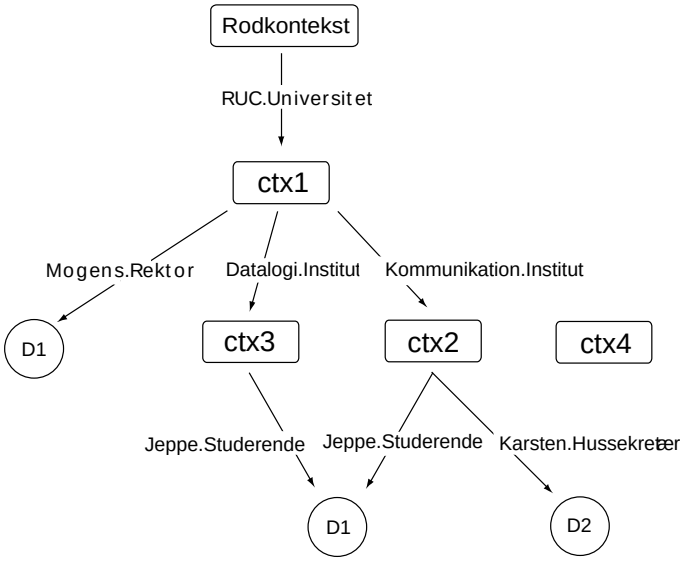
```
 8 Ikke fundet
 9 Fundet
10 Nyt objekt
```

Udsnit af kildekoden

```
      // object4: "RUC.Universitet\Kommunikation.Institut\Karsten.Hussekretær"
      NameComponent[] object4 = {nc1, nc4, nc12};
      try {
            rootNC.resolve(object4);
            System.out.println("Fundet");
      } catch(CosNaming.NamingContextPackage.NotFound nf) {
 8          System.out.println("Ikke fundet");
      } //end try-catch

      NameComponent[] context5 = {nc1, nc4}; // context5: "RUC.Universitet\Kommunikation.Institut"
      rootNC.rebind_context(context5, ctx2);

      try {
            rootNC.resolve(object4);
 9          System.out.println("Fundet");
      } catch(CosNaming.NamingContextPackage.NotFound nf) {
            System.out.println("Ikke fundet");
      } //end try-catch

      org.omg.CORBA.Object obj1 = rootNC.resolve(object4);
      ctx2.rebind(object2, dummyRef2); // object2: Karsten.Hussekretær
      org.omg.CORBA.Object obj2 = rootNC.resolve(object4);
      if (obj1._is_equivalent(obj2)) {
            System.out.println("Samme objekt");
      } else {
10          System.out.println("Nyt objekt");
      } // end if
```

I den første linie (linie 8) prøver vi at finde en reference til `Karsten.Hussekretær` via `Kommunikation.Institut`, som på dette tidspunkt peger på `ctx4` (se **Figur 4.1**). Dette gøres for at vise, at det ikke er bundet her, og der returneres derfor "`Ikke fundet`". Bagefter genbindes `Kommunikation.Institut` til at pege på `ctx2`, og vi prøver da igen, via `Kommunikation.Institut`, at finde `Karsten.Hussekretær`. Denne gang lykkedes det, og udskriften er derfor "`Fundet`". Tredje udskrift (linie 10) viser, at det er muligt, korrekt at genbinde objekt-navne til nye objekter. Det, at der står "`Nyt objekt`", henviser til en sammenligning af, om det dummy-objekt (`D2`), som navnet `Karsten.Hussekretær` er blevet genbundet til at pege på, er det samme, som det objekt den pegede på før.

Efter denne del af test-klassen er udført, har navnegrafen ændret sig en smule. Ændringerne kan ses på **Figur 4.2**.



Figur 4.2: Navnegrafen, efter navnet `Kommunikation.Institut`, er blevet genbundet til at pege på konteksten `ctx2`, og efter at navnet `Karsten.Hussekretær` er blevet genbundet til at pege på objektet `D2`.

4.2 Hukommelsesstyring

Begrebet hukommelsesstyring dækker her over behovet for at fjerne, det vil sige afregistrere og slette, ikke længere benyttede kontekster samt bindinger.

Det bør vises, at det er muligt, at nedlægge kontekster (via metoden `destroy`) og ikke længere nødvendige objektbindinger (via metoden `unbind`). Efter kørsel af den, i forhold til dette, relevante del af test-klassen, kommer følgende udskrifter:

Udskrifter fra programmet

```
11 Ctx3 ikke nedlagt
12 Før unbind: Object3 fundet via Ctx2
13 Før unbind: Object3 fundet via Ctx3
14 Efter unbind: Object3 fundet via Ctx2
15 Efter unbind: Object3 ikke fundet via Ctx3
16 Ctx3 nedlagt
17 Reference til Ctx3 modtaget
18 Ctx3 eksisterer ikke
19 Begynder oprettelsen af nye kontekster...
20 ...oprettelse slut.
```

Udsnit af kildekoden

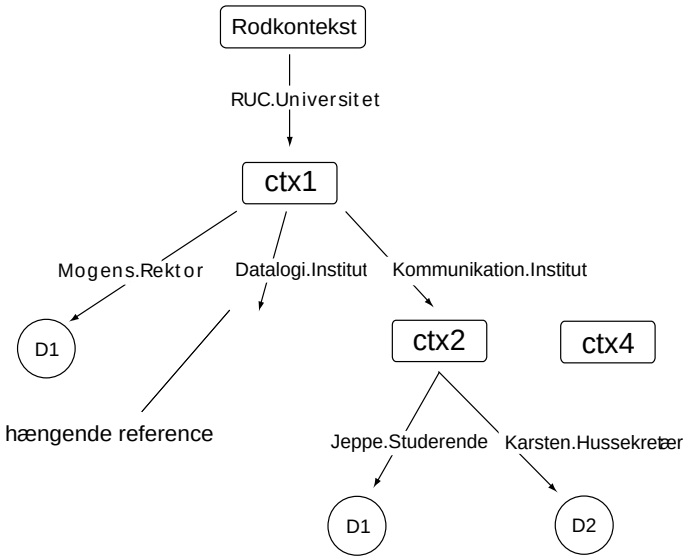
```
   try {
        ctx3.destroy();
        System.out.println("Ctx3 nedlagt");
   } catch(CosNaming.NamingContextPackage.NotEmpty ne) {
11      System.out.println("Ctx3 ikke nedlagt");
   } //end try-catch

   try {
        ctx2.resolve(object3); // object3: Jeppe.Studerende
12      System.out.println("Før unbind: Object3 fundet via Ctx2");
   } catch(CosNaming.NamingContextPackage.NotFound nf) {
        System.out.println("Før unbind: Object3 ikke fundet via Ctx2");
```

```
        } //end try-catch
        .
    13  .     Finder ikke Object3 via ctx3
        .
        ctx3.unbind(object3);

        try {
              ctx2.resolve(object3);
    14        System.out.println("Efter unbind: Object3 fundet via Ctx2");
        } catch(CosNaming.NamingContextPackage.NotFound nf) {
              System.out.println("Efter unbind: Object3 ikke fundet via Ctx2");
        } //end try-catch
        .
    15  .     Finder object3 via ctx3
        .
        try {
              ctx3.destroy();
    16        System.out.println("Ctx3 nedlagt");
        } catch(CosNaming.NamingContextPackage.NotEmpty ne) {
              System.out.println("Ctx3 ikke nedlagt");
        } //end try-catch
        .
    17  .     Slå ctx3 op og gem referencen i "deadLink"
        .
        try {
              deadLink.new_context();
              System.out.println("Ctx3 eksisterer stadig");
        } catch(org.omg.CORBA.OBJECT_NOT_EXIST one) {
    18        System.out.println("Ctx3 eksisterer ikke");
        } //end try-catch
        .
 19-20  .     Opret 100 nye kontekster
        .
```

For at kunne vise at metoden virker efter specifikationen, forsøger vi først, ved hjælp af `destroy`, at nedlægge konteksten `ctx3`. Dette lader sig ikke gøre, da `ctx3` ikke er tom (linie 11), og indholdet af `ctx3`, hvilket vil sige objekt-referencen `Jeppe.Studerende`, skal derfor først afbindes, via metoden `unbind`. Før og efter kaldet til `unbind` undersøges det, om det er muligt at skaffe en reference til objektet via de to kontekster `ctx2` og `ctx3`. Det er muligt for begge kontekster at skaffe en reference før kaldet, men kun muligt for `ctx2` efter kaldet, så metoden `unbind` virker efter hensigten. Efter dette forsøger vi så igen at nedlægge `ctx3`. Dette er denne gang muligt som udskriften viser (linie 16). For at vise at `destroy` ikke bare har gjort som `unbind`, undersøger vi, om det stadig er muligt, fra `ctx1`, at få en reference til `ctx3`, da dette burde være tilfældet. Det er det (linie 17), og at denne reference ikke peger på noget kan ses i line 18, hvor vi forsøger at benytte en metode på det nedlagte kontekst-objekt. Afsluttende vises, at der rent faktisk frigives systemressourcer. Dette gøres ved at fremprovokere Garbage Collectoren til at starte ved at oprette en masse tomme kontekster - i linie 19-20 ses oprettelsens start og slutning fra klienten, og på **Figur 4.3** ses i den næstsidste linie en udskrift fra kontekstens `finalize` metode, som kun kaldes lige før konteksten Garbage Collectes.

Figur 4.3: De to nederste linier viser at den i testen nedlagte kontekst og iterator bliver Garbage Collectet.

Vi er nu nået frem til en navnegraf, der ser ud som på **Figur 4.4**. De tomme kontekster, hvis eneste formål var at fremprovokere Garbage Colletoren, er ikke med på figuren, da de ikke er relevante for resten af testen. Samtidig vil det ikke være muligt at få fat på disse tomme kontekster igen, da de er oprettet på en sådan måde, at referencen til dem ikke er blevet gemt.



Figur 4.4: Navnegrafen efter `Jeppe.Studerende` **er blevet afbundet fra** `ctx3`**, og efter** `ctx3` **er blevet nedlagt.**

Da det også er muligt at oprette iteratorer til at gennemløbe bindingerne i en kontekst, og da disse også skal nedlægges på et tidspunkt, er denne del af hukommelsesstyringen også vigtig at teste. Udskrifter fra kørsel af denne del af test-klassen ses nedenfor.

Udskrifter fra programmet

```
21 Kontekst: Datalogi.Institut
22 Kontekst: Kommunikation.Institut
23 Objekt: Mogens.Rektor
24 Starter med at vente på iteratorens død...
25 ...iteratoren burde nu være død
26 Begynder oprettelsen af nye kontekster...
27 ...oprettelse slut.
```

Udsnit af kildekoden

```
        BindingListHolder bl = new BindingListHolder();
        BindingIteratorHolder blIt = new BindingIteratorHolder();
        ctx1.list(10, bl, blIt);
        Binding[] bindings = bl.value;

        for (int i=0; i < bindings.length; i++) {
              if (bindings[i].binding_type == BindingType.ncontext) {
21-22               System.out.println( "Kontekst: " + rootNC.to_string(bindings[i].binding_name));
              } else {
   23               System.out.println("Objekt: " + rootNC.to_string(bindings[i].binding_name));
              } //end if
        } //end for
        .
24-27   .     Venter 65 sekunder på iteratorens død, og opretter derpå 100 nye ubundne kontekster
        .
```

Først oprettes en bindings-iterator over indholdet af konteksten `ctx1` - et gennemløb af denne kan ses i linierne 21 til 23. Da iteratoren er implementeret til at slette sig selv efter 60 sekunder, ventes (i linie 24-25) på dette, og for, som i testen af konteks-hukommelsesstyringen, at fremprovokere Garbage Collectoren, oprettes der en masse tomme kontekster. Linerne 26-27 viser oprettelsen af de tomme kontekster, mens sidste linie på **Figur 4.3** viser at iteratoren bliver Garbage Collectet.

4.3 Persistens/servergenstart

Som afslutning af testen undersøges om metoderne, der skal stå for at genopbygge navnegrafen i hukommelsen efter servergenstart, er implementeret korrekt. På dette tidspunkt er navneservice-serveren blevet genstartet, og alle tidligere referencer til objekter i navnegrafen er derved kasseret. Kørsel af den respektive del af test-klassen giver følgende udskrifter:

Udskrifter fra programmet

```
28 Indholdet af rod-konteksten:
29     Kontekst: RUC.Universitet
30 Indholdet af konteksten RUC.Universitet:
31     Kontekst: Datalogi.Institut
32     Kontekst: Kommunikation.Institut
33     Objekt: Mogens.Rektor
```

Udsnit af kildekoden

```
       BindingListHolder bl = new BindingListHolder();
       BindingIteratorHolder blIt = new BindingIteratorHolder();
       rootNC.list(10, bl, blIt);
       Binding[] bindings = bl.value;

       NamingContextExt lowerCtx = null;
   28  System.out.println("Indholdet af rod-konteksten:");
       for (int i=0; i < bindings.length; i++) {
             if (bindings[i].binding_type == BindingType.ncontext) {
   29              System.out.println("    Kontekst: " + rootNC.to_string(bindings[i].binding_name));
                   lowerCtx = NamingContextExtHelper.narrow(rootNC.resolve(bindings[i].binding_name));
             } else {
                   System.out.println("    Objekt: " + rootNC.to_string(bindings[i].binding_name));
             } //end if
       } //end for

       lowerCtx.list(10, bl, blIt);
       bindings = bl.value;
   30  System.out.println("Indholdet af konteksten RUC.Universitet:");
       for (int i=0; i < bindings.length; i++) {
             if (bindings[i].binding_type == BindingType.ncontext) {
 31-32             System.out.println("    Kontekst: " + rootNC.to_string(bindings[i].binding_name));
             } else {
   33                System.out.println("    Objekt: " + rootNC.to_string(bindings[i].binding_name));
             } //end if
       } //end for
```

Hvis navnegrafen er blevet opbygget korrekt efter servergenstarten, så bør det forventes, at det er muligt, fra roden, at få fat på referencen `RUC.Universitet`, og via denne køre en metode på den dertil knyttede kontekst. I linie 28 og 29 er metoden `list` lige blevet udført på den genstartede navneservice's rodkontekst, og udskriften viser hvilke bindinger, der er i rodkonteksten. `RUC.Universitet` er, som forventet, den eneste binding i roden, og via kontekstreferencen, der er bundet til dette navn (den tidligere `ctx1` på **Figur 4.1** til **Figur 4.4**), udføres metoden `list` igen. Resultatet ses i linierne 31 til 33, som er en udskrift af de bindinger, der er registreret i konteksten. Da alle de tre referencer, som måtte forventes at være bundet i konteksten `RUC.Universitet`, er til stede, og da det er lykkedes at kalde en metode på denne kontekst, kan man konkludere at navneservicen er persistent.

5 Diskussion

Vi har i denne rapport valgt at implementere en navneservice, der opfylder kravene til en CORBA-navneservice. Specifikationen [OMG; 2001ns] beskriver, hvilke metoder en implementation skal have, og hvilken funktionalitet disse skal have. Denne specifikation beskriver også en række punkter, som er implementationsspecifikke. Altså funktionalitetsvalg der er op til den implementerende part. Vi vil i dette afsnit diskutere de problemstillinger, der har været i forbindelse med at skulle opfylde kravene.

De steder hvor funktionalitet var bestemt på forhånd, har vi fulgt specifikationen. Der er kun en undtagelse hvilken er `to_url` metoden. Vi har undladt at implementere den, da den ligger i periferien af vores projekt og ikke har nogen umiddelbar funktion i vores navneservice. At vi har valgt, at implementere de andre udvidede (Ext) metoder skyldes, at vi benytter dem som hjælpemetoder i nogle af de andre navneservice-metoder.

Som tidligere beskrevet i rapporten havde vi følgende valgmuligheder i forhold til funktionaliteten af interfacet:

- Begrænsninger på navnekomponenterne.
- Antallet af bindinger i en kontekst.
- Nedlæggelse af forældreløse kontekster.
- Nedlæggelse af iteratorer.
- Itereators opførsel i forbindelse med ændringer i aktuelle kontekst.

Da persistensen i vores navneservice kommer ved at gemme alle bindinger på en fast disk, og da de navne der gemmes bliver fire gange så lange, på grund af måden vi gemmer dem på, kan der opstå systemspecifikke problemer. Disse problemer kan opstå, hvis systemet ikke tillader filnavne over en vis længde – i vores system (Windows NT/x86) fik vi problemer, hvis længden af navnekomponenten (`kind` + `id`) var over 50 karakterer lang. I tilfælde af for lange navnekomponenter kaster `PersistentTreeStructure` en IO-undtagelse og undlader at skrive til disken. Dette resulterer i, at hvis dette var en kontekst, ville alt, der efterfølgende blev bundet i den samt den selv, ikke være persistent. Det skal også nævnes, at måden, vi gemmer vores navne på disken betyder, at filnavne med bindestreger kan opstå. Vi sætter ingen begrænsninger på de karakterer, der må benyttes i en navnekomponent.

Vi har ikke set det nødvendigt at sætte begrænsninger på antallet af tilladte bindinger i en kontekst. Dette kunne medføre hastighedsnedsættelse ved metoder der kræver opslag i en kontekst med et uhensigtsmæssigt stort antal bindinger.

I vores implementation har vi valgt at nedlægge forældreløse kontekster ved servergenstart. Dette er ikke en optimal løsning, men det sørger for, at vi har en mulighed for at rydde op i navnegrafen. Konsekvensen af denne funktion er at vi ikke kan sikre at vores klienter, deriblandt andre navneservicer, kan få fat i deres ubunde kontekster til et givent tidspunkt, da en servergenstart vil medføre at disse fjernes. En god ting ved denne funktion er, at skyldes et servernedbrud manglende systemressourcer, så kan en del af disse frigøres ved den oprydning, der foretages ved genstart. Andre løsninger til problemet med forældreløse kontekster beskrives i **7 Perspektivering**.

At vi har valgt at implementere nedlæggelse af iteratorer ved hjælp af en timer som venter 60 sekunder fra sidst udførte iterator-metode, før den nedlægger iteratoren, har den årsag, at dette er en overkommelig og funktionel løsning, som ikke kræver, at der holdes rede på iteratorene. Samtidig kræves det ikke, at der køres en separat oprydningsalgoritme til at rydde op i mængden af iteratorer. At vi hverken har valgt længere eller kortere tid før timeren nedlægger iteratoren, har den årsag, at det giver tid nok til, at en bruger kan have problemer med sin forbindelse til navneservice-serveren, mens det giver mindre chance for, at systemressourcerne opbruges, end et større tidsrum ville have gjort. Det sagt, så er 60 sekunder, ikke et tidsrum vi har valgt udfra større tests, men blot et vi har valgt udfra ovenstående overvejelser. Tidsrummet bør i praksis afgøres efter serverressourcer og belastning.

Ændres en af bindingerne i en kontekst, hvorpå en bruger benytter en iterator, vil det ingen indflydelse have på iteratorens opførsel. Grunden til dette er, at iteratoren konstrueres således, at den indeholder en liste over alle bindinger fra den kontekst den blev oprettet på. Dette vil så betyde, at i tilfælde af at der slettes en binding i en kontekst, så vil en bruger med en iterator oprettet før ændringen blev effektueret, stadig få returneret den slettede binding.

En mulig ulempe ved den nuværende implementation er, at koden ikke er portabel på tværs af Java-ORB'er. Dette skyldes, at vi, i forbindelse med initialisering af ORB'en, benytter nogle parametre, der er specifikke for SUN's ORB, der følger med jdk. Desuden benytter vi metoden `orb.register_initial_references` til at gøre vores navneservice tilgængelig via Corbaloc. Denne metode er også specifik for SUN's ORB. Derudover fanger vi `ClassCastException` i forbindelse med metoden `poa.servant_to_id` kald på en reference til et objekt, der ikke eksisterer på POA'en, imidlertid er det heller ikke specificeret i CORBA-standarden, at denne undtagelse skal kastes, og dette er derfor heller ikke portabelt.

Et andet portabilitetsproblem omhandler forskellige operativsystemer. Da navneservicen er skrevet i Java, burde den kunne køre på alle platforme, hvor der er implementeret en JVM. Imidlertid benytter vores persistensklasse sig af det faktum, at man i windows-filnavne bruger ”\” som separator i stier. Dette er ikke tilfældet på alle andre operativsystemer (f.eks. ikke på UNIX/Linux) og er derfor ikke portabelt på tværs af disse. Hvis navneservicen skal flyttes til en anden platform, er det således nødvendigt at rette i implementationen af persistensklassen.

Vores navneservice er implementeret således, at hele navnegrafen altid er repræsenteret i hukommelsen, når servicen kører. Dette betyder, at enhver kontekst, til ethvert tidspunkt, kan tilgås hurtigt. Imidlertid sætter det også en del krav til serverressourcerne, hvis der er tale om en stor navnegraf. Afhængig af hvordan navneservicen anvendes, kan man sagtens forestille sig, at det ikke er hele navnegrafen, der anvendes ofte. Derfor kunne det muligvis være fordelagtigt at have en del af navnegrafen på en fast disk i stedet for i hukommelsen, og så blot lade POA’en aktivere konteksterne, når de blev forespurgt.

Alt i alt har vi bestræbt os på at implementere en stabil og funktionel navneservice, som samtidig opfylder CORBA standardens krav. Mange af de ovenstående designvalg er således taget med dette mål i tankerne.

6 Konklusion

Vi har implementeret en persistent CORBA-navneservice, der opfylder kravene i CORBA-navneservice v. 1.1 specifikationen [OMG; 2001ns] på alle punkter undtaget implementation af metoden `to_url` i `NamingContextExt`-interfacet.

7 Perspektivering

Vi har haft flere mål for vores navneservice. Blandt andet har det været vores mål, at gøre vores navneservice pålidelig og skalerbar, af denne årsag har vi gjort vores navneservice persistent, foruden at vi har givet den samtidighedskontrol. Et andet designmål for vores service er stabilitet, dette er delvist opnået, gennem vores håndtering af forældreløse kontekster og iteratorer beskrevet senere i dette afsnit.

Skulle man ønske at forbedre stabiliteten og samtidig skalerbarheden af navneservicen, ville det være indlysende at gøre den replikeret. Replikering giver servicen større stabilitet, ved at klienter kan benytte replikerede servere ved servernedbrud. Samtidig kan replikering gøre servicen mere skalerbar ved at tilbyde belastningsfordeling. Dog er implementering af en replikeret navneservice ikke trivielt. Der rejses således en del problemstillinger: Hvordan skal de replikerede servere holde sig opdateret, skal de være synkrone eller asynkrone. Hvordan skal klienter og servere kommunikere, er det klientens opgave at skifte server, hvis den første server ikke svarer, eller skal der eksistere et mellemled, der også tager sig af belastningsfordelingen.

Vi har i vores rapport belyst og diskuteret et ikke ubetydeligt problem ved CORBA-navneservicen: Forældreløse kontekster. Fejlbehæftede klienter og uhensigtsmæssig brug af navneservicen kan fører til ubunde kontekster, som er ubrugelige for navneservicens brugere. Som løsning til dette problem har vi implementeret en oprydningsmetode, der fjerner forældreløse kontekster ved servergenstart.

Vi har, projektet igennem, diskuteret andre løsninger end den anvendte, men problemet er som sagt, at en navneservice ikke kan vide, om en forældreløs kontekst er bundet i andre navneservicer eller er benyttet af andre brugere. En mulig løsning på problemet er at udvikle et administratorværktøj, der giver en administrator mulighed for at selv at starte oprydningsalgoritmen. Dette værktøj kunne desuden tilbyde et grafisk billede af navnegrafen, der kunne gøre ham i stand til at identificere fejloprettede forældreløse kontekster og en metode til at fjerne disse. Foruden at løse problemet med forældreløse kontekster, kunne programmet gøre administratoren i stand til at indstille levetiden på oprettede iteratorer, og samtidig give ham et overblik over aktive iteratorer. Et måske bedre løsningsforslag er at benytte en ”papirkurv” til opryddede forældreløse kontekster. En sådan ”papirkurv” skulle gemme disse forældreløse kontekster på disken, og de skulle således kun aktiveres, hvis de blev kaldt. Samtidig skulle de slettes fra papirkurven, hvis serveren havde få systemressourcer tilbage. Dette ville stadig kunne forårsage fejl, men i langt mindre grad end den anvendte løsning, og samtidig ville det medføre at oprydningen af forældreløse kontekster blev automatisk, og oven i købet ville den kunne fungere uden servergenstart.

I dette afsnit har vi redegjort for nogle af de udvidelser, navneservicen kan udbygges med. Det ville først og fremmest være vigtigt at udbygge navneservicen med et administratorværktøj, således at servicen i sig selv kunne gøres stabil. Efterfølgende kunne servicen replikeres, hvis den blev benyttet af et stort antal brugere.

8 Kilder

Følgende kildeliste er arrangeret således, at kilder der direkte henvises til i rapportteksten, har en henvisningskode angivet i kantede paranteser. Øvrigt materiale, der hovedsageligt er medtaget fordi det har dannet baggrund for at forstå OMG-dokumenterne, er angivet sidst.

8.1 Lænker

[OMG; 2002] - *Object Management Group*

http://www.omg.org/

[OMG; 2001a] - *CORBA/IIOP 2.6.1*

http://www.omg.org/technology/documents/formal/corba_iiop.htm

[OMG; 2001b] - *CORBA/IIOP 2.6.1 - Kapitel 11: POA*

http://www.omg.org/technology/documents/formal/corba_iiop.htm

[OMG; 2001c] - *CORBA/IIOP 2.6.1 - Kapitel 12 og 13: Interoperabilitet*

http://www.omg.org/technology/documents/formal/corba_iiop.htm

[OMG; 2001d] - *CORBA/IIOP 2.6.1 - Kapitel 4: ORB-interface*

http://www.omg.org/technology/documents/formal/corba_iiop.htm

[OMG; 2000] - *CORBA/IIOP 2.4*

http://www.omg.org/cgi-bin/doc?formal/00-10-01

[OMG; 2001ns] - *Name Service 1.1*

http://www.omg.org/technology/documents/formal/naming_service.htm

[IDL; 2002] - *Teach yourself CORBA in 14 days – day 3: Mastering the Interface Definition Language (IDL)*

http://www.kaposnet.hu/books/corba14/ch03/ch03.htm

Borland Enterprise Server doc. Indholdsfortegnelse
http://info.borland.com/techpubs/books/bes/htmls/DevelopersGuide5/DevelopersGuide/contents.html

Borland Enterprise Server doc. Kap. 10: ServerBasics (POA)
http://info.borland.com/techpubs/books/bes/htmls/DevelopersGuide5/DevelopersGuide/servers.html

Borland Enterprise Server doc. Kap. 11: Using POA's
http://info.borland.com/techpubs/books/bes/htmls/DevelopersGuide5/DevelopersGuide/poa.html

Borland Enterprise Server doc. Kap. 18: Using the Naming Service
http://info.borland.com/techpubs/books/bes/htmls/DevelopersGuide5/DevelopersGuide/namesrvc.html

8.2 Bøger

[Coulouris; 2001]
Coulouris, G. et al. (2001). *Distributed Systems – Concepts and Design*, Addison-Wesley. ISBN: 0201619180

[Mowbray; 1997]
Mowbray, T. J. & Ruh, W. A. (1997). *Inside CORBA*, Addison-Wesley. ISBN: 0201895404

Rosenberger, J. (1998). *Teach yourself CORBA in 14 days*, Sams Publishing. ISBN: 0672312085

9 Appendiks A – Navneservice dokumentation

Der pointeres i OMG's navneservicespecifikation en række krav til dokumentationen af en navneservice. For at navneservicen opfylder CORBA-navneservice v. 1.1 specifikationen, er en dokumentation, der opfylder disse krav, påkrævet. Vores rapport forelægger som egentlig dokumentation, og dette appendiks er således skrevet ene og alene for at redegøre for de krav, der er sat af OMG.

Her følger dokumentationskravene med dokumentation:

- Begrænsninger på karakterer der benyttes i navnekomponenterne.
  - Vores navneservice sætter ingen begrænsninger på de karakterer, der benyttes i navnekomponenterne. Eneste begrænsning er, at strengene i et navnekomponent ikke må være `null`.
- Begrænsning af længden af strengene i navnekomponenterne.
  - Længden af strengene i vores navnekomponenter er begrænset, i det vores kontekster og objektreferencer gemmes på den faste disk under en byte-ificeret udgave af dets navnekomponent. Længden er således maskinspecifik. Eksempelvis tillader et Windows NT-system ikke, at `kind` og `id` felterne tilsammen fylder mere end ca. 50 karakterer.
- Begrænsninger på antallet af navnekomponenter i et navn.
  - Der er ingen begrænsninger på antallet af navnekomponenter i et navn.
- Begrænsninger på antallet af bindinger.
  - Ingen anden end den ressourcerne på serversiden sætter.
- Begrænsninger på antallet af kontekster.
  - Samme svar som forrige.
- Redskaber til rådighed til at håndtere ubundne kontekster.
  - Der findes ingen redskaber, der af brugeren kan sættes til at fjerne forældreløse kontekster. Dog rydder navneservicen ved genstart navnegrafen for alle forældreløse kontekster – de er ikke nødvendigvis ubundne, da de kan være bundet i andre navneservicer.
- Politik angående potentielt forældreløse kontekster.
  - Politikken er sådan, at de tolereres, i det navneservicen ikke kan være sikker på, at de er ubundne. Dog fjernes alle forældreløse kontekster ved proces genstart.
- Politik angående fjernelse af iteratorer der ikke menes at være i brug.
  - Hvis en iterator ikke har været benyttet i 60 sekunder nedlægges den af navneservicen. Tages den i brug inden de 60 sekunder er gået nulstilles timeren, hvorved nye 60 sekunder tildeles iteratoren.
- Under hvilke omstændigheder, hvis nogen, kastes undtagelsen `CannotProceed`.
  - Denne undtagelse kastes ikke af vores navneservice.

Det skal tilføjes, at vores navneservice ikke opfylder CORBA-navneservice v. 1.1 specifikationen [OMG; 2001ns] , da vi har undladt at implementere en metode i `NamingContextExt` interfacet. Den undladte metode er `to_url`.

10 Appendiks B – Navneservice IDL

Dette er OMG's interface til CORBA-navneservice v. 1.1 [OMG; 2001ns] . Nedenstående tabel er et indeks over metoderne i interfacet.

11 Appendiks C – Navneservice kildekode

For at gøre det overskueligt er der, før udskriften af hver klasse, en liste over metodernes placering i kildekoden.

Klassen ContextNode

048	konstruktor
057	main
135	bind
144	rebind
155	bind_context
164	rebind_context
175	resolve
211	unbind
247	new_context
256	bind_new_context
263	destroy
287	list
318	to_string
327	to_name
335	to_url
344	resolve_str
357	activateNamingContext
370	activateNamingContextWithID
391	generateUniqueID
419	bindImpl
498	checkAlreadyExists
523	static_to_string
548	addEscape
571	static_to_name
585	getNcStrings
616	extractComponents
644	indexOfNotEsc
672	removeEsc
698	testName
708	regenerateGraph
779	indre klasse BindingIteratorImpl
796	konstruktor
836	next_one
856	next_n
887	hasMore
892	destroy
908	finalize
916	destroyIterator
925	indre indre klasse Timer
931	Run

Klassen PersistentTreeStructure

024	konstruktor
046	createCtx
055	writeObject
060	writeContext
065	destroyContext
075	getIterator
080	getObjIterator
085	getCtxIterator
093	cleanUpOrphaned
149	writeReference
177	bytesToString
200	stringToBytes
213	indre klasse NameGraphContextIteratorImpl
220	next
228	hasNext
234	Indre klasse NameGraphBindingIteratorImpl
241	konstruktor
267	next
275	hasNext
281	Indre klasse ExtensionFilter
288	konstruktor
297	accept

Klassen NameBinding

019	konstruktor (med to parametre)
025	konstruktor (uden parametre)

Klassen NCWrapper

017	konstruktor
024	hashCode
031	getNC
042	equals

Interfacet PersistentNameGraph

014	createCtx
023	writeObject
034	writeContext
039	destroyContext
045	getIterator
053	getObjIterator
061	getCtxIterator

Interfacet NameGraphBindingIterator

013	next
018	hasNext

Interfacet NameGraphContextIterator

014	next
019	hasNext

12 Appendiks D – Test-klassen kildekode

Dette er kildekoden til vores test-klasse.