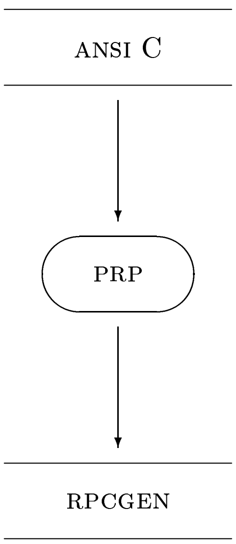
Vereenvoudiging van gedistribueerde Applicatie ontwikkeling

Integratie van Sun RPC en C

Doctoraalscriptie van Aske Plaat

24 juni 1992



Erasmus Universiteit Rotterdam
Faculteit der Economische Wetenschappen
Studienummer 53697

Vakgroep Informatica
Docent:
Ir. J. van den Berg

Inhoud

Lijst van figuren

Voorwoord

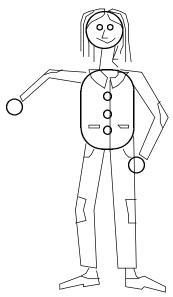
Het onderwerp van deze scriptie is voortgekomen uit het werk van het 'Netteam' (zie [Chin] in de literatuurlijst). Voor het door Jan van den Berg gegeven werkcollege 'Netwerken' hebben zij een gedistribueerd programma geschreven voor de Sun omgeving. Zij hebben de betekenis van 'matige transparantie' zeer concreet moeten ervaren. Met name de door RPCGEN gegenereerde stubs bleken ingewikkeld om te gebruiken, en aanleiding te geven tot veel frustraties bij het debuggen.

Hun ervaringen vormden de aanzet voor het onderwerp van deze scriptie. Ik wil hen hier bedanken voor hun doorzettingsvermogen en voor het met mij delen van hun ervaringen.

Jan van den Berg was de begeleider van dit onderzoek. Tijdens het hele traject heeft hij steeds kritisch de vorderingen gevolgd. Bij de keuze van het onderwerp, het bespreken van het programma, en met name bij het theoretisch gedeelte heeft hij, met de hem eigen positieve instelling, de ontwikkeling van het werk nauwgezet en met interesse begeleid. Hij heeft telkens het belang van precieze formuleringen en duidelijke onderbouwingen van uitspraken benadrukt.

Voor zijn inzicht, kennis van details, het in het oog houden van de rode draad, en vooral voor zijn positieve, kritische instelling wil ik hem op deze plaats hartelijk bedanken.

Door zijn commentaar heeft Arie de Bruin voor grote en kleine verbeteringen in de tekst gezorgd, met name op het gebied van parallellisme, operating systems en de semantiek van C. Hiervoor—en voor zijn enthousiasme—wil ik hem graag bedanken.



En tenslotte wil ik voor alle liefde, steun, slimme opmerkingen, en het onweerlegbare logisch inzicht nevenstaande figuur graag bedanken.

24 juni 1992.

Aske Plaat
Willem van Hillegaersbergstraat 34
3051 RK Rotterdam
email: plaat@theory.lcs.mit.edu

Hoofdstuk 1

Inleiding

In deze scriptie wordt onderzocht hoe het schrijven van toepassingsprogramma's voor gedistribueerde systemen zo goed mogelijk ondersteund kan worden. Hiervoor is een literatuurstudie verricht naar de eisen die men kan stellen aan een applicatieontwikkelomgeving voor gedistribueerde programma's. Vervolgens is de Sun omgeving op deze criteria beoordeeld. Voor de verbetering van een aantal zwakke punten is een programma geschreven dat beoogt het schrijven van gedistribueerde programma's te vereenvoudigen.

In dit hoofdstuk wordt de structuur van de scriptie besproken.

1.1 Achtergrond

Sinds de opkomst van computernetwerken wordt er veel onderzoek gedaan naar gedistribueerde computersystemen. Dit zijn systemen die zijn opgebouwd uit meerdere processoren met gescheiden geheugen die door middel van een netwerk zijn verbonden. De client/server-architectuur wordt bij dit soort systemen vaak toegepast. Met behulp van gedistribueerde operating systems en -talen is het mogelijk de totale werklast van een systeem te verdelen over de beschikbare processoren. Door programma's op verschillende processoren met lokaal geheugen uit te voeren heeft het systeem als geheel een grote verwerkingskracht [Bal].

Sommige systeemprogramma's kunnen om redenen van snelheid en betrouwbaarheid over meerdere processoren verdeeld worden. Voorbeelden hiervan, zoals fileservers, zijn in [Coulouris] beschreven. Ook veel applicatieprogramma's kunnen sneller en/of betrouwbaarder worden door ze gedistribueerd uit te voeren. Hiertoe moeten applicatieontwikkelaars in staat worden gesteld om hun programma's geschikt te maken voor gedistribueerde architecturen. Ter ondersteuning hiervan moeten adequate systeemontwikkelgereedschappen ontwikkeld worden. Deze gereedschappen of *tools* zijn het onderwerp van deze scriptie.

> *Het onderwerp van deze scriptie is hoe men de ontwikkeling van applicaties voor gedistribueerde computersystemen in het alge-*

meen en de Sun omgeving in het bijzonder adequaat kan ondersteunen.

1.2 Probleemstelling

Het schrijven van programma's is voor gedistribueerde omgevingen veel ingewikkelder dan voor sequentiële omgevingen (zie § 1.4). Sun biedt de applicatieprogrammeur hulpmiddelen om het ontwikkelproces te vereenvoudigen. In deze scriptie worden sterke en zwakke punten van deze hulpmiddelen geïdentificeerd, en er worden een aantal verbeteringen voorgesteld.

probleemstelling

De probleemstelling luidt als volgt:

- Aan welke criteria moet een goede ontwikkelomgeving voor gedistribueerde applicaties voldoen?
- In hoeverre voldoet de Sun ontwikkelomgeving aan deze criteria?
- Op welke wijze kan het programmeren voor de Sun omgeving vereenvoudigd worden?

De criteria worden in hoofdstuk 3 besproken. De structuur van de Sun ontwikkelomgeving wordt in § 4.1 behandeld. Op basis van de beoordeling van § 4.2 bestaat de wens om de Sun ontwikkelomgeving te verbeteren.

Hoofdstuk 5 beschrijft de verbeteringen, en de implementatie ervan. In hoofdstuk 6 vindt de beoordeling van de verbeteringen plaats op basis van de criteria uit § 3.3 en § 3.4.2. In dat hoofdstuk, en in hoofdstuk 7, is het antwoord op de laatste vraag van de probleemstelling te vinden.

1.3 Doelstelling

Deze scriptie onderzoekt applicatieontwikkeling voor gedistribueerde systemen. Het doel van deze scriptie is tweeledig: ten eerste het kunnen beoordelen van ontwikkelomgevingen, en ten tweede het verbeteren van (aspecten van) de Sun omgeving. Het doel van applicatieontwikkelomgevingen is het schrijven van toepassingsprogramma's zo eenvoudig mogelijk te maken. Om te kunnen beoordelen of dit doel bereikt wordt, wordt een lijst criteria aangelegd. Vervolgens wordt een concrete omgeving bekeken: de Sun omgeving. Uit de beoordeling van deze omgeving komt een gedetailleerde doelomschrijving voor de verbetering ervan naar voren.

Deze omschrijving is: *de syntax en semantiek van een remote procedure call zo veel mogelijk op die van een lokale procedure aanroep laten lijken.* Dit is in de Sun omgeving de syntax van de taal C. Het doel wordt aldus:

- Stel een lijst criteria op voor de beoordeling van gedistribueerde applicatieontwikkelomgevingen.

- Beoordeel de Sun omgeving; stel een lijst met sterke en zwakke punten op.
- Onderzoek hoe de Sun omgeving veranderd zou kunnen worden zodat *standaard C programma's* als input geaccepteerd worden.

Om tot deze gedetailleerde beschrijving van de gewenste verbeteringen te komen wordt een kader voor de beoordeling van applicatieontwikkelomgevingen aangelegd. Dit gebeurt in hoofdstuk 2 en 3.

1.4 Legitimering

Nadat het onderzoek op het gebied van gedistribueerde systemen zich voornamelijk op operating systems gericht heeft komt er steeds meer aandacht voor de ondersteuning van de ontwikkeling van gedistribueerde applicaties [Bal]. Het doel van dit onderzoek is het vereenvoudigen van applicatieontwikkeling voor een gedistribueerd computersysteem. Er worden criteria aangelegd voor de beoordeling van gedistribueerde applicatieontwikkelomgevingen in het algemeen, en voor één specifiek voorbeeld, de Sun omgeving, wordt geprobeerd het schrijven van toepassingsprogramma's te vereenvoudigen. Naar aanleiding van de beoordeling van deze veelgebruikte omgeving worden mogelijke verbeteringen voorgesteld en onderzocht. Birrell & Nelson schrijven in [Birrell, p. 41]:

> *The primary purpose of our RPC project was to make distributed computation easy. Previously, it was observed within our research community that the construction of communicating programs was a difficult task, undertaken only by members of a select group of communication experts. [...] Our hope is that by providing communication with almost as much ease as local procedure calls, people will be encouraged to build and experiment with distributed applications.*

Gezien het vele onderzoek naar gedistribueerde systemen en de verspreiding van deze systemen is, onderzoek naar het vereenvoudigen van het schrijven van gedistribueerde programma's de moeite waard.

1.5 Wijze van aanpak

Het doel van het onderzoek is het vereenvoudigen van applicatieontwikkeling voor een gedistribueerd computersysteem. De basis van dit onderzoek wordt gevormd door bestudering van literatuur over systeemontwikkeling, operating systems en gedistribueerde systemen. Eén van de belangrijkste concepten die hier naar voren is gekomen is 'abstractie'. Door abstractie kan de ingewikkeldheid (complexiteit) van gedistribueerde systemen tot een bruikbaar model worden teruggebracht. Dit wordt in § 2.2.1 en § 2.3.4 toegelicht.

Na de bestudering van de theorie is het onderzoeksobject, de Sun omgeving, bestudeerd. Op basis van een lijst criteria uit de literatuur zijn sterke en zwakke punten van het onderzoeksobject geïdentificeerd. Hierna zijn verbeteringen voor de zwakke plekken van Sun's RPC-omgeving voorgesteld.

In de probleemstelling wordt het vereenvoudigen van de Sun omgeving—het verhogen van de transparantie—centraal gesteld. Voor het verbeteren van de transparantie is een model ontworpen, waarvan een deel als prototype is geïmplementeerd. Ook het model is op basis van de literatuurcriteria beoordeeld. De resultaten van deze beoordeling zijn in hoofdstuk 6 beschreven. De onderzoeksbevindingen en aanbevelingen zijn in hoofdstuk 7 beschreven.

1.6 Opbouw scriptie

Het volgende hoofdstuk behandelt theorie over gedistribueerde computersystemen in het algemeen. Allereerst wordt de evolutie die tot gedistribueerde systemen geleid heeft geschetst. Hierna wordt een begrippenkader besproken. Daarna wordt ingegaan op de relatie tussen complexiteit en transparantie bij gedistribueerde systemen.

Hoofdstuk 3 behandelt applicatieontwikkeling voor gedistribueerde systemen. In dit hoofdstuk worden criteria geformuleerd waarmee men applicatieontwikkelomgevingen voor gedistribueerde systemen kan beoordelen. Het laatste deel van dit hoofdstuk gaat over remote procedure calls—één van de pijlers van gedistribueerde systemen. RPC's zijn de basis van het onderzoeksobject, de Sun omgeving.

De criteria worden in hoofdstuk 4 gebruikt om de Sun omgeving te beoordelen. Punten die voor verbetering in aanmerking komen worden hier besproken.

Hoofdstuk 5 beschrijft een voorstel voor de verbetering van een aantal zwakke punten, het model. Het prototype dat deze verbeteringen implementeert wordt beschreven in § 6.3 en appendix A. Ter illustratie van de transparantieverhoging zijn in de appendices twee versies van een voorbeeldprogramma opgenomen. Eén versie voor de ongewijzigde Sun omgeving, en één voor de verbeterde. Appendix A geeft een handleiding bij het prototype.

De criteria, waarmee de Sun omgeving is beoordeeld, worden in hoofdstuk 6 gebruikt om te kijken in hoeverre de doelstellingen zijn bereikt.

Tenslotte volgen de bevindingen en aanbevelingen van het onderzoek, en een samenvatting. Een lijst met de geraadpleegde literatuur completeert het geheel.

Hoofdstuk 2

Gedistribueerdheid

Het doel van dit hoofdstuk is een kader voor het onderwerp van deze scriptie te schetsen.

2.1 Ontstaansgeschiedenis

Deze paragraaf behandelt ontwikkelingen die tot het concept 'gedistribueerdheid' geleid hebben. Aan de ene kant is dat de historische ontwikkeling van de verdeling van rekencapaciteit. Hier wordt niet getracht een complete geschiedenis van de ontwikkeling van geautomatiseerde systemen te geven. Het doel is om een aantal ontwikkelingen uit het verleden in verband te brengen met de hedendaagse problematiek van gedistribueerde systemen. Aan de andere kant wordt in § 2.1.4 wordt deze ontwikkeling met verschillende categorieën toepassingen (applicaties) in verband gebracht.

De term 'gedistribueerd' zegt iets over de structuur van een computersysteem, over de wijze waarop delen in het geheel samenhangen. Het uitgangspunt van de historische ontwikkeling is de verdeling van rekenkracht over de gebruikers. De structuurveranderingen die in de loop der jaren hebben plaatsgevonden in de verschillende lagen waaruit een computersysteem is opgebouwd zijn op het niveau van het operating system duidelijk zichtbaar. In onderstaande tabel is het soort operating system in verband gebracht met de verdeling van de rekenkracht.

Operating systems

	gecentraliseerd	*gedecentraliseerd*
single user	batchverwerking	PC
multi user	multiprogramming	NOS/gedistribueerd

Hieronder wordt deze tabel uitgewerkt.

2.1.1 Gecentraliseerd

Sinds er computers zijn wordt er gezocht naar manieren om de verdeling van rekenkracht over de gebruikers zo efficiënt mogelijk te laten plaatsvinden. De dure computer moet liefst honderd procent van de tijd gebruikt worden en de rekencapaciteit moet voor gebruikers eenvoudig toegankelijk zijn, zonder toegangsbeperkingen of wachttijden.

single user

Rond 1955 kwamen de eerste computers die met transistors werkten. De programma's voor deze computers werden in assembleertaal of FORTRAN geschreven. De computers bestonden uit een enkele fysieke processor met ponskaarten of tapes als in- en uitvoermedium. Programma's werden in batches aan de computer aangeboden. Een programma had de machine voor de duur van de run helemaal voor zichzelf alleen.

Een nadeel van batchverwerking is de lange periode tussen aanbieding van de job en ontvangst van de uitvoer. Dit kon uren duren. Een ander nadeel is dat de processor relatief inefficiënt gebruikt werd. Bij een I/O operatie, bijvoorbeeld een gegevensbestand op een tape opzoeken, kon de processor niets anders doen dan wachten. Hierbij ging zeer kostbare tijd verloren.

multi user

Dit laatste probleem werd rond 1965 opgelost door het zogenaamde *multiprogramming*. Hierbij werd het geheugen van de computer in een aantal partities verdeeld. Eén voor elke job. Wanneer een job wachtte op een I/O operatie, kon een andere job de processor benutten. De complexiteit van het operating system nam toe.

Zo'n multiprogramming operating system was nog steeds batch georiënteerd. Men moest vaak op de output wachten. Een voorbeeld van een multiprogramming operating system is OS/360.

Door de roep om een snellere respons kwamen *time-sharing* systemen opzetten. Time-sharing is een variant van multiprogramming waarbij elke gebruiker een on-line terminal heeft. Bij time-sharing hoeven programma's niet in een batchwachtrij te staan. Korte opdrachten krijgen een snelle respons. Voorbeelden van zulke operating systems zijn VAX/VMS en UNIX.

Time-sharing en multiprogramming systemen bieden de gebruiker de abstractie dat men de computer voor zich alleen heeft. Een ontwerpdoel bij deze systemen is dat de gebruikers zo min mogelijk van de aanwezigheid van andere gebruikers mag merken. Het operating system moet de gebruiker een virtuele machine aanbieden. Eén zo'n time-sharing operating system heet 'Virtuele Machine': IBM's VM/360. De enige taak van dit operating system is om de onderliggende computer te repliceren in evenzovele virtuele 360's als er gebruikers zijn [De Bruin].

Time-sharing operating systems zijn ingewikkelder dan multiprogramming systemen. Wanneer de interne taken van het operating system—de kernel—zijn ontworpen als één proces (een zogenaamde monolitische monitor), zoals bij UNIX, neemt het relatief veel geheugen in beslag. In de jaren zeventig was zo'n ontwerp gebruikelijk. Door het toevoegen van functionaliteit is het operating system in de loop der tijd steeds omvangrijker worden. Door de omvang werd het moeilijk om onder alle omstandigheden een snelle respons te garanderen. Zo'n monolitische monitor is daarom in het algemeen niet geschikt voor real-time gebruik (procesbesturing).

Een reactie op de monolitische monitor zijn de zogenaamde micro-kernel operating systems. Deze zijn opgebouwd uit een kleine kern die alleen de scheduling (en eventueel virtueel geheugen beheer) verricht. De andere taken, geheugentoewijzing, I/O en bestandsbeheer worden als apart gebruikersproces behandeld. Met zo'n ontwerp worden complexiteit en geheugenbeslag van kernel en operating system teruggedrongen. Doordat de verschillende delen van het operating system in aparte processen draaien zijn gewone procedure aanroepen als bij monolitische monitors niet mogelijk. Een mogelijke oplossing is message-passing. Micro-kernel operating systems werken vaak via het client/server model: de kleine kernel regelt verzoeken tussen vragers en aanbieders van andere taken van het operating system [De Bruin]. Voorbeelden van micro-kernel message-passing operating systems zijn Mach [Coulouris] en MINIX [Tanenbaum 2].

2.1.2 Gedecentraliseerd

Batch, multiprogramming en time-sharing systemen gaan uit van één processor die zijn rekenkracht aan gebruikers aanbiedt. De rekenkracht staat centraal opgesteld. In het begin van de jaren zestig werden, naast multiprogramming, minicomputers ontwikkeld. De afdelingen werden zo voor hun rekenkracht minder afhankelijk van een centraal rekencentrum.

single user

Door de steeds verdergaande miniaturisering van elektronische componenten werden computers steeds kleiner qua omvang en prijs. Rond 1980 werden de eerste microcomputers ontwikkeld. De micro- of personal computer was zo goedkoop dat men een computer voor een enkel persoon kon aanschaffen. Met name de zogenaamde werkstations—krachtige personal computers—bieden veel verwerkingskracht voor één gebruiker. Door de lage prijs kwamen er steeds meer gebruikers. Deze gebruikers hadden behoefte aan een computer die eenvoudig bediend kon worden. Door de invoering van grafische gebruikersinterfaces met windows en muizen probeert men de gebruiksvriendelijkheid van computersystemen te vergroten. Een GUI (Graphical User Interface) is een programma dat het de gebruiker mogelijk maakt met behulp van pictogrammen en muisbewegingen met de computer te communiceren. Dit programma schermt de gebruiker af van de tot dan toe gebruikelijke operating system gebruikersinterface in de vorm van tekstcommando's. Een GUI is een schil om het operating system heen. GUI's vergen snelle beeldschermen en veel verwerkingskracht van de processor.

Personal computers (PC's) bieden decentrale verwerkingskracht. Ze zijn bedoeld voor één gebruiker. Het operating system kan in tegenstelling tot multiprogramming of time-sharing systemen eenvoudig gehouden worden.

Een voorbeeld van een simpel single user operating system is MS-DOS.

multi user

In veel organisaties zijn PC's tot de werkplek doorgedrongen. Om het delen van bestanden en randapparatuur als laserprinters mogelijk te maken worden ze vaak in een LAN (Local Area Network) gekoppeld. Zo'n LAN is vaak opgebouwd rond één of meer fileservers. Deze fileserver regelt het netwerkverkeer en bevat gemeenschappelijke bestanden. Het operating system van zo'n fileserver moet zaken als toegangscontrole, bestandsbeheer en printerbeheer regelen. Doordat het met meerdere gebruikers te maken heeft moet dit *network operating system* (NOS) met multi-user zaken als record- en filelocking rekening houden. Een voorbeeld van een network operating system is Novell Netware.

Met een LAN probeert men de voordelen van centrale en decentrale computersystemen te combineren. De gebruiker blijft de baas over de eigen PC[1] en kan toch van de faciliteiten van een groter geheel profiteren.

2.1.3 Gedistribueerd

Netwerk operating systems maken samenwerking tussen afzonderlijke single user operating systems mogelijk. Bij een NOS blijven de delen van het systeem als losse componenten zichtbaar. Een grotere mate van integratie wordt met gedistribueerde operating systems nagestreefd [Fortier]. Bij beide bestaat de hardware uit computers die in een LAN verbonden zijn. Bij NOS weet de gebruiker dat er verschillende computers zijn. Een gedistribueerd operating system daarentegen lijkt voor zijn gebruikers op een traditioneel gecentraliseerd-één-processor-time-sharing systeem. Een gedistribueerd operating system biedt de gebruiker de abstractie van een enkelvoudige machine. Details als welke processor een programma uitvoert en waar bestanden zich bevinden behoren transparant te zijn voor de gebruiker.

Het voordeel van gedistribueerde operating systems is de mogelijkheid van parallellisme en fouttolerantie (§ 2.1.4 en § 2.3.3). Een nadeel voor ontwerpers is de grote complexiteit van ervan. Gedistribueerde operating systems zijn ingewikkelder dan andere operating systems. Voorbeelden van operating systems die in meer of mindere mate het predikaat 'gedistribueerd' verdienen zijn Sun's NFS, Mach en Amoeba. Sun NFS bestaat uit bestandsbeheeruitbreidingen van een monolitische UNIX kernel. Mach en Amoeba zijn beide micro-kernel operating systems. De Mach kernel emuleert een UNIX kernel [Coulouris].

2.1.4 Redenen gedistribueerde applicaties

Er zijn verschillende redenen waarom men applicaties op gedistribueerde computersystemen, en niet op uniprocessors, zou willen implementeren. In

[1] Het feit dat men de processor niet met anderen hoeft te delen is in [Coulouris] als volgt onder woorden gebracht: „The nicest thing about workstations is that they don't run faster at night."

[Bal] worden de volgende vier genoemd.

1. Een kortere doorlooptijd voor een enkele berekening.
2. Toegenomen betrouwbaarheid en beschikbaarheid.
3. De mogelijkheid bepaalde delen van het systeem te gebruiken om specifieke functionaliteit efficiënt aan te kunnen bieden.
4. De mogelijkheid inherente gedistribueerdheid van een applicatie te benutten.

De redenen worden hieronder een voor een toegelicht.

snelheidswinst door parallelle uitvoering (1)

Snelheidswinst door parallellisme is een veel voorkomende reden om applicaties op een gedistribueerd systeem uit te voeren. Door verschillende delen van een programma tegelijk uit te voeren op verschillende processoren zullen sommige programma's sneller klaar zijn. Dit soort applicaties kunnen in principe net zo goed op shared-memory multiprocessors worden uitgevoerd. Een probleem van shared-memory systemen is dat ze moeilijk te schalen zijn naar grote aantallen (duizenden) processoren. Gedistribueerde systemen zijn beter schaalbaar, al is de communicatie via shared-memory sneller dan via een netwerk.

Parallelle applicaties kunnen geclassificeerd worden naar de korrelgrootte (*grain*) van hun parallellisme. Deze korrelgrootte is de lengte van de rekentijd voordat er weer gecommuniceerd moet worden. Large-grain parallelle programma's brengen het grootste gedeelte van hun tijd door met rekenen en communiceren weinig; fine-grain parallelle programma's communiceren vaker; medium-grain zit er tussenin (zie ook § 2.3.1).

fouttolerante applicaties (2)

Voor applicaties die essentieel zijn voor de bedrijfsvoering van een organisatie—zogenaamde 'mission critical applications' zoals de besturing van een olieraffinaderij, een vliegtuig, of de administratie van een bank—is een uniprocessor wellicht niet betrouwbaar genoeg. Vanwege de *partial failure* eigenschap zijn gedistribueerde computersystemen potentieel betrouwbaarder. Omdat de fysieke processoren autonoom zijn blijven bij een fout in een processor de andere processoren werken. Door nu procedures en gegevens van de applicatie op meerdere processoren te dupliceren kunnen, wanneer een processor stopt, andere doorgaan met het programma, of de gegevensintegriteit waarborgen.

Men zou sommige fouttolerante applicaties wellicht ook op shared-memory multiprocessors kunnen implementeren. Ook op deze systemen kan een applicatie uitvallen van delen van het systeem overleven. Echter, doordat ze niet geografisch verspreid kunnen worden opgesteld zoals gedistribueerde systemen, zijn ze niet in staat rampen als brand en aardbevingen te doorstaan. Voor een bedrijfszekere bankapplicatie zou een loosely-coupled distributed system een verstandige keuze zijn.

Het onderzoek op dit gebied richt zich voornamelijk op software technieken om betrouwbaarheid en beschikbaarheid te vergroten. In § 3.3.3 wordt hier dieper op ingegaan.

specifieke functionaliteit (3)

De functionaliteit die computersystemen moeten bieden kan zeer divers zijn. In plaats van alle verschillende eisen door één algemeen soort computerconfiguratie te laten uitvoeren kan men ook voor verschillende eisen hardware inzetten die speciaal op deze eisen is toegesneden. In § 2.1.3 werd reeds over fileservers gesproken. Personal computers of werkstations zijn geschikt voor GUI's. Voor het uitvoeren van veel numerieke berekeningen kan men een supercomputer (number cruncher) inzetten. Databasemachines zijn geschikt voor het efficiënt uitvoeren van transacties.

In [Bal] wordt beschreven hoe bij de implementatie van het gedistribueerde operating system Amoeba de keuze voor gespecialiseerde file-, print-, proces-, terminal-, tijd-, boot- en gatewayservers op gedistribueerde hardware als van nature uit de aard der problematiek voortsproot. Elke service kan één of meer gespecialiseerde processoren gebruiken. De servers zenden elkaar boodschappen via het netwerk. Wanneer het systeem met nieuwe functionaliteit wordt uitgebreid kunnen eenvoudigweg nieuwe processoren worden toegevoegd.

inherent gedistribueerde applicaties (4)

Sommige applicaties zijn gedistribueerd van aard. Het versturen van e-mail tussen de werkstations van gebruikers is hier een voorbeeld van. De verzameling werkstations kan als een gedistribueerd systeem beschouwd worden.

Een organisatie met verscheidene filialen en fabrieken zou een gedistribueerd systeem kunnen opzetten om mensen en machines op verschillende plaatsen met elkaar te kunnen laten communiceren.

Toepassingen van Electronic Data Interchange (EDI) zijn ook voorbeelden van applicaties waarvan verschillende delen zich op geografisch verschillende lokaties afspelen.

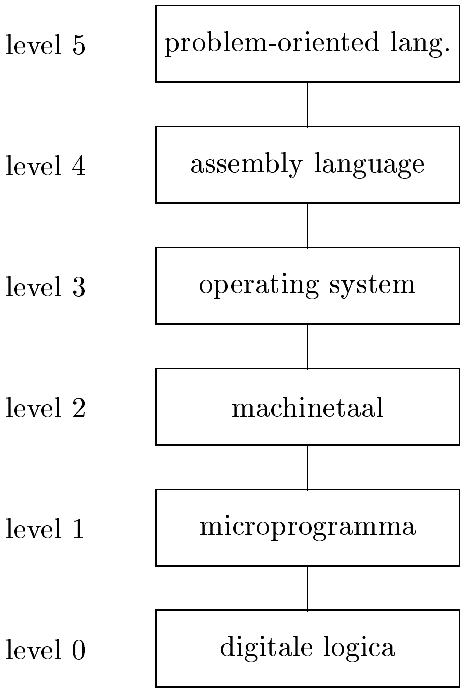
2.2 Begrippenkader

2.2.1 Virtuele machines

Computersystemen zijn complexe systemen. Om het ontwerpen van computer en programma hanteerbaar te houden wordt gebruik gemaakt van het concept van de *gelaagde* machine. Figuur 2.1 geeft een mogelijke indeling van een computer in een aantal lagen.

Computerhardware is ongeschikt om hoog niveau problemen als het oplossen van een differentiaalvergelijking in te programmeren. Het lagenmodel beschrijft een stapsgewijze vertaling van gebruikerstoepassing naar hardware-niveau. Elke laag beschrijft een interface van een virtuele machine die een 'taal' accepteert. Het laagste niveau betreft de hardware, de elektronische schakelingen van de computer. In de taal van het hoogste niveau worden de problemen die het systeem moet oplossen gespecificeerd.

Figuur 2.1: Lagenmodel [Tanenbaum 1]



De vertaler tussen deze lagen kan men beschouwen als een *virtuele machine* die de taal implementeert. Een virtuele machine is een schijnbare machine. De echte machine doet zich voor als een andere, aangenamere, machine. Elke virtuele machine heeft zijn eigen taal. Hij biedt op basis van de simpelere, meer machine georiënteerde taal van zijn voorganger een taal aan zijn opvolger aan die meer mogelijkheden biedt, meer probleem georiënteerd is [Tanenbaum 1, p. 5].

Een centraal begrip bij het ontwerp van gedistribueerde systemen is *complexiteit*—in de zin van ingewikkeldheid. De complexiteit moet worden teruggebracht onder de voorwaarde dat de voordelen van gedistribueerdheid—snelheid en bedrijfszekerheid—behouden blijven. Het ordenen van complexiteit is *structureren*. Door structuren in een afbeelding van de werkelijkheid aan te brengen—door een model te bouwen—wordt het systeem beter te begrijpen. De complexiteit zoals de beschouwer deze ervaart wordt verminderd.

Het lagenmodel beschrijft een stapsgewijze reductie van de complexiteit. Elke laag reduceert de complexiteit van zijn voorganger.

abstractie

Bij het redeneren over *gedrag* van computersystemen levert de complexiteit een vergelijkbaar probleem op. Het heeft geen zin om de problematiek van het vinden van de kortste route langs de provinciehoofdsteden in termen van NAND en OR poorten te bespreken. Bij het denken over een probleem moet men zoveel mogelijk van bijzaken *abstraheren*. Men moet zich op de kern van de problematiek concentreren. [Watt 1] zegt het zo:

> ***Abstraction*** *is a mode of thought by which we concentrate on general ideas rather than on specific manifestations of these ideas.*

Door abstractie is men in staat te generaliseren. In de informatica is het concept 'abstractie' heel goed bruikbaar.

Een virtuele machine is een abstractie. Door een probleem in de taal van een hogere laag te formuleren abstraheert men van de details van de lagere machine. Het is eenvoudiger om het opvragen van een banksaldo in een SQL query te formuleren dan in COBOL [Date].

Bij systeemanalyse is abstractie de kunst van het zich concentreren op de essentiële aspecten van de praktijksituatie. Door het bestuderen van richtlijnen en procedures moet men voldoende inzicht in de problematiek verwerven om een volledige en ondubbelzinnige systeemspecificatie te kunnen opstellen [DeMarco].

Het doel van het schrijven van een computerprogramma is dat de gebruiker uiteindelijk kan abstraheren van *hoe* het programma werkt en zich kan concentreren op *wat* het doet. Abstractie is hier het verschil tussen de interfacebeschrijving en de implementatie van een probleem. Een programmeertaal bestaat uit constructies die een abstractie zijn van machinetaal. Elke procedure of functie is een abstractie. Elke module is een abstractie. Een programma is een hiërarchie van abstracties [Watt 1].

imperfectie

In verhandelingen over computersystemen wordt vaak getracht consequent op één abstract niveau te blijven redeneren. Gegeven het model van figuur 2.1 moet bij het redeneren over computersystemen consequent worden aangegeven van welk model men uitgaat. Soms blijkt uit de context over welke laag het gaat, vaak ook niet. Door waar nodig het niveau expliciet te noemen voorkomt men dat het zorgvuldig opgebouwde begrippenkader vertroebeld raakt door verwarring met termen van een andere laag.

Vaak redeneert men op het niveau van een probleemgeoriënteerde taal. Hierbij kunnen zich een aantal complicaties voordoen. Virtuele machines kunnen in het algemeen de onderliggende lagen niet perfect afschermen. Hierdoor loopt men het risico dat de verhandeling wordt vertroebeld door details van lagere niveaus. Veel implementaties van probleemgeoriënteerde talen bijvoorbeeld schermen de woordbreedte van de machinetaal-virtuele-machine niet goed af. Overflowproblemen die hier het gevolg van zijn kunnen de helderheid van een algoritme behoorlijk frustreren.

Ter verduidelijking van een verhandeling worden vaak voorbeelden aangehaald, of metaforen uit het dagelijks leven gebruikt. Bepaalde aspecten van een virtuele machine kunnen zo aanschouwelijk gemaakt worden. Voorbeelden en metaforen kunnen in het algemeen maar een deel van de virtuele machine beschrijven. Het kan gebeuren dat er hierdoor onbedoeld verkeerde implicaties in de verhandeling sluipen [Leipoldt].

Door gebruik te maken van abstracties kan men over complexe computersystemen redeneren. Computers zijn opgebouwd uit virtuele machines. Een virtuele machine is een abstractie. Bij het redeneren op logisch niveau moet men zich steeds realiseren dat men gebruik maakt van een abstractie van de werkelijkheid. Een virtuele machine is in werkelijkheid vaak imperfect. Om fouten of afwijkingen te voorkomen kan men formele methoden gebruiken om gedrag te beschrijven. Hiervoor moet men het gedrag van een systeem *exact* kunnen beschrijven. Soms is de kennis over het exacte gedrag niet aanwezig of zijn de gebruikte methoden niet toereikend.

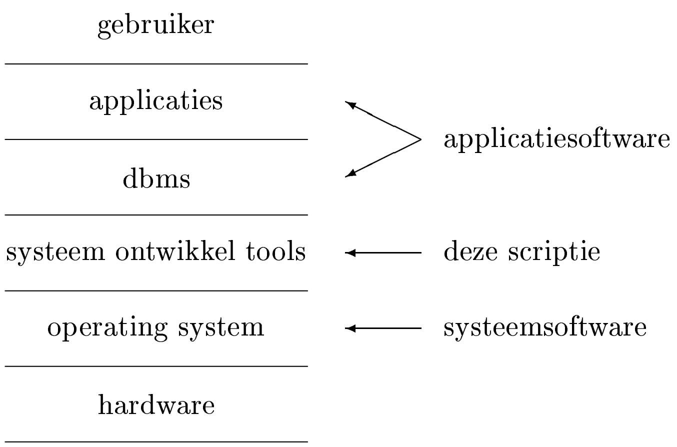
Formele bewijzen zijn geldig voor algoritmes. De implementatie van deze algoritmes kan imperfect zijn. Naast een formeel bewijs van de juistheid van een algoritme moet de juistheid van de implementatie met testen worden vastgesteld [Van Katwijk, Mullender 1].

2.2.2 Gerelateerde termen

Voor de lagen uit figuur 2.1 worden nogal eens andere termen gebruikt. Deze zijn in figuur 2.2 weergegeven. Het meest algemene onderscheid is tussen hardware en software. Binnen de software onderscheidt men systeemsoftware en applicaties. Operating systems behoren tot de systeemsoftware, compilers tot de systeem*ontwikkel*software. Voorbeelden van applicaties zijn een boekhoudpakket of een tekstverwerker. Ze worden met behulp van systeemontwikkelsoftware ontwikkeld.

Problem oriented languages (level 5), worden ook 'derde generatie talen' of 'high-level languages' genoemd. Tot en met level 5 is figuur 2.1 goed toepasbaar als model voor veel bestaande machines. Boven level 5 bestaat een grotere variëteit. Om deze reden is niet getracht een geforceerd simpele weergave van de werkelijkheid te geven [Tanenbaum 1]. Volgens de indeling van figuur 2.1 zouden applicaties 'level 6 virtuele machines' genoemd moeten

Figuur 2.2: Applicatie- versus systeemsoftware [Van Renesse]



worden. In grote lijnen zou men applicaties kunnen beschouwen als een laag die tussen de problem-oriented-language en de gebruiker in ligt.

Gegevensbeheerpakketten of vierde generatie-software als Oracle en Ingres worden soms tot applicaties en soms tot systeemsoftware gerekend. Deze scriptie richt zich op de systeemontwikkeling van gedistribueerde applicaties. Gegevensbeheerpakketten liggen gezien vanuit systeemontwikkeling als in figuur 2.2 gepositioneerd aan de applicatie kant.

implementatie

Met het 'implementeren' van een applicatie bedoelt men het realiseren van een interface met de gewenste high level functionaliteit met behulp van de onderliggende virtuele machine. De taal van een hogere laag wordt uitgedrukt in een lagere laag. Zo kan men een compiler voor het vertalen van een problem-oriented-language (bijvoorbeeld FORTRAN) naar assembleertaal schrijven. Of men kan een boekhoudpakket bouwen in een problem-oriented-language. Men spreekt soms van de FORTRAN-implementatie als men een specifieke vertaler bedoeld, of van de implementatie van de boekhouding als men het programma zelf bedoelt.

Een implementatie biedt een hogere taal aan door deze te vertalen in een lagere taal. Een implementatie is een virtuele machine. Implementeren is het realiseren van de gewenste functionaliteit—van de vertaalslag tussen twee lagen.

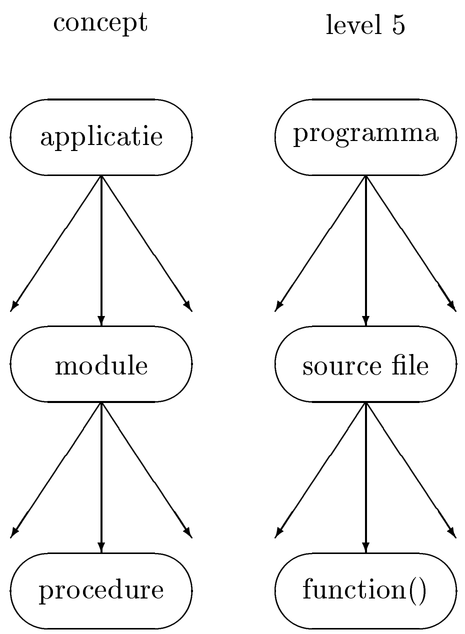
De term 'processor' wordt gebruikt om de abstracte automaat die een bepaalde taal implementeert aan te duiden. Een processor is het actieve gedeelte van een virtuele machine. De automaat voert het programma dat in zijn taal is geschreven daadwerkelijk uit.

Het aanbieden van de gewenste functionaliteit—van een concept, een abstract idee—wordt ook 'implementeren' genoemd. Zo kan een operating system (level 3) bijvoorbeeld *de procedureabstractie gegeneraliseerd naar gedistribueerde omgevingen* implementeren. Zo'n operating system biedt een 'RPC implementatie' aan aan de hogere lagen.

modules

Vanwege de complexiteit van toepassingen worden applicatieprogramma's in modules opgebouwd. Er bestaan equivalente namen voor de verschillende modulariseringsniveaus op de verschillende virtuele machines. Het concept

Figuur 2.3: Programma's, modules en procedures



'applicatie' wordt op level 5 uit figuur 2.1 een 'programma' genoemd. Wanneer de programmeur de source code (C of COBOL bijvoorbeeld) geschikt gemaakt heeft voor uitvoering door de level 2 processor wordt het een executable file of een executable image genoemd.

Applicaties zijn opgebouwd uit één of meer modules. Het concept 'module' wordt in deze scriptie geacht equivalent te zijn met een source file. Het equivalent van een source file op machine taal niveau heet object file. In deze scriptie geldt een één op één relatie tussen module en source/object file.[2]

Modules zijn opgebouwd uit één of meer procedures. De term voor procedures verschilt tussen de verschillende high level languages. Ze worden ook wel functies of routines of subroutines genoemd. In figuur 2.3 wordt op verschillende manieren van groot naar klein gegaan. De figuur kent drie

[2]Dit wijkt af van [Page-Jones], waar een equivalentie tussen module en *procedure* geldt.

dimensies. Ten eerste: van links naar rechts wordt van hoge naar lage abstractieniveaus afgedaald. Ten tweede: van boven naar beneden wordt het concept 'applicatie' in steeds kleinere entiteiten opgesplitst. Ten derde: elke applicatie bestaat uit *een aantal* modules, en elke module bestaat weer uit *een aantal* procedureabstracties.

2.3 Gedistribueerde computersystemen

In deze paragraaf wordt beschreven welke systemen gedistribueerd zijn en welke niet. Er wordt een onderscheid gemaakt naar hardware, software en operating system aspecten van gedistribueerdheid.

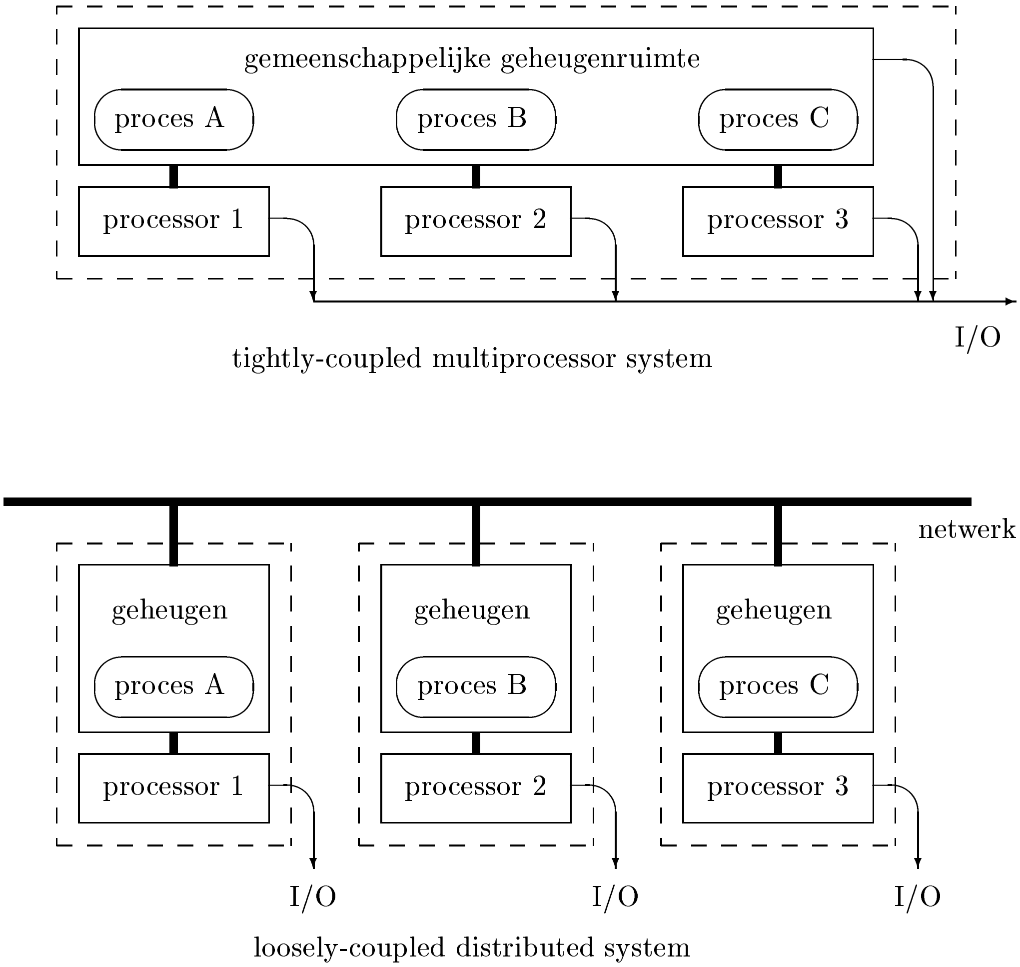
2.3.1 Typologie hardware

Hier wordt een overzicht gegeven van de verschillende vormen van distributie die ontwikkeld zijn.

[Coulouris] geeft een typologie van systemen met meer dan één processor op basis van de mate van *fysieke* samenhang tussen processor en geheugen. De systemen bestaan uit meerdere processoren. Ze kunnen alle gebruikt worden om programma's parallel uit te voeren. In een tightly-coupled systeem hebben de processoren fysiek gemeenschappelijk geheugen. In een loosely-coupled systeem hebben ze elk hun eigen geheugen. Zie figuur 2.4.

- In een *loosely-coupled distributed system* hebben de verschillende processoren elk hun eigen geheugen. Ze zijn via een communicatie netwerk met elkaar verbonden. Hierbinnen worden twee subcategorieën onderscheiden [Bal]:
  - Computers die via een LAN met elkaar verbonden zijn. Elke computer vormt een werkplek voor de gebruiker. De processoren/computers kunnen geografisch verspreid staan opgesteld. Dit model wordt het 'werkstation/server model' genoemd. Veel commercieel verkrijgbare systemen, zoals de Sun systemen, hebben deze structuur.
  - Naast het werkstation/server model is er het *multicomputer* model. Dit zijn systemen die zijn opgebouwd uit processoren met eigen geheugen die via message-passing communiceren. De deelcomputers missen—afgezien van de interne communicatie—een eigen I/O systeem. De I/O van het geheel wordt door een aparte hoofdcomputer verzorgt. Transputers zijn een voorbeeld van deze categorie.
- Bij *tightly-coupled multiprocessor systems* delen de processoren fysiek een gezamenlijk geheugen. Dit heeft als voordeel dat communicatie tussen de delen van het gedistribueerde systeem snel plaats vindt. Deze systemen zijn geschikt voor het gelijktijdig uitvoeren van verschillende deeltaken van een programma. Een nadeel is dat geografische spreiding van de verschillende processen niet mogelijk is.

Figuur 2.4: Wel of niet gedistribueerd [Coulouris]



Gedistribueerde computersystemen zijn loosely-coupled systemen. Het criterium is of de processoren een gemeenschappelijk geheugen delen. Computers volgens het werkstation/server model en multicomputers vallen hieronder. Definitie [Bal, p. 3]:

> *Een gedistribueerd computersysteem bestaat uit meerdere autonome [hardware] processoren die geen primair geheugen delen, maar samenwerken door boodschappen over een communicatienetwerk te zenden.*

Communicatie via een netwerk is ongeveer een factor 10^4 trager dan communicatie via gemeenschappelijk geheugen (zie pagina 46). Applicaties met modules die frequent communiceren kunnen beter op tightly-coupled systemen geïmplementeerd worden.

Het begrip 'grain' (§ 2.1.4) beschrijft het soort applicatie, het begrip 'coupling' slaat op de architectuur waarop deze applicaties worden uitgevoerd. Fine-grain en medium-grain parallellisme zijn eerder geschikt voor implementatie op tightly-coupled systemen omdat de communicatie-overhead op loosely-coupled systemen de uitvoering teveel zou ophouden. Large-grain parallellisme kan zowel op tightly als op loosely-coupled systemen geïmplementeerd worden [Bal].

2.3.2 Fysiek/logisch

Voorgaande definitie en classificatie zijn gebaseerd op de fysieke machine—de hardware. De meeste gebruikers zullen bij het werken met computersystemen met een *virtuele* machine te maken hebben. Een onderscheid op het niveau van virtuele machines is daarom zeker zinvol.

Volgens de definitie communiceren gedistribueerde systemen via message-passing. Omdat ze zijn opgebouwd uit meerdere processoren, kunnen programma's parallel uitgevoerd worden. Het communicatiemodel zal voor sommige toepassingen een efficiënte implementatie mogelijk maken, voor andere problemen is het te ingewikkeld. Andere communicatieparadigma's kunnen door softwarelagen (virtuele machines) geïmplementeerd worden.

Het is zinvol om een onderscheidt tussen *fysieke* en *logische* distributie te maken. Definitie [Bal, p. 10]:

> *Een* ***logisch*** *gedistribueerd software systeem bestaat uit meerdere software processen die via explicitie message-passing communiceren.*

Een logisch gedistribueerd software systeem biedt zijn gebruiker een gedistribueerd beeld van de machine. Het softwaresysteem kan een ander model dan de hardware bieden.

Deze definitie en de voorgaande maken de volgende vier combinaties van logische en fysieke distributie mogelijk:

1. logisch gedistribueerde software die wordt uitgevoerd op fysiek gedistribueerde hardware
2. logisch gedistribueerde software die wordt uitgevoerd op fysiek niet-gedistribueerde hardware
3. logisch niet-gedistribueerde software die wordt uitgevoerd op fysiek gedistribueerde hardware
4. logisch niet-gedistribueerde software die wordt uitgevoerd op fysiek niet-gedistribueerde hardware

In het eerste geval bestaat het systeem uit een verzameling processen die op een aparte fysieke processor uitgevoerd worden en via `send` en `receive` primitieven communiceren over een netwerk. Een e-mail systeem kan zo geïmplementeerd zijn.

In het tweede geval heeft het systeem dezelfde logische multi-proces structuur. Nu wordt de fysieke message-passing met shared-memory (gemeenschappelijke adresruimte) gesimuleerd. De interproces-communicatie in een MINIX systeem is hier een voorbeeld van, zij het dat dit een pseudo-parallel systeem voor één fysieke processor is.

In de derde klasse probeert de virtuele machine de abstractie van een gemeenschappelijk-geheugen-multiprocessor te bieden aan de applicatieprogrammeur. De fysieke verdeling van het geheugen wordt onzichtbaar gemaakt. Gemeenschappelijk virtueel geheugen (§ 3.3.2) is hier een voorbeeld van.

In het vierde geval wordt óók de functionaliteit van gemeenschappelijk geheugen geboden. Nu is de implementatie van deze virtuele machine triviaal geworden door de aanwezigheid van fysiek shared-memory.

Een gedistribueerd operating system dat transparant voor zijn gebruiker is, is fysiek gedistribueerd en logisch *niet* gedistribueerd. Een gedistribueerd systeem probeert de fysieke gedistribueerdheid zoveel mogelijk te camoufleren met vertaallagen die het gebruik van niet-gedistribueerde software mogelijk maken.

Vanwege deze transparantie is een gedistribueerd operating system een logisch *niet* gedistribueerd systeem gebouwd op een fysiek wel gedistribueerd systeem. Een gedistribueerde compiler is een compiler voor een fysiek gedistribueerde architectuur. Het is niet nodig dat de taal die deze compiler aanbiedt logisch gescheiden geheugen implementeert. Voor de term 'gedistribueerd systeem', of 'gedistribueerde compiler', is *fysieke* gedistribueerdheid het criterium.

2.3.3 Operating system

De definities van gedistribueerde systemen in de literatuur richten zich op verschillende virtuele machines. In § 2.3.1 is als criterium beschreven dat gedistribueerde systemen bestaan uit meerdere processoren met elk een apart geheugen. Naast dit hardwarecriterium zijn beschrijvingen van een hoger niveau noodzakelijk om eisen voor de hogere virtuele machines te kunnen vaststellen. Deze beschrijvingen stellen eisen aan de vertaalslag die verschillende lagen van virtuele machines van de hardware naar de gebruiker moeten implementeren.

gedistribueerd systeem

In [Mullender 1, p. 5] wordt de volgende omschrijving van gedistribueerde systemen gegeven:

> *The fundamental properties of a distributed system are* ***fault tolerance*** *and the possibility to use* ***parallelism****.*
> *[...]*
> *A distributed operating system is one that looks to its users like an ordinary centralized operating system, but runs on multiple, independent CPU's. The key concept here is transparency, in*

> *other words, the use of multiple processors should be invisible (transparent) to the user. Another way of expressing the same idea is to say that the user views the system as a 'virtual uniprocessor', not as a collection of distinct machines.*

Doordat een gedistribueerd systeem is opgebouwd uit meerdere computers is het mogelijk processen gelijktijdig uit te voeren—parallellisme. Het is ook mogelijk gegevens gelijktijdig op meerdere plaatsen op te slaan. Hierdoor kunnen beschikbaarheid en betrouwbaarheid van gegevens vergroot worden. Deze mogelijkheden passen achtereenvolgens bij applicatiesoort 1 en 2 uit § 2.1.4.

In de laatste alinea wordt beschreven hoe de 'hoogste' virtuele machine door de gebruiker gezien wordt. De lagere virtuele machines (meerdere onafhankelijke processoren) worden aan de hoogste virtuele machine (virtuele uniprocessor) gerelateerd.

gedistribueerd operating system (level 3)

Ook voor de karakterisering van gedistribueerde *operating* systems (level 3 virtuele machines) is transparantie een sleutelbegrip. In [Coulouris, p. 9] worden een aantal kenmerken opgesomd:

- Uitbreidbaarheid (schaalbaarheid)
- Verbeterde beschikbaarheid
- Beter gebruik van hulpmiddelen
- Meerdere gelijkvormige componenten
- Onderling verbonden verwerkingseenheden
- Transparantie
- Afwezigheid van hiërarchische beheersstructuren
- Processen met gescheiden adresruimten die via expliciete message-passing communiceren

Transparantie wordt in deze context verder uitgewerkt. Men definieert transparantie als:

> *het verhullen van de delen voor de gebruiker en applicatieprogrammeur, zodat het systeem gezien wordt als een geheel in plaats van een verzameling onafhankelijke componenten.*

De volgende acht vormen van transparantie worden genoemd:

- *Access transparency* zorgt ervoor dat lokale bestanden en andere objecten met dezelfde operaties benaderd kunnen worden als bestanden op een andere machine.

- *Location transparency* zorgt ervoor dat objecten benaderd kunnen worden zonder dat men hun plaats hoeft te kennen.
- *Concurrency transparency* stelt een aantal gebruikers of applicaties in staat gelijktijdig operaties op gemeenschappelijke gegevens te kunnen uitvoeren zonder dat er conflicten optreden.
- *Replication transparency* staat meervoudige instanties van bestanden toe en maakt het mogelijk andere bestanden te gebruiken teneinde de betrouwbaarheid te verhogen zonder dat gebruikers of applicaties hier kennis van hoeven te dragen.
- *Failure transparency* verhult falen van hardware of software componenten zodat gebruikers en applicaties hun taken ongestoord kunnen volbrengen.
- *Migration transparency* staat het verplaatsen van objecten binnen een systeem toe zonder dat de operaties van gebruikers of applicaties hier hinder van ondervinden.
- *Performance transparency* maakt het mogelijk het systeem te herconfigureren om de prestatie te verbeteren bij wisselende belasting.
- *Scaling transparency* maakt het mogelijk schaalvergrotingen van systeem en applicaties door te voeren zonder de systeemstructuur of applicatiealgoritmen te wijzigen.

De verscheidenheid aan vormen van transparantie illustreert de grote gevolgen die het gebruik van meerdere computers op het ontwerp van een operating system heeft. De gevolgen van de scheiding van processoren komen tot uiting in de noodzaak tot communicatie en expliciete beheers- en integratietechnieken. Voordelen van scheiding zijn de mogelijkheid van parallelle uitvoering van programma's en ongevoeligheid voor fouten in delen van het systeem.

Bij de transparantie-eisen wordt er van uitgegaan dat de verschillende gedistribueerde elementen op het niveau van het operating system transparant zijn. Level 4, level 5 en de applicatie of eindgebruiker zien een operating system dat een virtuele uniprocessorinterface aanbiedt. Deze transparantie-eisen zijn als doel van een ideaal systeem geformuleerd. De realiteit is dat operating systems niet volledig transparant zijn (§ 2.3.5).

tegenstelling NOS en GOS

In [Fortier] wordt het begrip 'transparantie' gebruikt om een onderverdeling tussen wel of niet gedistribueerde operating systems te kunnen maken. Hier wordt weer alleen de level 3 virtuele machine omschreven. Een gedistribueerd operating system wordt door zijn gebruikers als volgt gezien [Fortier, p. 32]:

> *The users have a uniform and transparent view of resources, and global resources are fairly allocated to independent processes.*

Naast gedistribueerde operating systems zijn er ook netwerk operating systems. Dat zijn operating systems die wel de architectuur van § 2.3.1 kennen, maar niet aan de eis 'transparant voor level 4' voldoen.

Een *netwerk operating system* biedt een verzameling autonome diensten aan. De processen en hulpmiddelen (*resources*) zijn alleen lokaal bekend. Wanneer een gebruiker een proces wil laten uitvoeren moet deze van het bestaan van een netwerk afweten en expliciet een verzoek aan een ander station richten. De gebruiker moet zelf de namen van bepaalde stations kennen.

Volgens [Fortier] kent een *gedistribueerd operating system* een transparantiemechanisme dat het netwerk en zijn details voor de gebruikers verbergt. Het vormt één logisch geheel waarin processen en hulpmiddelen globaal bekend zijn. De gebruiker kent geen stationnamen, want stations zijn voor de gebruiker geen herkenbare entiteiten.

Een gedistribueerd operating system is meer dan som van de (hardware) delen: het is transparant. Het biedt een grotere bedrijfszekerheid door replicatie en een grotere verwerkingssnelheid door parallellisme [Mullender 1]. De waarde van een netwerk operating system is *niet* groter dan de som van de samenstellende delen.[3]

Primitieven (*system-calls*) die door operating systems worden aangeboden die volledig transparant zijn worden *Gedistribueerd Operating System primitieven* (GOS-primitieven) genoemd. Primitieven waarbij niet alle details voor de gebruiker worden afgeschermd worden *Netwerk Operating System primitieven* (NOS-primitieven) genoemd.

Zowel gedistribueerde- als netwerk operating systems zijn geïmplementeerd op gedistribueerde hardware. Een gedistribueerd operating system biedt daarbij transparantie voor de hogere lagen. Wat betreft de interne opbouw van de virtuele machine geldt dat er in principe geen hiërachische structuur in is aangebracht. [Fortier, p. 100] zegt dat de technologie nog niet in staat is een volledig gedistribueerde omgeving te verwerkelijken zonder een gecentraliseerde beheerscomponent.

Bestaande operating systems die 'gedistribueerd' genoemd worden zijn een mix van netwerk en gedistribueerd operating system.

[3] Een gebruiker kan in een netwerk operating system processen expliciet parallel schedulen. In een gedistribueerd operating system gebeurt dit impliciet—door de virtuele machine zelf.

2.3.4 Beheersen van complexiteit

De reden waarom transparantie zo belangrijk gevonden wordt door veel auteurs,[4] is dat de complexiteit (ingewikkeldheid) van computersystemen bestudering van hun kenmerken *en detail* ondoenlijk maakt. Zonder een model van de werkelijkheid is onderzoek niet mogelijk (zie ook § 2.2.1). Figuur 2.1 schetst een model dat is opgebouwd uit lagen. Elke laag beschrijft een model van de werkelijkheid. Elke laag is een abstractie, een schijnbare werkelijkheid, een virtuele machine.

Gedistribueerde systemen zijn opgebouwd uit meerdere conventionele computersystemen. Hierdoor is hun complexiteit groter dan van conventionele computersystemen. Het ontwerp van gedistribueerde systemen is zo moeilijk omdat de complexiteit het bevattingsvermogen te boven gaat. De bron van deze buitengewone complexiteit is dat de *koppeling* van componenten die men op zich goed begrijpt onvermoede problemen kan veroorzaken.

[Mullender 1, p. 9] geeft hiervan het volgende voorbeeld. Bij een tokenring netwerk kan een station een pakket versturen zodra hij het token heeft. Daarna krijgt het volgende station het token om eventueel een pakket te versturen. De kern van het token-ring protocol is dat een station bij het versturen van een grote multi-pakket boodschap een ander station zo nooit kan buitensluiten.

Dit gebeurde toch. De ontvangstcircuits van de token-ring interfacekaarten bleken een pakket dat direct een ander pakket volgt niet op te merken. Het volgende gebeurde: (1) A zendt het eerste pakket van een boodschap naar O. (2) B zendt direct na A een pakket naar O, maar wordt genegeerd omdat het te dicht op het vorige pakket volgt. (3) A zendt het tweede pakket van de boodschap direct na B's pakket. O is weer hersteld, en ontvangt het pakket. (4) B verzendt het eerste pakket nogmaals, en wordt weer genegeerd. Dit gaat zo door, met als resultaat dat B geblokkeerd is juist *omdat* het token-ring protocol garandeert dat A en B om beurten mogen zenden.

Het token-ring protocol werkte op zich goed, de ontvangstcircuits ook. Maar de combinatie van beide zorgde voor een resultaat dat niet was voorzien door de ontwerpers.

Gedistribueerde systemen zijn zo ingewikkeld omdat de functionele eisen zo complex zijn.[5] Gebruikelijke taken die elk computersysteem moet verrichten worden door de gedistribueerdheid ingewikkelder. [Mullender 1] geeft als voorbeeld een file systeem. Een *single user* file systeem moet een zekere functionaliteit bieden. Men moet files kunnen maken, een naam geven, lezen, wijzigen en verwijderen. Een *time-sharing* systeem heeft meerdere gebruikers. Het systeem moet nu rekening houden met zaken als gebruikersidentificatie, toegangscontrole, lees/schrijf concurrency etc. Een *gedistribu-*

[4] Zie [Watt 1, Tanenbaum 1, Tanenbaum 5, Coulouris, Fortier, Mullender 1, Page-Jones, Date, De Bruin, Birrell, Bal, Weihl 1].

[5] Naast functionele redenen zijn er economische redenen. Zelfs wanneer dit fysiek mogelijk zou zijn is het te duur om elke computer waar ook ter wereld direct met elke andere computer te verbinden. Wide area netwerken hebben hierom een vermaasde structuur [Matthijsen]. Netwerksoftware wordt hierdoor complexer. Het moet zaken als routering, tussentijdse opslag en flow-control implementeren.

eerd file systeem moet zich daarbij ook bezig houden met plaatsbepaling van files, coördinatie van replicatie, en herstelmechanismen bij uitval van delen van het systeem.

Een ontwerpdoel van gedistribueerde systemen is de complexiteit terug te brengen tot de mate van complexiteit van een *gecentraliseerd* systeem.

semantiek van een concept

De taal van een laag—de interface—bestaat uit concepten en primitieven. De primitieven van een interface hebben een bepaalde betekenis: de semantiek. Deze betekenis vertegenwoordigt een deel van het probleemgebied—een deel van de te structureren *complexiteit*. Primitieven zijn bouwstenen waarmee men zinvolle structuren in het probleemgebied kan aanbrengen. Een primitieve staat voor een bepaalde hoeveelheid gestructureerde complexiteit.

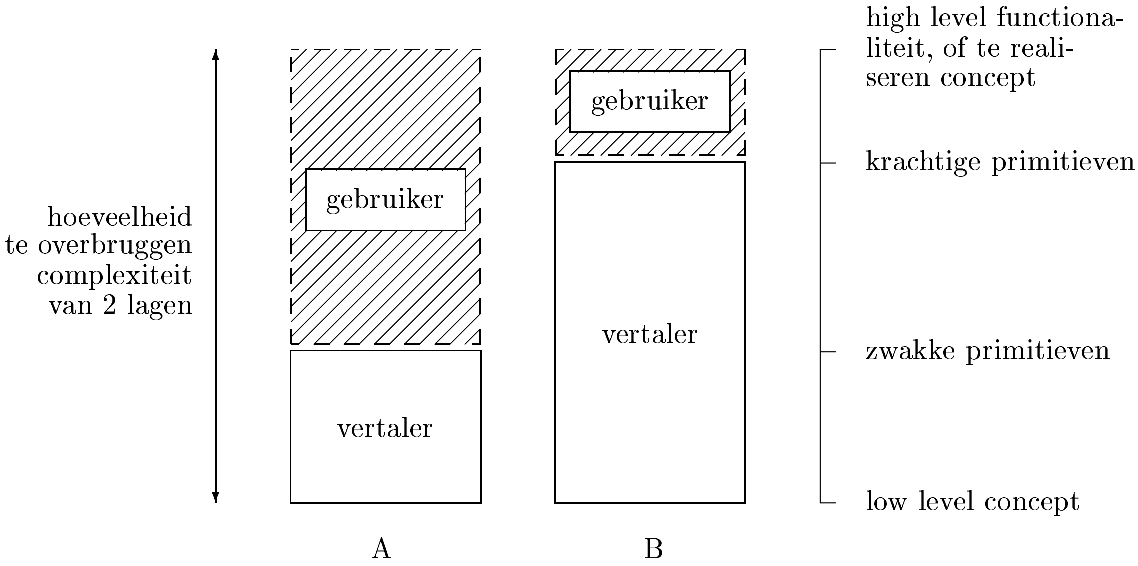
Primitieven die veel complexiteit vertegenwoordigen heten een 'krachtige semantiek' te hebben. Een gebruiker kan met weinig krachtige primitieven een probleem implementeren of representeren. Wanneer de virtuele machine genoeg krachtige primitieven aanbiedt is het relatief eenvoudig om een eenvoudige—snelle en foutvrije—oplossing voor een probleem te bouwen. Door enkele krachtige bouwstenen te combineren kan de gebruiker een grote hoeveelheid complexiteit structureren. Het gebruik van primitieven met een *beperkte* semantiek is complex. De gebruiker moet veel primitieven gebruiken om het gewenste hoeveelheid complexiteit te structureren. Dit vergt meer inventiviteit en inzicht van de ontwerper van een hogere laag dan bij krachtige primitieven. De kans is groot dat de hogere laag snel noch foutvrij wordt uitgevoerd.

Tot nu toe zijn primitieven vanuit de gebruiker beschreven. De andere kant van de primitieven wordt gevormd door de implementatie. De primitieven die een virtuele machine aanbiedt worden door een vertaler vertaald in primitieven van een lagere laag. Primitieven met een krachtige semantiek vertegenwoordigen veel complexiteit. Het is moeilijk om een snelle, foutvrije vertaler voor zulke primitieven te schrijven. Het is eenvoudiger om een grote hoeveelheid complexiteit *in een aantal stappen* te reduceren (structureren).[6] Voor primitieven met een beperkte semantiek is het schrijven van een vertaler voor een snelle, foutvrije vertaling eenvoudiger. De primitieven vertegenwoordigen minder complexiteit. Het vinden van een verband tussen *high-level* en *low-level* concepten is eenvoudiger.

In figuur 2.5 wordt krachtige en beperkte semantiek uitgebeeld. Primitieve A biedt een beperkte semantiek aan de gebruiker aan. De gebruiker moet veel complexiteit zelf structureren om in één keer het doel te bereiken—de *sematic gap* is groot. De ontwerper van de vertaalslag—de virtuele machine—implementeert een relatief eenvoudig probleem. De kans op fouten of een trage implementatie is bij de gebruiker groter dan bij de ontwerper van de vertaling.

[6]Dit is wat het lagenmodel doet: complexiteit in een aantal stappen tot een begrijpelijk niveau reduceren.

Figuur 2.5: Complexiteit en semantiek van primitieven



De situatie is bij primitieve B juist andersom. De gebruiker ziet krachtige primitieven die zich voor een snelle en eenvoudige implementatie lenen. De ontwerper van de vertaling heeft een zeer complexe materie gestructureerd met primitieve B. De kans dat deze vertaling snel en correct is, is kleiner dan bij primitieve A.

Voorbeelden van krachtige concepten met een trage implementatie zijn op level 5 logische talen als Prolog en op level 2 CISC-processorarchitecturen.[7] Voorbeelden van zwakke semantiek met een snelle implementatie zijn procedurele talen als C en RISC-processorarchitecturen.[8]

Voorbeelden in het kader van gedistribueerde systemen van krachtige concepten met een trage implementatie zijn logisch gemeenschappelijk geheugen (pagina 46) en zero-or-one semantiek (§ 3.3.3). Voorbeelden van zwakke semantiek met een snelle implementatie zijn point-to-point messages (pagina 43) en at-most-once semantiek (pagina 62).

Door naar 'elegante' concepten te zoeken kan de ontwerper van een vertaling proberen de complexiteit die men moet overbruggen terug te brengen. Een vertaalslag van high-level naar low-level primitieven kan op het eerste gezicht ingewikkeld zijn—een grote hoeveelheid complexiteit omspannen. Door gericht deze complexiteit te proberen te doorgronden en tot eenvoudige concepten te reduceren kan het mogelijk blijken toch een eenvoudige—snelle en foutvrije—vertaling te maken. In [Hoare] is het belang van streven naar eenvoud en elegantie bij het ontwerp van een taal beschreven.

Concepten die door de gebruiker als 'krachtig' worden gezien omvatten

[7] Complex Instruction Set Computer.
[8] Reduced Instruction Set Computer.

soms minder complexiteit dan deze veronderstelt.

2.3.5 Transparantie

Gedistribueerde hardware—meerdere processoren—is in het model van figuur 2.1 te vertalen als *meerdere level 2 virtuele machines*. Het gebruik van meerdere level 2 virtuele machines geeft de mogelijkheid van parallelle uitvoering van processen en verhoogde bedrijfszekerheid door replicatie.

Virtuele machines bieden de gebruiker in principe een andere interface dan hun voorganger. Soms echter laat een virtuele machine delen van de oorspronkelijke interface intact. Zo geven level 3 machines de level 2 interface ongewijzigd door aan level 4. Ze voegen slechts concepten toe (geprivilegeerde instructies worden *wel* afgeschermd). Het operating system biedt aan de assembler naast system-calls ook ongewijzigde machine-code instructies aan, en meerdere processoren.[9]

parallellisme bij applicaties

Op gecentraliseerde uniprocessoren worden sommige omvangrijke applicaties om beheersmatige (software engineering) redenen als pseudo-parallelle multiprocesprogramma's gemodelleerd [Rochkind]. In gedistribueerde omgevingen kunnen dit soort applicaties echt parallel uitgevoerd worden. Voor applicatiesoort 3 en 4 van § 2.1.4 kan dit voordelig zijn. Deze toepassingen maken gebruik van *expliciet parallellisme*.

Daarnaast zijn er toepassingen die geschikt zijn voor impliciet parallellisme. De applicatieprogrammeur heeft hierbij geen invloed op de verdeling in processen en kan het probleem als een enkelvoudig programma specificeren. Applicaties van soort 1 kunnen van de verhoogde snelheid van parallelle uitvoering profiteren.

Voor sommige toepassingen is de mogelijkheid parallellisme expliciet te specificeren noodzakelijk, voor andere is het mogelijk dat het parallelliseren door de compiler verricht wordt (§ 2.1.4).

In een gedistribueerde omgeving biedt de operating system-interface meerdere processoren aan de hogere virtuele machines aan. Waar voor sommige toepassingen impliciet parallellisme gewenst is moeten level 5 vertalers hierin voorzien. Sommige problemen laten zich goed specificeren in talen waar parallelliserende compilers voor bestaan.[10] Bij andere problemen moet de applicatieprogrammeur zelf een parallel ontwerp maken [Watt 1, Bal].

NOS-primitieven en GOS-primitieven

Een virtuele machine is 'gedistribueerd' als deze impliciet gebruik maakt van de mogelijkheden van gedistribueerde hardware, èn de complexiteit van de lagere virtuele machines voor zijn gebruiker afschermt. Het ideaal uit § 2.3.3

[9] Op pagina 33 wordt beschreven waarom meerdere processoren niet al door de operating system-laag als virtuele uniprocessor worden aangeboden.

[10] Voor bepaalde problem oriented programmeerparadigma's (zoals het procedurele) is het ontwerpen van parallelliserende compilers moeilijk [Bal].

is dat de eerste virtuele machine boven level 2 het abstraheren (wegstructureren) van de complexiteit implementeert—het operating system zou de vertaalslag in een keer moeten maken. In de praktijk zijn 'gedistribueerde' operating systems niet volledig transparant, de hogere lagen zien meerdere processoren.[11] Ze vormen een compromis tussen netwerk- en 'echt' gedistribueerde operating systems.

'Gedistribueerde' operating systems bieden zowel GOS-primitieven als NOS-primitieven aan. Primitieven die wel transparantie bieden, de GOS-primitieven, hebben bijvoorbeeld betrekking op het file systeem. De naam 'NOS-primitieven' is van toepassing op niet-transparante primitieven, bijvoorbeeld voor procesbeheer en interproces-communicatie. 'Gedistribueerde' operating systems vormen een compromis tussen snelle NOS-primitieven en transparante GOS-primitieven.

Dat met name file-systeem-primitieven transparantie bieden heeft te maken met de overhead van het resultaat van de vertaalslag tussen high level en low level concepten. De overhead is te groot voor operaties als primair geheugen-toegangen, die de snelheid van een systeem sterk beïnvloeden. De toegangstijd van secundair geheugen is ongeveer een factor 10^5 groter. Voor primitieven voor toegang tot het secundaire geheugen—als 'open bestand' en 'schrijf naar file'—is de overhead acceptabel, getuige de wijd verbreide acceptatie van operating systems met gedistribueerde file systemen en fileservers door gebruikers [Coulouris].

Bij een operating system dat niet volledig transparant is kan men proberen een level 5 virtuele machine te ontwerpen die de overgebleven problemen oplost. Men kan hiertoe een nieuwe high level language of een nieuwe vertaler voor een bestaande taal ontwerpen. De ontwerper van de level 6 interface—de applicatieprogrammeur—wordt in beide gevallen een transparante virtuele machine geboden. Bestaande applicatieprogramma's kunnen dan eenvoudig op de gedistribueerde systemen gebruikt worden. Om een transparante interface aan te kunnen bieden moet de level 5 vertaler—de compiler—de GOS-primitieven doorgeven en de NOS-primitieven vertalen in transparante concepten.

Wanneer emulatie van de primitieven van een bestaand operating system te complex is kan men er naar streven om de interface volgens *gelijksoortige concepten* en paradigma's op te zetten. Men hoeft de software dan niet volgens fundamenteel andere concepten te herschrijven. Het RPC-paradigma is met dit doel geïntroduceerd. De RPC biedt het veelgebruikte procedure-concept aan.

Voorbeelden van niet-transparante NOS-primitieven zijn de remote fork en point-to-point message passing. Voorbeelden van transparante GOS-primitieven zijn primitieven voor logisch virtueel geheugen en voor een gedistribueerd file systeem.

[11] Daar sommige applicaties van expliciet parallellisme gebruik maken, is het de vraag of een virtuele uniprocessor op level 3 onder alle omstandigheden wenselijk is.

transparantie-eisen

De taak van de level 5 vertaler is het eventueel vertalen van high level concepten in NOS-primitieven, en, indien geen expliciet parallellisme gewenst is, één processor in meerdere level 2 processoren.

Naast het transparant maken van NOS-primitieven moet een parallelliserende compiler op basis van meerdere level 2 processoren een virtuele uniprocessor interface aanbieden. Sommige conventionele level 5 interfaces zijn geschikt voor verdeling over meerdere level 2 processoren. Op pagina 39 worden het functionele en het logische programmeerparadigma als mogelijkheden genoemd die geschikt zijn voor een vertaling van sequentiële high level naar geparalleliseerde low level concepten.[12]

In het algemeen vereist de representatie van een probleem op een hoog abstractieniveau minder primitieven dan op een laag niveau. Hogere concepten worden in een groot aantal kleine lagere concepten vertaald. Op een laag niveau (level 2) omvatten de primitieven weinig complexiteit. De semantiek is beperkt, er moeten *veel* instructies *snel* worden uitgevoerd. Op een hoog niveau (level 5) bezitten primitieven een krachtiger semantiek. Problemen kunnen met minder primitieven worden gerepresenteerd.

Het detecteren van large grain parallellisme vereist een hoog-niveau-analyse met high level primitieven, zeker indien men applicaties om reden 3 en 4 van § 2.1.4 in parallelle eenheden wil partitioneren. Door van hogere, abstractere, minder gedetailleerde concepten uit te gaan is het mogelijk bijvoorbeeld de verdeling in parallelle processen of het gebruik van gemeenschappelijke variabelen te implementeren.

Het operating system speelt geen rol in deze vertaling. Het bevindt zich op een te laag niveau om deze high level concepten snel te kunnen vertalen. Het tranformeren van meerdere processoren in een virtuele uniprocessor—detectie van large grain parallellisme—vindt op level 5 (en eventueel hoger) plaats. Bij de beoordeling van level 5 omgevingen spelen naast conventionele beoordelingscriteria de volgende punten een rol:

- Ondersteunt de omgeving expliciet of impliciet parallellisme? Indien expliciet parallellisme mogelijk is, zijn de primitieven dan transparant?
- Welke level 3 NOS-primitieven worden niet door level 5 in transparante concepten vertaald?

Indien deze punten transparant zijn voor de applicatieprogrammeur is de vraag in hoeverre de omgeving nog flexibel is, of dat het implementeren van sommige problemen complex is geworden door de afscherming van bepaalde mogelijkheden.

Wanneer low level concepten niet zijn afgeschermd is het de vraag of de omgeving niet te complex is voor applicatieprogrammeurs, en meer voor systeemprogrammeurs geschikt is.

Het volgende hoofdstuk behandelt beoordelingscriteria voor deze vragen.

[12]Een complicatie hierbij is dat deze vorm van parallellisme veel communicatie tussen de processoren met zich mee brengt. Het is fine-grain parallellisme. Deze vorm is moeilijk op systemen zonder gemeenschappelijk geheugen snel te implementeren.

Hoofdstuk 3

Applicatieontwikkeling

Het doel van dit hoofdstuk is vast te stellen waar een goede gedistribueerde applicatieontwikkelomgeving aan moet voldoen.

3.1 Soorten toepassingen

De taal waarin toepassingsprogramma's worden geschreven is de level 5 interface uit figuur 2.1—de problem-oriented-language. De level 5 virtuele machine moet de fuctionaliteit die het gedistribueerde operating system aanbiedt geschikt maken voor het implementeren van toepassingen. De problem-oriented-language is de interface tussen het operating system en de gebruikersapplicaties. Kenmerkend voor een gedistribueerd *systeem* zijn de volgende punten (§ 2.3.1 en 2.3.3).

1. *Structuur* Een gedistribueerd systeem bestaat bij het werkstation/server-model uit geografisch verspreid opgestelde computers. Dit kunnen gelijksoortige systemen zijn, zoals een verzameling werkstations. Het is ook mogelijk dat bepaalde computers een specifieke functionaliteit hebben. Zo kan een machine voor databasetoepassingen geoptimaliseerd zijn, terwijl andere speciaal machines voor GUI-toepassingen geschikt zijn.

2. *Parallellisme en fouttolerantie* Als gevolg van de structuur kunnen processen gelijktijdig door meerdere processoren worden uitgevoerd, en gegevens op meerdere plaatsen opgeslagen worden.

Gebruikers wenden deze mogelijkheden voor het implementeren van toepassingen aan. De aplicatieontwikkelomgeving moet de mogelijkheden vertalen in een vorm die aansluit op de problematiek van de gebruikers.

Er zijn verschillende redenen waarom men *applicaties* op gedistribueerde systemen wil implementeren. In § 2.1.4 worden de volgende genoemd.

1. *Snelheid* Een kortere doorlooptijd voor een enkele berekening.

2. *Betrouwbaarheid* Toegenomen betrouwbaarheid en beschikbaarheid.

3. *Specialisatie* De mogelijkheid bepaalde delen van het systeem te gebruiken om specifieke functionaliteit efficiënt aan te kunnen bieden.

4. *Inherente gedistribueerdheid* De mogelijkheid inherente gedistribueerdheid van een applicatie te benutten.

Eén of meer van deze redenen kunnen aanleiding geven tot implementatie van een applicatie op een gedistribueerd systeem.

3.1.1 Parallellisme

De structuur van een gedistribueerd computersysteem maakt het mogelijk applicaties die een gedistribueerde structuur hebben te implementeren. Dit zijn applicaties die om reden 3 en 4 worden geïmplementeerd (bijvoorbeeld client/server-applicaties—zie § 3.4.4). Gedistribueerde applicaties zijn eenvoudig in verschillende processen te verdelen. Het parallellisme van deze toepassingen is large-grained—er wordt relatief infrequent gecommuniceerd.

De partitionering in processen is afhankelijk van de omgeving waarin de applicatie uitgevoerd gaat worden. Een parallelliserende compiler zou kennis over databaseservers, werkstations, en netwerktopologie moeten hebben. De programmeur is beter in staat tot het nemen van dit soort ontwerpbeslissingen. Bij een gedistribueerde probleemstructuur is een applicatie goed met *expliciet parallellisme* te implementeren. Voor reden 3 en 4 is een taal met expliciet parallellisme geschikt.

De andere redenen om een applicatie op een gedistribueerd systeem te implementeren zijn *snelheidsverhoging* en *betrouwbaarheid*—reden 1 en 2. Deze zeggen niets over de structuur van de applicatie. Het kan zowel om large- als om fine-grain parallellisme gaan, en de partitionering in processen kan voor de hand liggen, of juist complex zijn. Sommige applicaties zijn voor een (in bepaalde mate) parallelliserende compiler geschikt,[1] andere kunnen bij de huidige stand van de compilertechnologie beter door de programmeur gepartitioneerd worden. De keuze voor impliciet of expliciet parallellisme hangt van het soort probleem af.

Naar parallellisme en parallelle algoritmen wordt veel onderzoek verricht. Vanwege de complexiteit en omvang van de problematiek wordt parallellisme hier slechts zijdelings behandeld—voor zover het voor het onderwerp van belang is.

De keuze van de taal impliceert welke problemen geïmplementeerd kunnen worden. In figuur 2.5 en op pagina 29 is aangegeven dat de keuze van de semantiek van taalconcepten—in de zin van elegantie, flexibiliteit, efficiëntie en transparantie—bepalend is voor de mogelijkheid verschillende concepten/problemen efficiënt te kunnen representeren. De verschillende toepassingen die men moet kunnen implementeren bepalen uit welke taalelementen de level 5 interface moet bestaan.[2]

[1] Zoals problemen die in een functionele of logische taal (als ML en Prolog) gespecificeerd kunnen worden.

[2] Of, andersom, de taalelementen die de ontwerper gekozen heeft bepalen voor welke applicaties een taal geschikt is.

Talen met expliciet parallellisme stellen de applicatieprogrammeur in de gelegenheid zich expliciet met de verdeling in (pseudo-)parallelle processen bezig te houden. Talen met impliciet parallellisme komen dichter bij het 'virtuele uniprocessor' ideaal van § 2.3.3—ze zijn transparanter. Expliciet parallellisme biedt de programmeur meer mogelijkheden—het is flexibeler, maar legt de verantwoordelijkheid bij de programmeur.

3.1.2 Fouttolerantie

Naast parallellisme is fouttolerantie een belangrijk kenmerk van gedistribueerde systemen. Doordat een gedistribueerd systeem is opgebouwd uit meerdere gelijksoortige componenten kan men door het bewust inbouwen van redundantie bij uitval van enkele componenten de continuïteit van een applicatie verhogen.

Het aanbieden van een fouttransparante level 5 interface kost veel in termen van verwerkingskracht en communicatietijd (pagina 51). Uit het oogpunt van efficiëntie is het beter alleen die delen van de applicatie foutbestendig uit te voeren waarvoor het echt nodig is, met name bij applicaties die om een andere reden dan betrouwbaarheid op een gedistribueerd systeem geïmplementeerd worden.

Door niet de gehele taal fouttransparant te maken maar fouttolerante bouwstenen (objecten) aan te bieden wordt de flexibiliteit van de taal vergroot. De programmeur kan zelf de gewenste afweging tussen foutbestendigheid en snelheid (efficiëntie) voor de applicatie bepalen. De keuze voor fouttolerante bouwstenen of een geheel fouttransparante taal is—net als bij parallellisme—een afweging tusen transparantie en flexibiliteit die door de toepassing bepaald wordt [Bal].

Voor applicaties, waarbij foutbestendigheid niet van belang is kan men eventueel geheel afzien van het gebruik van talen die dit soort primitieven implementeren.

3.2 Programmeertaal

Er zijn drie benaderingen voor de concrete vormgeving van de level 5 interface: een sequentiële taal, een sequentiële uitgebreid met parallelle taalelementen, en een speciaal voor gedistribueerde omgevingen ontworpen taal.

3.2.1 Standaard sequentiële taal

Door op een gedistribueerd operating system een standaard sequentiële taal (bijvoorbeeld C of FORTRAN) te implementeren ontstaat een applicatieontwikkelomgeving waarop men in principe bestaande applicaties direct kan gebruiken. Afgezien van eventueel impliciet parallellisme wordt de extra functionaliteit van een gedistribueerde omgeving door middel van system-calls aan de programmeur aangeboden. Wanneer deze system-calls niet transparant zijn (NOS-primitieven) moeten bestaande programma's hieraan door de programmeur aangepast worden.

De meeste bestaande talen zijn in zo'n omgeving gebaseerd op expliciet parallellisme: de programmeur moet het probleem in verschillende parallelle eenheden partitioneren.[3] Bij veel operating systems zijn er meer NOS-primitieven dan GOS-primitieven. Vanwege hun zwakke semantiek geven deze system calls aanleiding tot complexe applicatieprogramma's. Een meer fundamenteel nadeel van standaard sequentiële talen is dat het programmeerparadigma vaak van sequentiële verwerking en één adresruimte uitgaat. Wanneer het operating system een logisch gedistribueerde virtuele machine aanbiedt moet de applicatieprogrammeur zelf rekening houden met concurrency-problemen en gescheiden adresruimten.

Het voordeel van een bestaande taal is de bekendheid voor applicatieprogrammeurs. Een nadeel is dat sequentiële en gedistribueerde concepten vaak slecht samengaan. Bij de beoordelingscriteria voor RPC systemen in § 3.4.2 komen voorbeelden hiervan aan bod. Bij het gebruik van een bestaande taal hangt de mate van transparantie sterk af van het operating system, en geeft al gauw aanleiding tot gecompliceerde programma's [Bal].

3.2.2 Uitgebreide sequentiële taal

Een mogelijkheid om de transparantie van het gebruik van low level NOS-primitieven te verhogen is het toevoegen van high level concepten aan een bestaande sequentiële taal. De primitieven voor parallellisme, communicatie en eventueel foutbestendigheid worden nu niet als system-calls aangesproken maar zijn in de taal geïntegreerd. Concurrent C en Concurrent Prolog zijn voorbeelden van deze benadering.

Het voordeel is dat de eventueel zwakke semantiek van operating system primitieven verhoogd kan worden waarbij nieuwe concepten in een voor de programmeur bekende omgeving geïntroduceerd worden. Het nadeel dat sequentiële en gedistribueerde concepten vaak slecht samengaan blijft bestaan. Daar komt bij dat hierdoor vaak een taalkundig slechte integratie tussen sequentiële en parallelle taalelementen ontstaat [Bal].

3.2.3 Gedistribueerde taal

Omdat integratie van gedistribueerde en sequentiële primitieven in een bestaande taal soms moeilijk te verwezenlijken is, zijn er talen speciaal voor parallel en gedistribueerd programmeren ontworpen. Deze talen bevatten alleen concepten die goed aansluiten op gedistribueerde omgevingen. Wanneer de taal impliciet parallellisme implementeert zijn dit concepten die efficiënt parallelliseerbaar door de vertaler zijn. Bij expliciet parallellisme ligt de nadruk op het verhogen van de transparantie van NOS-primitieven voor interproces-communicatie, bijvoorbeeld door ze in high level concepten als RPC te vertalen.

[3] Sinds de ontwikkeling van parallelle architecturen wordt er veel onderzoek gedaan naar talen met impliciet parallellisme vanwege de transparantie voor de programmeur [Bal]. Omdat deze talen speciaal voor gedistribueerde omgevingen ontwikkeld zijn passen ze beter in § 3.2.3 dan in § 3.2.1.

Het voordeel van deze talen is dat alle primitieven van de taal voor gedistribueerde omgevingen geschikt zijn. Het nadeel is dat een applicatieprogrammeur een nieuwe taal met nieuwe concepten moet leren. Er wordt veel onderzoek naar dit soort talen verricht. De keuze van een taal is vanwege het grote aanbod—om en nabij de honderd—geen eenvoudige zaak.

3.3 Beoordelingscriteria

Voor de beoordeling van een gedistribueerde applicatieontwikkelomgeving zijn in de literatuur een aantal criteria beschreven [Bal, Hoare, Mullender 1]. Bij de beoordeling van een gedistribueerde omgeving moeten *gedistribueerdheids*aspecten en algemene *omgevings*aspecten beoordeeld worden. Aspecten die met **gedistribueerdheid** samenhangen zijn:

1. *Parallellisme* Expliciet/impliciet, eenheid van parallellisme (grain), scheduling.
2. *Interproces-communicatie* Message-passing, data-sharing.
3. *Fouttolerantie* Fouttransparantie, fouttolerante objecten met atomaire transacties, exceptions.
4. *Integratie* Integratie van de semantiek van sequentiële en parallelle/gedistribueerde primitieven.

Naast deze aspecten moet een ontwikkelomgeving beoordeeld worden op gebruikelijke eigenschappen, die met de **omgeving** als geheel samenhangen:

5. *Complexiteit* De level 5 interface moet zo eenvoudig mogelijk zijn.
6. *Typesecurity* Het maken van fouten moet zoveel mogelijk door de omgeving voorkomen worden.
7. *Efficiëntie* Taalprimitieven moeten efficiënt uitgevoerd kunnen worden.
8. *Overdraagbaarheid* Door standaardisatie kan het overzetten van applicaties tussen verschillende platforms vereenvoudigd worden.

Het doel van de criteria is het mogelijk maken de Sun omgeving te beoordelen. Hierna wordt van de vier aspecten die het meest met gedistribueerdheid in verband staan beschreven welke mogelijkheden er voor het ontwerp van een level 5 interface zijn.

3.3.1 Parallellisme (1)

Parallellisme is een fundamentele eigenschap van gedistribueerde systemen. Door de aanwezigheid van meerdere processoren kunnen processen tegelijkertijd worden uitgevoerd. Er is een grote variëteit in benaderingen van parallellisme. Sommige talen proberen parallellisme volledig transparant te maken. Andere talen kennen expliciete taalelementen om parallelle eenheden op logisch niveau te specificeren. Deze talen zijn minder transparant, maar ze bieden de programmeur meer flexibiliteit. Nog een stap verder (in de zin van flexibiliteit) gaan talen die expliciet *fysieke* processoren kennen. In zulke talen kunnen processen op dezelfde processor variabelen gemeen hebben, en processen die de programmeur aan verschillende processoren heeft toegewezen kunnen dat niet.[4] De complexiteit van zo'n taal is groter, de flexibiliteit echter ook. Een nadeel van deze benadering is dat een programma afhankelijk wordt van de configuratie van een computersysteem. Het verschil tussen impliciet en expliciet parallellisme is het al of niet op level 5 zichtbaar zijn voor de programmeur. Impliciet parallellisme is minder flexibel en in hogere mate transparant.

eenheid van parallellisme

De eenheid van parallellisme is een fundamenteel punt waarop men gedistribueerde talen kan onderscheiden. In de volgende lijst worden de mogelijke ontwerpkeuzes behandeld.

- *Proces* De meeste gedistribueerde procedurele talen zijn gebaseerd op het begrip 'proces'. Veel operating systems bieden het proces (expliciet) als eenheid van parallellisme aan. Elk proces heeft zijn eigen toestand, code en gegevens. De instructies van een proces worden sequentieel uitgevoerd.

  Processen die veel toestandsinformatie bevatten worden *heavyweight* genoemd. Het gaat hier bijvoorbeeld om filedescriptors, signal handlers, gegevens over gebruikte rekentijd, en toestand van dochterprocessen. Een proces- of contextswitch kost veel tijd. Sommige operating systems staan een onderverdeling toe in kleinere *lightweight* processen (of *threads of control*). Lightweight processen delen een gemeenschappelijke adresruimte en toestand en communiceren via gemeenschappelijke variabelen.

  Op uniprocessors en loosely-coupled systemen worden threads pseudoparallel (vaak non-preemptive) uitgevoerd. Op tightly-coupled systemen kunnen ze echt parallel worden uitgevoerd.

  Heavyweight processen zijn geschikt voor large-grain parallellisme, en large-grain parallellisme is geschikt voor gedistribueerde omgevingen (zie § 2.3.1).

- *Object* Het begrip 'objectgeoriënteerd programmeren' zorgt voor net zoveel verwarring als de term 'gedistribueerd systeem'.

  Een object is een onafhankelijke eenheid die zowel *gegevens* als *bewerkingen op die gegevens* bevat en met de buitenwereld via messagepassing communiceert. De gegevens zijn alleen voor het object zelf direct bereikbaar. Het type van een object wordt zijn *class* genoemd.

[4] Voorbeelden hiervan zijn SR en Argus.

Classes kunnen via *inheritance* als uitbreidingen van een eerder gedefinieerde class opgebouwd worden. Het primaire doel van objectoriëntatie is het structureren van grote programma's. Getracht wordt de structuur van een probleem uit de werkelijkheid zoveel mogelijk in de structuur van het programma te vangen.

In een sequentiële objectgeoriënteerde taal wordt een object actief wanneer het een boodschap van een ander object ontvangt. Parallellisme kan op de volgende manieren geïntroduceerd worden:

– Laat objecten actief worden zonder eerst een boodschap af te hoeven wachten.
– Laat het ontvangende object doorgaan na het verzenden van het resultaat.
– Zendt boodschappen naar verschillende objecten tegelijk.
– Laat de verzender van een boodschap parallel aan de ontvanger doorgaan in plaats van op antwoord te moeten wachten.

De eerste twee methodes komen neer op het toekennen van een proces aan elk object. De objecten worden actief [America]. De derde methode gebruikt een multicast (zie pagina 44). De laatste methode kan met asynchrone in plaats van synchrone message-passing geïmplementeerd worden, of door een enkel object uit meerdere threads of control te laten bestaan.

- *Statement* Occam biedt de mogelijkheid om statements in blokken te groeperen die *parallel* of *sequentieel* moeten worden uitgevoerd. Deze vorm biedt de mogelijkheid op een laag abstractieniveau expliciet parallellisme voor fine-grain toepassingen te specificeren. Begin en einde van het parallellisme van een programma zijn duidelijk aangegeven.

  Deze methode biedt weinig ondersteuning voor het structureren van gedistribueerde programma's. Het aantal parallelle eenheden is at compile-time gefixeerd.

- *Expressie* In parallelle functionele talen is de eenheid van parallellisme de expressie. In een functionele taal hebben functies geen *side-effects*[5]—net als wiskundige functies. Dit in tegenstelling tot procedurele talen waarin functies bijvoorbeeld via globale variabelen en pointers elkaar kunnen beïnvloeden. Door de afwezigheid van side-effects maakt het niet uit in welke volgorde functies geëvalueerd worden—functies zijn *idempotent*. In de expressie `h(f(3,4), g(8))` kunnen `f` en `g` parallel geëvalueerd worden. Alle functieaanroepen kunnen parallel uitgevoerd worden. De enige beperking is dat `h` mogelijkerwijs op het resultaat van `f` en `g` moet wachten.

  Deze vorm van impliciet parallellisme is fine-grained. Hierdoor is zij geschikt voor shared-memory multiprocessors (§ 2.3.1). Voor gedistribueerde architecturen is de communicatieoverhead te groot.

[5] Zie pagina 62.

- *Clause* Logische talen bestaan uit clauses die waar of niet waar zijn. Het evalueren van deze clauses kan parallel geschieden. Dit wordt ook wel AND/OR parallellisme genoemd. Het is een vorm van impliciet fine-grain parallellisme—het is parallellisme waarbij relatief vaak gecommuniceerd wordt.

  Bij de implementatie van logische talen wordt gebruik gemaakt van backtracking. Er is geen bevredigende oplossing voor de efficiëntieproblemen van parallelle vormen van backtracking gevonden.

conclusie

Functionele en logische talen zijn volgens [Bal] vanwege de grain van het parallellisme niet voor loosely-coupled gedistribueerde omgevingen geschikt. Om large-grain parallellisme te implementeren wordt vaak voor processen of objecten gekozen. De keuze tussen objecten of processen als eenheid van parallellisme is eerder van het soort applicatie afhankelijk dan van taalontwerpoverwegingen [Bal].

scheduling

De strategie die ten grondslag ligt aan de toekenning van processoren en processortijd aan processen is van groot belang bij het ontwerp van een gedistribueerd systeem. De taak van de *scheduler* is de belasting optimaal over de beschikbare processoren te verdelen. In een gedistribueerde omgeving valt de schedulingproblematiek in twee delen uiteen: globale en lokale schedulingstrategieën. De globale scheduling bepaalt welke processen aan welke processor worden toegewezen. Lokale scheduling bepaalt de verdeling van de processortijd van een enkele processor over de processen. Over beide problemen is veel literatuur geschreven [Bal].

Het toekennen van processen aan processoren gaat in twee stappen. Eerst worden parallelle eenheden aan fysieke processoren toegekend. Daarna wordt met een prioriteitenmechanisme de lokale scheduling van processen op elke processor bepaald.

Zowel bij lokale als bij globale scheduling speelt de vraag op welk *moment* de scheduling wordt vastlegd. Dit kan (a) at compile-time, (b) at run-time, of (c) in het geheel niet vast zijn.

Het eerste geval is het minst dynamisch, maar heeft het voordeel dat de programmeur weet welke processen op dezelfde processor draaien. Hierdoor kan deze gebruik maken van het feit dat bepaalde processen geheugen delen.

Wanneer de toekenning at run-time plaats vindt, krijgt het proces bij creatie een processor toegewezen. Deze oplossing maakt een efficiënter gebruik van de hardwareresources van gedistribueerde systemen mogelijk.

De derde mogelijkheid, migratie—verplaatsing van processen van de ene processor naar de andere, wordt door weinig talen geboden. Processen kunnen in dit geval van veranderende systeembelastingen gebruik maken om steeds een zo adequaat mogelijk—snel of betrouwbaar—gebruik van de hardware te maken.

conclusie

Bij de toewijzing van processen aan processoren moet een afweging gemaakt worden tussen snelheid en betrouwbaarheid. Afhankelijk van het soort applicatie zal men kiezen voor maximale snelheid door parallellisme, of voor maximale beschikbaarheid door replicatie van processen. Aangezien het run-time systeem van de taal geen kennis over de applicatie heeft, biedt een taal die de keuze aan de applicatieprogrammeur laat meer mogelijkheden voor het optimaliseren van de performance. Zo'n taal met expliciete scheduling is complexer dan een taal waarin de toekenningsstrategie transparant is, maar ook flexibeler.

In de volgende tabel wordt ter illustratie een aantal talen opgesomd die de hiervoor besproken mogelijkheden implementeren.

Parallellisme [Bal]

primitieve		*voorbeeldtaal*
eenheid van parallellisme	proces	Ada, Concurrent C, Linda, NIL
	object	Emerald, Concurrent Smalltalk
	statement	Occam
	expressie	ParAlfl, FX-87
	clause	Concurrent Prolog, PARLOG
globale scheduling	compile-time	Occam, StarMod
	run-time	Concurrent Prolog, ParAlfl
	migratie	Emerald

3.3.2 Interproces-communicatie (2)

Bij een programma dat in meerdere processen verdeeld is moeten de programmadelen gegevens uitwisselen. Een level 5 interface kan gebaseerd zijn op een logisch gedistribueerde virtuele machine, of op logisch gemeenschappelijk geheugen. In een logisch gedistribueerde taal vindt de communicatie via expliciete primitieven plaats. Bij gemeenschappelijke variabelen verzorgt de vertaler de communicatie tussen de processen (impliciet). Deze laatste vorm is eenvoudig voor de programmeur—transparantie—maar moeilijk efficiënt te implementeren.

Aan gegevensuitwisseling tussen delen van een programma zijn twee kanten te onderkennen: communicatie en synchronisatie. Bijvoorbeeld: proces A heeft de uitkomst van een berekening die proces B uitvoert nodig. Deze uitkomst moet van B naar A—communicatie. Wanneer A de uitkomst nodig heeft maar B nog niet klaar is moet A er op kunnen wachten—synchronisatie.

Gegevensuitwisseling kan op twee manieren worden geïmplementeerd: met expliciete boodschappen (message-passing) en daarvan afgeleide concepten, en met logisch gemeenschappelijke variabelen.[6]

[6] Een logisch gemeenschappelijke variabele of *shared data* is iets anders dan een gemeenschappelijke, logische variabele. Zie daarvoor onder het kopje 'shared data' op pagina 45.

message-passing

De ontvangst van een boodschap kan *expliciet* of *impliciet* door de ontvanger plaatsvinden. Bij expliciete ontvangst moet de ontvanger een `accept`-primitieve uitvoeren waarin hij aangeeft welke boodschappen hij accepteert en welke acties er moeten worden uitgevoerd wanneer een boodschap aankomt. Bij impliciete ontvangst wordt er automatisch code bij de ontvanger uitgevoerd. Meestal wordt een nieuwe thread of control opgestart. Dit alles is transparant voor de verzender.

De soorten message-passing die door gedistribueerde talen ondersteund worden kan men in vier categorieën onderverdelen.

- *Point-to-point* Een point-to-point message is de eenvoudigste vorm van message-passing. Point-to-point messages zijn een low level vorm van communicatie. De verzender voert een `send`-primitieve uit en de ontvanger verwerkt de boodschap door een `receive`-primitieve uit te voeren.

  De belangrijkste ontwerpkeuze is de vorm van synchronisatie bij de verzender. Bij synchrone *verzending* wacht de verzender van een boodschap op een ontvangstbevestiging van de ontvanger. Bij asynchrone verzending wacht de verzender hier niet op.

  Door het uitvoeren van een synchrone *ontvangst*primitieve wacht de ontvanger op een boodschap. Asynchrone ontvangstprimitieven stellen de ontvanger in staat zonder eventueel geblokkeerd te worden te kijken of er een boodschap is (polling).

  Van de vier mogelijke combinaties (synchrone verzending en ontvangst, asynchrone verzending en ontvangst, synchrone verzending/asynchrone ontvangst, en asynchrone verzending/synchrone ontvangst) worden de vormen met asynchrone ontvangst in weinig talen toegepast. Asynchrone ontvangst blijkt—behalve voor polling—geen praktische manier voor het lezen van boodschappen te zijn [Laffra].

  Bij synchrone message-passing worden niet alleen gegevens uitgewisseld, processen synchroniseren ook. Synchrone verzending is niet zo concurrent als asynchrone omdat de verzender altijd wacht tot de ontvanger de boodschap geaccepteerd heeft, ook wanneer er geen antwoord nodig is.

  Asynchrone communicatie is ingewikkelder, zowel qua implementatie als gebruik. Bij dit model treden semantische problemen op. Daar de verzender niet wacht op de ontvanger kan het voorkomen dat er meerdere berichten bij de ontvanger in een wachtrij staan. Ze worden door het operating system of het run-time systeem van de taal gebufferd. Voor het beheersen van een buffer-overflow kunnen de ontwerpers kiezen uit twee mogelijkheden. Men kan de overdracht van de boodschap via een foutmelding laten falen. Dit verlaagt de betrouwbaarheid van de verbinding zoals de programmeur deze ziet. Men kan ook *flow control* toepassen en de verzender laten wachten tot de ontvanger weer in staat is boodschappen te verwerken. Hiermee introduceert men synchronisatie en de mogelijkheid van deadlocks zonder dat de programmeur hier rekening mee houdt.

De synchrone vorm is eenvoudiger te implementeren en makkelijker in het gebruik, maar verlaagt de mate van parallellisme. De verzender en ontvanger kunnen immers niet gelijktijdig worden uitgevoerd.

- *One-to-many* Voor operating systems en run-time systemen van talen is het soms wenselijk om een boodschap aan alle beschikbare processoren te zenden. Dit kan het geval zijn bij het lokaliseren van een server die een bepaalde dienst aanbiedt. Ook voor het consistent bijwerken van gerepliceerde gegevens is een *broadcast* communicatieprimitieve wenselijk [Birman]. Talen die logisch gemeenschappelijke gegevens implementeren maken intern vaak gebruik van multicast. De taal is dan *geïmplementeerd* met behulp van de multicast-primitieve. De taal biedt de programmeur deze primitieve *niet* zelf aan.

  Veel netwerksoftware biedt een broadcast- of multicastfaciliteit. Men onderscheidt gebufferde en ongebufferde broadcast systemen. In het laatste geval ontvangen alleen processen die er op wachten een boodschap. Bij gebufferde systemen zal elk proces uiteindelijk de boodschap ontvangen.

- *Rendez-vous* Point-to-point en one-to-many messages kunnen gebruikt worden om eenrichtingscommunicatie tussen twee processen te verzorgen. Veel interactievormen, zoals vraag/antwoord bij het client/server model (§ 3.4.4), vereisen twee-wegcommunicatie. De rendez-vous constructie biedt een high level alternatief voor twee point-to-point messages. Rendez-vous heeft een krachtiger semantiek. [Tanenbaum 5, p. 604] zegt: „Separate `send` and `receive` primitives can be thought of as the distributed system's answer to the `goto` statement: parallel spaghetti programming."

  Het rendez-vous model kent een client (zender) en een server (ontvanger) kant. Een server kan een *entry* declareren (een soort proceduredeclaratie) en met een *accept* statement acties specificeren die bij een interactie moeten worden uitgevoerd. Een client kan met een *entry call* (een soort procedureaanroep) parameters naar de server sturen.

  De interactie vindt synchroon plaats: de client wacht op de accept van de server of de server wacht op de entry call van de client. Nadat de interactie heeft plaatsgevonden worden client en server verder parallel uitgevoerd. In de literatuur zijn real-time toepassingen van Ada's rendez-vous beschreven [Bal].

- *Remote procedure call* De RPC is een andere twee-weg primitieve. De RPC is niet zoals rendez-vous voor real-time toepassingen bedoeld. Een RPC is een synchrone communicatievorm. Een RPC lijkt op het rendez-vous concept in een bekende abstractie. Het doel van RPC is om een procedureabstractie aan te bieden waarbij de communicatie transparant is voor de applicatieprogrammeur. Het verschil met de sequentiële, lokale procedure is dat RPC zich tussen twee verschillende processen afspeelt. In § 3.4 wordt de RPC verder besproken.

conclusie

Het belangrijkste keuzeprobleem bij message-passing is of men synchrone of asynchrone communicatie gebruikt. Het programmeren met synchrone communicatieprimitieven is eenvoudiger. Met asynchrone primitieven is een grotere mate van parallellisme te bereiken. Daarnaast speelt ook hier de keuze tussen de flexibiliteit van low level primitieven (point-to-point messages), en de transparantie van high level primitieven (RPC of rendez-vous). De keuze hangt af van de soort applicatie die men wil gaan implementeren [Bal].

shared data

Voor sommige toepassingen is het ontbreken van globale toestandsinformatie bij message-passing een probleem (zie § 3.4.5). Naast communicatie en synchronisatie met boodschappen kan men ook *shared data*, gemeenschappelijke variabelen, voor communicatie gebruiken. Hiermee is het mogelijk om op eenvoudige wijze globale toestandsinformatie te programmeren.

Het gebruik van gemeenschappelijke variabelen is reeds lang bekend als communicatie- en synchronisatiemechanisme tussen processen die pseudo-parallel op één processor uitgevoerd worden. Veel gedistribueerde talen ondersteunen zulke pseudo-parallelle processen en communicatie- en synchronisatiemechanismen. De `mutex` in Argus en semaforen in SR zijn hiervan een voorbeeld [Bal].

Er zijn ook talen die gemeenschappelijke variabelen tussen processen op verschillende processoren ondersteunen. In een gedistribueerd computersysteem worden deze modellen—bij afwezigheid van fysiek gemeenschappelijk geheugen—met message-passing geïmplementeerd. Het nabootsen van gemeenschappelijk geheugen op fysieke architecturen die dit niet bezitten (figuur 2.4) heeft voor- en nadelen. Een voordeel is dat gemeenschappelijke variabelen voor elk proces toegankelijk zijn, terwijl boodschappen—behalve bij one-to-many messages—gegevens tussen twee specifieke processen uitwisselen. Het toekennen van een waarde aan een gemeenschappelijke variabele wordt conceptueel direct geëffectueerd, terwijl bij message-passing tijd verstrijkt tussen verzending en ontvangst.

Een nadeel is dat het direct simuleren van de volledige semantiek van fysiek geheugen op gedistribueerde systemen een slechte responstijd oplevert. Dit wordt opgelost door bepaalde restricties aan de toegankelijkheid van gemeenschappelijke gegevens op te leggen. In [Bal] worden drie methoden beschreven om gemeenschappelijke variabelen op gedistribueerde computersystemen te implementeren: gedistribueerde gegevensstructuren, gemeenschappelijke logische variabelen en gemeenschappelijke gegevensobjecten. Daarnaast zijn er paradigma's voor de implementatie van gemeenschappelijk geheugen op operating system niveau [Van Renesse].

Het ontwerpdoel is transparantie voor de gebruiker—dezelfde semantiek als gewone globale variabelen—en de mogelijkheid van een efficiënte implementatie op systemen zonder fysiek gemeenschappelijk geheugen—gedistribueerde systemen.

- *Gemeenschappelijk fysiek geheugen* Parallelle computersystemen die over processoren met gemeenschappelijk fysiek geheugen beschikken zijn niet gedistribueerd.[7] Ze hebben het voordeel dat lezen en schrijven van gegevens (met behulp van level 2 `load` en `store` opdrachten) voor elk proces snel plaats vindt. Wanneer meerdere processen dezelfde geheugenlokatie gelijktijdig willen raadplegen kunnen race-condities zich voordoen. Hiervoor kan men op level 2 niveau—'in hardware'—bus locking toepassen. Bus locking is een vorm van `mutex`. Hiermee kan slechts één proces tegelijk een geheugenlokatie benaderen.

- *Gemeenschappelijk virtueel geheugen* Gemeenschappelijk virtueel geheugen is functioneel identiek aan gemeenschappelijk fysiek geheugen. Het verschil is dat de processoren niet via een bus maar via een netwerk verbonden zijn. Het verschil in toegangstijd van deze media bedraagt ongeveer een factor 10^4. Wanneer elke `load` of `store` van een proces via het netwerk zou gaan levert dit een onacceptabele responstijd op. Daarom wordt het geheugen in segmenten (pagina's) opgedeeld. Deze pagina's worden bij een page-fault van de memory management unit van een processor[8] van het primaire geheugen van de ene naar dat van de andere processor verplaatst. Bij deze verzoeken moet met race-condities en deadlock rekening worden gehouden [Van Renesse].

  Voor toepassingen die veel centrale variabelen—of veel verschillende geheugenlokaties—moeten bijwerken levert gemeenschappelijk virtueel geheugen een te trage respons op [Van Renesse, Bal]. Door de semantiek van primair geheugen niet volledig te ondersteunen probeert men in de volgende drie benaderingen een acceptabele respons te verwerkelijken.

- *Gedistribueerde gegevensstructuren* De taal Linda implementeert een *distributed data structure* genaamd 'Tuple Space'. Naast hun lokale geheugen beschikken processen over een logisch globaal geheugen waarin men van via een beperkt aantal operaties gegevens in de vorm van tuples kan opslaan.

  Een tuple is vergelijkbaar met een record in Pascal. Tuples hebben geen adres of index maar worden via associatieve adressering geïdentificeerd. Tuples worden op basis van hun inhoud gevonden. Er zijn drie operaties op tuples gedefinieerd: `in`, `out` en `read`. Met `out` plaatst men tuples in de Tuples Space, met `in` worden ze gelezen en er uit verwijderd, en met `read` worden ze gelezen. Om de waarde van een

[7] Ze zijn voor de volledigheid toch in deze opsomming opgenomen.

[8] Zie [De Bruin, Tanenbaum 1] voor een beschrijving van paged virtual memory.

variabele te veranderen moet een `in` èn een `out` operatie uitgevoerd worden.

Het run-time systeem van de taal verzorgt de verspreiding van de tuples over de processoren. Voor de applicatieprogrammeur is dit transparant.

Een nadeel van de Tuple Space van Linda is het ontbreken van een uniforme wijze van representeren van gegevensstructuren als een vector. Een efficiënte representatiewijze die een hoge mate van parallellisme toestaat vraagt een niet triviale oplossing van de applicatieprogrammeur. Een ander punt is de communicatieoverhead van Linda's runtime systeem. Operaties op Tuple Space variabelen kosten beduidend meer tijd dan operaties op lokale variabelen [Bal].

Door een beperkte semantiek van de gemeenschappelijke gegevens te implementeren met high level primitieven wordt een grotere efficiëntie bereikt dan bij gemeenschappelijk virtueel geheugen.

- *Gemeenschappelijke, logische variabelen* Concurrent Prolog en PARLOG zijn talen die gebaseerd zijn op *shared logical variables.* Deze variabelen bezitten de *single assignment* eigenschap. Wanneer er eenmaal een waarde aan is toegekend kan deze niet meer veranderd worden.

  Men kan ze als communicatie- en synchronisatie mechanisme gebruiken. De vereniging `doel_1(X,Y), doel_2(X,Y), doel_3(X)` kan parallel door proces P1, P2 en P3 opgelost worden. Zo gauw een proces een waarde aan `X` toekent kunnen de andere processen dit gebruiken. Eén proces krijgt het recht om een waarde aan `X` toe te kennen. De andere wachten zo lang op dat ene proces (synchronisatie). De gemeenschappelijke variabele `X` is een communicatiekanaal tussen de processen.

  Het modelleren van client/server toepassingen met meerdere clients per server geeft aanleiding tot ingewikkelde constructies. Het is moeilijk om een helder en duidelijk programma te schrijven met gemeenschappelijke, logische variabelen [Bal].

- *Gemeenschappelijke gegevensobjecten* Op pagina 39 is de procedurele kant van objecten besproken. Objecten kan men ook beschouwen als entiteiten die gedistribueerd gegevens kunnen bevatten. Daar objecten zelf de toegang tot hun gegevens bewaken, kunnen ze de synchronisatie regelen van andere processen (objecten) die toegang tot die gegevens wensen—analoog aan de synchronisatie van variabeletoegang van pseudo-parallelle processen door een monitor.

  De plaats van deze objecten is transparant voor de programmeur. Het is de taak van het run-time systeem van de taal een efficiënte verdeling van objecten over processoren te verzorgen.

  De toegang tot variabelen wordt bij lokale variabelen (variabelen op één processor) met simpele low level (level 2) `load` en `store` operaties verricht. Voor gedistribueerde systemen is deze oplossing te inefficiënt. In Orca [Bal] moet de applicatieprogrammeur zelf ondeelbare operaties op de gegevens van een object definiëren om het optreden van race condities te voorkomen. Deze operaties zijn van een hoger niveau dan `load` en `store`. Ze worden slechts eenmaal per variabeletoegang van een object uitgevoerd en geven zo minder communicatieoverhead.

- *File systemen* De bekendste manier om gegevens uit te wisselen is via bestanden. Bestandsoperaties zijn onder andere maken, lezen, schrijven en verwijderen. Daarbij worden file-systemen in multi-user en gedistribueerde omgevingen met synchronisatieprimitieven als locking voor concurrency-beheersing uitgerust.

  Veel traditionele pseudo-parallelle interproces-communicatie is op bestandsoperaties gebaseerd [Rochkind]. Een bestand is in beginsel een speciaal type object waarop in de meeste operating systems slechts een beperkt aantal operaties zijn toegestaan.[9] De snelheid van bestandsoperaties is vaak door de overhead van het operating system aanmerkelijk minder dan die van geheugenoperaties. Voor sommige toepassingen met large-grain parallellisme is interproces-communicatie via het file systeem voldoende snel [Van Renesse].

- *Databases* In het databaseparadigma worden gegevens in records gegroepeerd. Het adres van een record is een uniek veld genaamd 'sleutel' of *key*. Op records zijn creatie, lees, wijzig en verwijder operaties gedefinieerd. Records worden in het relationele model in tabellen geordend [Date]. Database management systemen (DBMS) bieden faciliteiten voor omvangrijke tabellen en fouttransparante toegang van veel gebruikers tegelijk (concurrency).

  Voor toepassingen die snelle toegang tot minder omvangrijke gegevensverzamelingen vereisen zijn relationele databases—al dan niet gedistribueerd—niet geschikt. Andere modellen voor fouttolerante toegang tot gegevens die niet op een databaseparadigmastructuur gebaseerd zijn worden in § 3.3.3 beschreven.

conclusie

Volledige transparantie èn efficiëntie zijn voor veel applicaties niet te bereiken. De toegestane bewerkingen op gemeenschappelijke gegevens zijn beperkt of de programmeur moet ze zelf specificeren.

Voor applicaties die geen gemeenschappelijke variabelen vereisen[10] zijn message-passing extensies van conventionele, procedurele talen beter geschikt. De zwakke semantiek van shared-data talen wordt dan vermeden. Procedurele talen gelden als flexibeler dan andere talen [Bal].

[9] In operating systems met *memory mapped files* als MULTICS zijn alle level 2 primair geheugenoperaties mogelijk [Tanenbaum 1].

[10] In tegenstelling tot sommige optimaliseringsvraagstukken: deze zijn eenvoudiger in shared-data talen te programmeren (§ 3.4.5).

In de volgende tabel wordt ter illustratie een aantal talen opgesomd die de hiervoor besproken mogelijkheden implementeren.

Communicatie [Bal]

primitieve		*voorbeeldtaal*
message passing	point-to-point messages	CSP, Occam, NIL
	rendez-vous	Ada, Concurrent C
	remote procedure call	DP, Concurrent CLU, LYNX
	one-to-many messages	BSP, StarMod
shared data	gedistribueerde variabelen	Linda
	gemeenschappelijke objecten	Orca
	gemeensch. logische variabelen	Concurrent Prolog, PARLOG

3.3.3 Fouttolerantie (3)

Volgens § 2.1.4 en § 2.3.3 is een tweede fundamentele eigenschap van gedistribueerde systemen fouttolerantie. Door gegevens en processen op verschillende processoren te dupliceren kan de betrouwbaarheid en beschikbaarheid van een applicatie vergroot worden [Mullender 1].

Gedistribueerde computersystemen zijn potentieel in staat een hogere mate van fouttolerantie te bieden dan gecentraliseerde systemen. Ontwerpers van een taal kunnen voor fouttolerantie uit drie benaderingen kiezen: (a) alles aan de programmeur overlaten, (b) atomaire transacties aanbieden, of (c) een fouttransparant run-time systeem aanbieden. Deze drie benaderingen worden hieronder besproken.

de programmeur

Voor taalontwerpers is het negeren van de mogelijkheid van fouten in lagere virtuele machines de simpelste oplossing. Voor sommige toepassingen is dit aanvaardbaar omdat processor- en andere fouten zelden voorkomen [Coulouris].

Een andere oplossing is de applicatieprogrammeur de mogelijkheid te bieden fouten zelf adequaat af te handelen. Wanneer het operating system of het run-time systeem van de taal een fout detecteert kan het programma op basis van de foutcode van een procedure actie ondernemen. Een variant hierop is de programmeur de mogelijkheid te bieden een exception handler voor bepaalde soorten fouten te schrijven. Deze exception handler wordt door het taal-run-time-systeem buiten het programma om aangeroepen. Dit resulteert in duidelijker programmatekst omdat de code voor de uitzonderingssituaties niet tussen de code van het eigenlijke algoritme staat.

Er zijn systemen die exceptions centraliseren in een *exception server*. Een applicatie kan de exception server laten weten welke acties er in het geval van bepaalde calamiteiten ondernomen moeten worden. Het operating system zendt foutmeldingen niet meer naar de applicatie, maar naar de exception server.

Een ander mechanisme is de *boot server* van het Amoeba systeem. Deze kijkt periodiek of vooraf geregistreerde processen nog bestaan. Zoniet, dan wordt een nieuwe versie van het proces opgestart. Op deze wijze zorgt het operating system ervoor dat belangrijke functies steeds in het systeem aanwezig zijn [Coulouris].

De acties, die de programmeur moet nemen, verschillen per soort applicatie. Wanneer een procedure idempotent is (geen side-effects heeft, pagina 62) kan bij een processorfout dezelfde procedure eenvoudig aan een andere processor worden aangeboden. Wanneer een procedure side effects heeft—zoals een getal in een bestand veranderen—moet bij herstarten van de procedure na een fout precies zijn bijgehouden welke side effects hebben plaatsgevonden en welke nog niet. Voor toepassingsprogramma's is het eenvoudiger deze procedures in een taal met meer faciliteiten op het gebied van fouttolerantie te schrijven.

atomaire transacties

Verschillende soorten applicaties stellen verschillende eisen aan snelheid en bedrijfszekerheid. Om deze reden zijn er omgevingen die de applicatieprogrammeur de mogelijkheid geven om zelf de benodigde mate van bedrijfszekerheid te kiezen. Op systemen die nauwelijks faciliteiten op dit gebied bieden kunnen idempotente toepassingen eenvoudig fouttolerant gemaakt worden. Voor algoritmes met side-effects wordt de implementatie vereenvoudigd door faciliteiten als atomaire transacties.

Talen die ondeelbare operaties—atomaire transacties—aanbieden maken het programmeren van fouttolerante applicaties met side effects minder complex, arbeidsintensief en foutgevoelig.

Een gedistribueerd programma met side-effects is te beschouwen als een verzameling parallelle processen die operaties op gegevenselementen uitvoeren.[11] Door aan te geven welke operaties niet los van elkaar mogen worden uitgevoerd kan de programmeur de consistentie van gegevensobjecten waarborgen [Date].

Technieken voor de implementatie van atomaire transacties in een gedistribueerde context zijn in [Davidson, Liskov, Spector] beschreven.

Het vastleggen van toestandsinformatie kost tijd. Uit vergelijkingen tussen remote procedure calls en atomaire transacties is gebleken dat transacties vele malen langer duren [Bal]. Voor het verkrijgen van een write-lock wordt eerst een kopie van het object gemaakt. Dit kan tot 1 ms duren. Een RPC duurde tientallen ms, een transactie honderden ms. Het soort fouttolerante primitieven dat Argus, Aeolus en Camelot [Spector] aanbieden is voor sommige toepassingen niet snel genoeg. Voor toepassingen als parallel sorteren

[11] Bijvoorbeeld het overmaken van een bedrag van de ene naar de andere bankrekening. De eerste operatie verlaagt saldo A, de tweede verhoogt saldo B. Vanwege de mogelijkheid van een processorfout halverwege is het in het belang van A dat beide operaties ondeelbaar worden uitgevoerd—helemaal òf helemaal niet.

zijn andere oplossingen met minder overhead nodig [Bal].

fouttransparantie

Er zijn talen die processorfouten geheel verhullen voor de applicatieprogrammeur. Omdat de taalimplementatie geen kennis over de applicatie heeft kan het geen onderscheid maken in processen of gegevens die essentieel zijn voor de applicatie en welke niet. Elk proces of gegevenselement wordt gelijk behandeld. Om te voorkomen dat de overhead van dit soort systemen te groot wordt is gezocht naar efficiënte herstelmethodes.

Voor een message-passing systeem gebaseerd op processen kan men voor elk proces een backup-proces op een aparte processor achter de hand houden. Dit proces bewaart automatisch alle boodschappen die naar het primaire proces verzonden zijn. Periodiek wordt de toestand van het primaire proces naar de backup gekopieerd. De boodschappen kunnen dan gewist worden. Wanneer het primaire proces niet meer functioneert neemt het backup-proces het werk over op basis van de vorige toestand en het logboek met verzonden boodschappen [Bal].

Deze benadering kost extra processoren, en tijd voor het maken van checkpoints.

NIL gebruikt optimistic recovery. Deze techniek is ook gebaseerd op checkpoints en logboeken met alle verzonden boodschappen. Er wordt hier geen backup proces gebruikt. Checkpoints en boodschappen worden naar stable storage (foutvrij achtergrondgeheugen) geschreven. In tegenstelling tot de vorige methode gebeurt dit wegschrijven asynchroon aan de normale berekeningen. Aangenomen dat de I/O bandbreedte voldoende is wordt de doorlooptijd hierdoor niet negatief beïnvloed. In de literatuur zijn geen tijdmetingen beschreven [Bal].

In de volgende tabel wordt ter illustratie een aantal talen opgesomd die de hiervoor besproken mogelijkheden implementeren.

Fouttolerantie [Bal]

primitieve	*voorbeeldtaal*
foutdetectie	Ada, SR
atomaire transacties	Argus, Aeolus, Avalon
fouttransparantie	NIL

conclusie: kosten

De prijs die voor fouttolerantie betaald moet worden wordt door twee factoren bepaald. Wanneer een proces een aantal maal gerepliceerd wordt kost dit evenzoveel processoren. Bij een parallel programma met lineaire versnelling kost tweevoudige replicatie een factor twee in snelheid. Ten tweede is de communicatieoverhead van de gerepliceerde processen aanzienlijk. Volgens

[Bal] is de prijs die voor fouttolerantie betaalt moet worden hoog. Fouttolerantie moet alleen toegepast worden wanneer dit noodzakelijk is voor de toepassing.

3.3.4 Integratie (4)

De integratie tussen sequentiële en gedistribueerde taalconstructies is vaak problematisch. De complexiteit van een programmeertaal hangt nauw samen met de semantiek van de primitieven. De kans is groot dat een taal die is samengesteld uit primitieven die gebaseerd zijn op twee fundamenteel verschillende paradigma's ingewikkeld is (§ 3.2.2). Dit geldt nog sterker wanneer primitieven van twee verschillende lagen gebruikt moeten worden (§ 3.2.1).

Een taal die geen constructies voor expliciet parallellisme kent, of volgens § 3.2.3 is opgezet kent deze integratie problemen niet.

De integratie van sequentiële en gedistribueerde primitieven levert op sommige punten problemen op. Gedistribueerde systemen, zonder gemeenschappelijke adresruimte, ondersteunen het gebruik van globale variabelen, pointervariabelen en referenceparameters niet. Veel sequentiële talen staan constructies toe die in een gedistribueerde context onjuist zijn. Dit probleem wordt in § 3.4.2 voor RPC-systemen behandeld.

In sommige sequentiële talen worden reference- of pointerparameters gebruikt. In een gedistribueerde omgeving geeft dit problemen omdat pointers en variabelereferenties alleen betekenis hebben binnen in één adresruimte. Procedurele sequentiële talen staan het gebruik van globale variabelen toe. Het simuleren van logisch shared-memory op fysiek gedistribueerde systemen geeft (pagina 45) problemen. De efficiëntie of de semantiek van variabelen uit de sequentiële taal gaan verloren.

Het resultaat is dat de semantiek van globale variabelen op een gecentraliseerde multiprocessor architectuur anders is dan op een gedistribueerde architectuur.

3.3.5 Beschrijving aspecten omgeving

In deze paragraaf wordt van de vier aspecten die met algemene aspecten van de omgeving in verband staan beschreven welke punten bij het ontwerp van een level 5 interface van belang zijn.

Omdat deze aspecten niet specifiek voor gedistribueerde omgevingen zijn worden ze hier beknopt behandeld.

complexiteit (5)

De complexiteit van een programmeertaal neemt af naarmate de primitieven een krachtiger semantiek bezitten (§ 2.3.4) en een grotere samenhang vertonen (§ 3.2).

De system calls van veel operating systems zijn deels NOS-primitieven, deels GOS-primitieven. Een level 5 omgeving die bestaat uit een sequentiële taal en overwegend NOS-primitieven, biedt, zeker bij afwezigheid van impliciet parallellisme, weinig transparantie.

Het bestaan van een aantal niet-transparante NOS-primitieven aan een sequentiële taal kan het schrijven van programma's aanmerkelijk compliceren. Wanneer het operating system niet de vereiste high level primitieven biedt moet de programmeur deze zelf bouwen. Ook de afwezigheid van ondersteuning van de gewenste mate van fouttolerantie kan de ontwikkeltijd en de kans op fouten nadelig beïnvloeden.

Het aanroepen van NOS-primitieven in de vorm van geparameteriseerde procedureaanroepen verlaagt de leesbaarheid van programma's. Om deze reden bestaan van veel sequentiële talen gedistribueerde varianten, meestal 'Concurrent X' of 'Parallel X' genoemd (§ 3.2.2).

Ook wanneer een taal volgens één principe is ontworpen—òf sequentieel, òf gedistribueerd—is een zorgvuldige keuze van paradigma, concepten en primitieven van groot belang. [Hoare] bespreekt het belang van een simpele taal met elegante primitieven. Ontwerpers van een nieuwe, gedistribueerde taal hoeven geen rekening te houden met de integratie van twee verschillende concepten. Ze zijn vrij om de structuur van de taal geheel op de nieuwe elementen af te stemmen. Ada is met haar veelheid aan concepten en primitieven een voorbeeld van een onnodig complexe taal [Bal]. Linda is een voorbeeld van een simpele taal. Het Tuple Space model kent slechts vier primitieven en is eenvoudig te begrijpen [Bal].

typesecurity (6)

Volgens [Hoare] moeten *alle* afwijkingen van de taalregels door compiler of run-time system worden gedetecteerd. Dit kan met compile-time checks, of met run-time checks van bijvoorbeeld array-indices en pointervariabelen. Veel veelgebruikte talen, zoals C en Pascal, zondigen tegen dit principe. [Hoare, p. 76] zegt hierover: „In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law." Talen moeten zou min mogelijk onveilige constructies toelaten.

Sequentiële talen staan gebruik van variabelen en parameters toe die in een gedistribueerde omgeving geen betekenis hebben. Doordat de verschillende processen door compiler en run-time system van de taal niet als onderdeel van één programma gezien worden is foutdetectie tussen processen onderling niet mogelijk. Operating systems voeren bij het uitvoeren van interproces-communicatie meestal geen type-checks uit.

Het opsporen van fouten in gedistribueerde programma's is ingewikkeld. Omdat het verloop vaak non-deterministisch is, zijn programmeerfouten zeer moeilijk te reproduceren. Zelfs wanneer een duidelijke foutmelding optreedt, kan het een probleem zijn de foutieve code te lokaliseren [Bal]. Daarnaast kan een verborgen fout uit een proces een correct proces in dezelfde (logische) adresruimte corrumperen. Juist gedistribueerde talen moeten typesecure zijn.

De kans op fouten is groter naarmate de complexiteit van een taal groter is (punt 5). Het nut van typesecurity wordt dan steeds groter. Ada is tot op zekere hoogte typesecure. Via het concept 'typestate' wordt in NIL typesecurity gerealiseerd [Watt 1, Bal].

Typesecurity verkort de testperiode van programmatuurontwikkeling, het verlaagt echter de efficiëntie van de uitvoering van een programma.

efficiëntie (7)

In paragraaf 2.3.4 en 2.3.5 is beschreven dat de keuze van paradigma's en primitieven een grote rol speelt voor het efficiënt kunnen implementeren van high level concepten. Sommige paradigma's staan de efficiënte implementatie van bepaalde toepassingen in de weg—zoals RPC bij branch-and-bound algoritmen—en voor sommige paradigma's is het moeilijk een efficiënte vertaling te maken—zoals logisch virtueel geheugen op gedistribueerde hardware.

De opbouw van computersystemen volgens het lagenmodel van Tanenbaum brengt een zekere inefficiëntie met zich mee. Wanneer een level 5 virtuele machine geïmplementeerd wordt op een operating system dat niet speciaal voor hoge overdrachtssnelheid gestructureerd is kan het gebeuren dat communicatie niet op de meest efficiënte wijze plaats vindt (zie § 4.2.1).

Efficiëntie is een relatief begrip. De vraag of een applicatie efficiënt geïmplementeerd kan worden hangt nauw samen met de vraag of de architectuur en het programmeerparadigma bij de applicatie passen. Multiprocessor architecturen met gemeenschappelijk geheugen zijn meer geschikt voor applicaties met fine-grain parallellisme dan gedistribueerde architecturen.

Uit vergelijkingen in de literatuur kan men de volgende conclusies trekken [Bal]:

- Bij de implementatie van communicatieprimitieven kunnen onderliggende operating systems met een minder geschikte structuur (bijvoorbeeld UNIX) veel overhead opleveren. Low level communicatiesoftware levert in het algemeen geen betrouwbaar transport van boodschappen op. Software protocols in hogere lagen bieden dit aan. Deze bevinden zich vaak in de kernel van operating systems. Bij de aanroep van een communicatieprotocol wordt een context-switch uitgevoerd. Bij het ontwerp van efficiënte gegevensoverdracht speelt minimaliseren van deze overhead een grote rol [Coulouris, Birrell].

- De uitvoering van run-time checks door een taal kost tijd. Typesecurity heeft een prijs.

- Procedurele talen zijn efficiënter dan functionele of logische talen. Door sommige optimaliserende compilers van functionele talen wordt het verschil tot een factor twee teruggebracht.

- Gedistribueerde implementaties van talen zijn duidelijk trager dan implementaties op shared-memory multiprocessors.

  Door onderzoek naar snellere implementatietechnieken en hardware wordt de communicatieoverhead van gedistribueerde systemen steeds kleiner. De prestaties van gedistribueerde systemen zullen in de toekomst steeds beter worden.

overdraagbaarheid (8)

Om een applicatie eenvoudig van het ene systeem naar het andere systeem te kunnen overzetten moet de taal aan een aantal eisen voldoen. Een taal moet op een aantal gedistribueerde computerconfiguraties, processortypen, en operating systems te implementeren zijn. De taal moet zo veel mogelijk algemene concepten implementeren, en zo min mogelijk architectuurspecifieke kenmerken.

De overdraagbaarheid van applicaties van het ene platform (operating system) naar het andere is voor programma's die veel operating system afhankelijke primitieven benutten een groter probleem dan voor applicaties die dat niet doen.

De problemen bij het overzetten van een applicatie zijn het grootst bij delen van de applicatie die niet aan een standaard voldoen. Wanneer men een programma wil overdragen op een operating system dat gebaseerd is op andere concepten kan het voorkomen dat een substantieel deel van de programmatekst moet worden herschreven [Bal].

Wanneer de gehele applicatie in één (nieuwe) gestandaardiseerde taal is geschreven is het overdragen van een applicatie eenvoudiger. De kans dat de voor een toepassing benodigde mogelijkheden op een overdraagbare manier geprogrammeerd kunnen worden is groter naarmate de taal meer primitieven implementeert en gebruik van system calls niet nodig is.

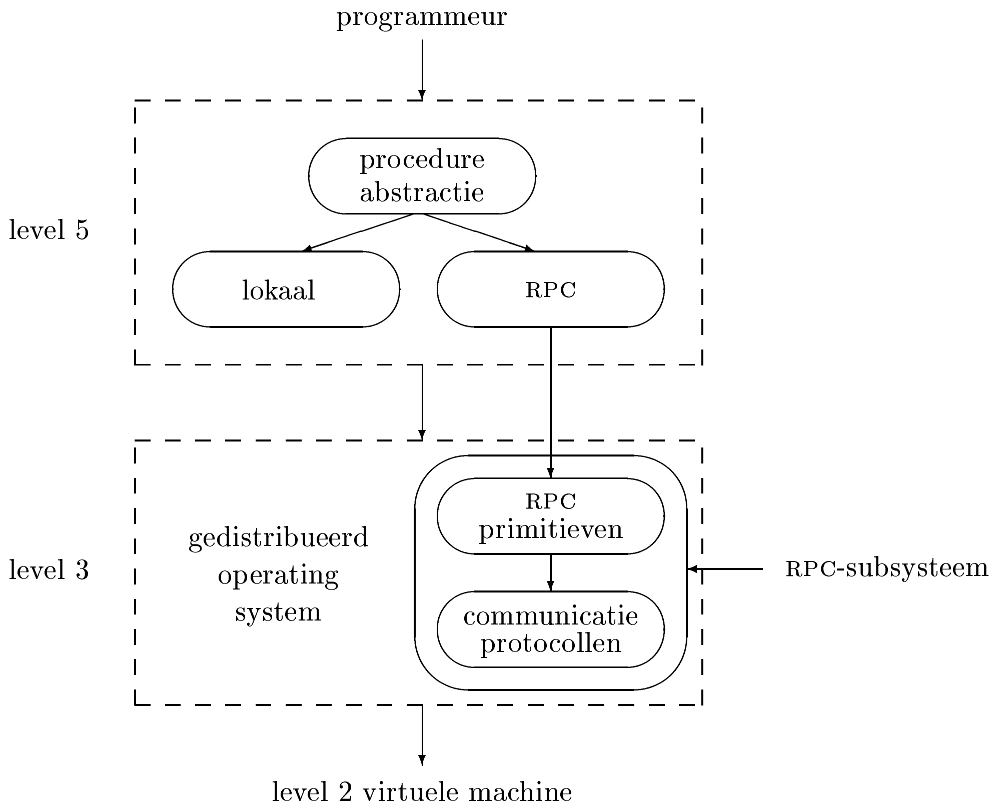
Een applicatieontwikkelomgeving moet zoveel mogelijk aan standaards voldoen.

3.4 Remote Procedure Call

Veel gedistribueerde operating systems ondersteunen de remote procedure call als communicatieprimitieve [Coulouris, Bal, Tanenbaum 4, Weihl 1]. Ook het gedistribueerde computersysteem van Sun, het onderwerp van deze scriptie, werkt met RPC's. Daarom wordt het hier uitgebreid behandeld.

3.4.1 Algemene kenmerken

Figuur 3.1: Plaats van RPC



Een RPC-systeem is een deel van het gedistribueerde operating system. Het breidt de taal die de operating system laag aanbiedt (figuur 2.1) met een aantal RPC-primitieven uit. De RPC-primitieven worden door de problem-oriented-language laag aan de applicatieprogrammeur aangeboden. Deze ziet een programmeertaal waarin remote procedures geïntegreerd zijn. Ontwerpers van RPC-systemen proberen de nieuwe primitieven zoveel mogelijk op bestaande abstracties uit de problem-oriented virtuele machine aan te laten sluiten.

In figuur 3.1 is in figuur 2.1 een RPC-subsysteem toegevoegd aan het operating system.[12] Dit subsysteem tracht communicatieprimitieven in de vorm van een bekende abstractie aan de programmeur aan te bieden.

Gedistribueerde applicaties—applicaties die om reden 3 en 4 van § 2.1.4 op gedistribueerde systemen worden geïmplementeerd—worden vaak volgens het client/server model ontworpen (zie § 3.4.4). In plaats van verzoeken van de client aan de server met point-to-point boodschappen te implementeren kan men hiervoor ook een enkele communicatieprimitieve van een hoger niveau gebruiken. Rendez-vous (pagina 44) en remote procedure call zijn zulke hoog-niveau primitieven.

Veel programmeurs zijn met de procedureabstractie bekend. De RPC probeert deze abstractie in een gedistribueerde omgeving te bieden. Eén van de redenen dat veel gedistribueerde omgevingen op RPC gebaseerd zijn is de bekendheid met de procedureabstractie.

procedure

De remote procedure call implementeert een communicatie- en synchronisatiemiddel *in de vorm van een procedureabstractie*. Hier volgen enkele kenmerken van de procedureabstractie.

De eenheid van parallellisme van procedurele talen is het *proces*. Het proceduremechanisme bestaat in sequentiële procedurele talen uit een procedure*definitie* en een procedure*aanroep*. In de definitie beschrijft de programmeur de opdrachten die moeten worden uitgevoerd wanneer de procedure wordt aangeroepen. De aanroeper kan bij de aanroep *parameters* meegeven om de functionaliteit van de procedure te beïnvloeden.

Procedures kunnen operaties op globale en lokale variabelen uitvoeren. De semantiek van parameters kan door het gebruik van verschillende overdrachtmechanismen worden veranderd [Watt 1]. Een parameter die *by reference* wordt overgedragen is als een globale variabele te beschouwen. Een *value*parameter is te beschouwen als een geïnitialiseerde lokale variabele [Van der Sluis].

gemeenschappelijke adresruimte

De remote procedure call implementeert een communicatie- en synchronisatiemiddel *tussen processen die geen gemeenschappelijk geheugen hebben*.

Binnen conventionele monolitische programma's hebben procedures dezelfde geheugenruimte. Gedistribueerde applicaties delen de (fysieke) adresruimte niet. Ze bestaan uit verschillende processen. Alleen procedures binnen eenzelfde (heavy-weight) proces kunnen over dezelfde adresruimte beschikken. De aanroep van een procedure vanuit een andere adresruimte is een *remote* procedure call. Een remote procedure call is een interprocesprocedureaanroep. De gewone aanroep, binnen eenzelfde adresruimte, wordt ter onderscheiding wel *lokale* procedureaanroep genoemd.

[12]Level 4 is voor de overzichtelijkheid weg gelaten.

communicatie

RPC is een vorm van message-passing. Onzichtbaar voor de applicatieprogrammeur wordt de procedure aanroep door het RPC-subsysteem geïmplementeerd met boodschappen. Meestal zijn dit point-to-point messages, soms ook one-to-many messages.

Een RPC speelt zich tussen twee partijen af: de aanroeper en de aangeroepene. Door het ontbreken van gemeenschappelijk geheugen hebben de twee partijen geen globale variabelen gemeen. Gegevensoverdracht kan slechts via parameters plaatsvinden. Variabelen met geheugenadressen hebben geldigheid *binnen* een proces. Pointervariabelen en referenceparameters bevatten derhalve binnen de context van het andere proces onjuiste informatie.

RPC is een synchrone communicatievorm: de aanroeper wacht op antwoord. In veel gedistribueerde omgevingen worden servers voor bepaalde diensten ingesteld. Deze servers hebben met meerdere clients te maken. Wanneer er een aanvraag voor service binnenkomt terwijl een vorig verzoek nog niet is afgehandeld zijn er verschillende oplossingen mogelijk. De server kan het nieuwe verzoek laten wachten. Ter verhoging van de mate van parallellisme binnen het systeem kan de server ook een extra proces voor het nieuwe verzoek opstarten dat dit direct gaat verwerken.

semantiek en transparantie

De semantiek van de RPC moet zo veel mogelijk op die van de gebruikelijke procedureabstractie lijken [Weihl 1]. Wanneer bij een gewone lokale procedure aanroep een processor faalt stoppen zowel aanroeper als aangeroepene. In een gedistribueerde omgeving kan het gebeuren dat één van beide processoren faalt. Bij herstarten is onbekend welk deel van de procedure aanroep is uitgevoerd en welk deel moet worden overgedaan. De semantiek van een remote aanroep is in principe anders dan van een lokale aanroep. Een ontwerpdoel van RPC-implementaties is om zo veel mogelijk transparantie op dit punt te bieden [Birrell].

Het optreden van systeemfouten kan op verschillende wijzen worden afgehandeld. Een gebruikelijke manier is om procedures foutcodes te laten teruggeven. Na elke procedure moet de applicatieprogrammeur code schrijven om deze foutcodes te testen. Een wat betere methode is de programmeur exceptionhandlers te laten definiëren. Hierdoor wordt de code voor de foutafhandeling uit het algoritme naar een centrale plaats in de code gebracht [Coulouris, Bal].

taal

RPC's kunnen aan een sequentiële taal worden toegevoegd in de vorm van system calls. Het is ook mogelijk RPC-primitieven in een sequentiële taal te integreren. De problemen van pointer- en globale variabelen in een gedistribueerde omgeving worden niet opgelost. In § 3.4.3 wordt een voorbeeld uit de literatuur uitgebreider beschreven.

Het is ook mogelijk om een nieuwe gedistribueerde taal te ontwerpen die RPC's voor de communicatie gebruikt. Veel talen die atomaire transacties implementeren gebruiken RPC's, bijvoorbeeld Argus [Liskov].

3.4.2 Specifieke RPC criteria

Na deze globale schets van RPC-systemen volgt hier een op [Coulouris, Bal, Birrell, Weihl 1] gebaseerde beschrijving van criteria voor de beoordeling van RPC-implementaties. Het zijn achtereenvolgens: globale variabelen, pointervariabelen, parameters, faal-semantiek, foutcode/exceptions, integratie in programmeertaal, interfacedefinitie, gegevensoverdrachtprotocol, netwerkadressering, beveiliging, en parallellisme. Deze criteria zijn van toepassing op het RPC-subsysteem van een op RPC gebaseerde gedistribueerde applicatieontwikkelomgeving. Voor de beoordeling van de gedistribueerde omgeving *als geheel* zijn de criteria uit § 3.3 en § 3.3.5.

a. globale variabelen

In sequentiële, gecentraliseerde systemen zijn applicaties gestructureerd als monolitische programma's. Zo'n programma vormt één proces. Elk proces heeft een adresruimte. In deze ene adresruimte bevinden zich alle procedures en alle variabelen van de monolitische applicatie. Doordat de procedures een gemeenschappelijke adresruimte hebben kunnen ze met behulp van globale variabelen onderling communiceren.

In § 2.3.1 en § 2.3.2 is een gedistribueerd systeem gedefinieerd als een systeem waarin de processoren en processen niet via gemeenschappelijk geheugen met elkaar communiceren. In een RPC-systeem kunnen remote procedures geen gebruik maken van globale variabelen. Wanneer ze beide toegang moeten hebben tot dezelfde gegevens zal via parameteroverdracht steeds de gewenste variabele expliciet doorgegeven moeten worden. Op dit punt wijken remote procedures af van lokale procedures.

Indien men een in een conventionele sequentiële taal als C of FORTRAN geschreven programma naar een gedistribueerde omgeving wil overzetten zal elk gebruik van globale variabelen tussen procedures in verschillende processen door parameteroverdracht moeten worden vervangen. Veelvuldige overdracht van grote hoeveelheden gegevens—fine-grain parallellisme—zoals matrices zal het programma mogelijk te traag maken. Bij het partitioneren van de modules en procedures tijdens het systeemontwerp vormt dit een extra randvoorwaarde.

Wanneer men een sequentiële taal met system-calls gebruikt onderkent de compiler het foutief gebruik van globale variabelen niet. Op pagina 53 is gewezen op het belang van het voorkomen van programmeerfouten bij gedistribueerde applicatie-ontwikkeling. Vanuit dit oogpunt is het beter een omgeving te gebruiken die 'interproces globale variabelen' kan onderkennen.

b. pointervariabelen

Wanneer men geheugenadressen tussen procedures in verschillende processen overzendt moet bij elk gebruik van dat adres de context ervan in het oog worden gehouden. Wanneer proces A de inhoud van een geheugenlocatie van geheugenruimte B wil gebruiken zou het adres van A naar B gezonden moeten worden en vervolgens de inhoud van B naar A. Vanwege de communicatieoverhead leidt dit tot onacceptabele prestaties.[13] Pointervariabelen hebben in RPC-systemen geen betekenis buiten hun context. Ter voorkoming van programmeerfouten zou verhinderd moeten worden dat ze worden overgezonden. Mocht de compiler van de taal dit niet doen dan zal het RPC-subsysteem van het operating system dit vaak kunnen ondervangen.

In veel procedurele talen worden recursieve datastructuren—gelinkte lijsten, bomen—met behulp van pointers gerepresenteerd. Deze kunnen niet direct worden overgezonden. Het RPC-systeem voor zo'n taal moet zulke datatypen herkennen en converteren. Bij dit converteren wordt de datastructuur afgelopen en worden de waarden achter elkaar gezet. Vervolgens worden type en waarden overgezonden. De ontvanger moet de datastructuur in zijn eigen adresruimte weer opbouwen. Het gebruik van zulke samengestelde datastructuren als parameter zorgt voor meer overhead dan enkelvoudige variabelen.

De semantiek van remote procedures is op het gebied van variabelen duidelijk verschillend van lokale procedures. Het nastreven van transparantie levert door de communicatieoverhead te trage systemen op [Bal, Coulouris, Tanenbaum 3].

c. parameters

Een essentieel onderdeel van de procedureabstractie is de wijze van parameter overdracht. In [Watt 1] worden twee parametermechanismen onderscheiden. Een taal kan het *copy mechanism* en het *definitional mechanism* voor parameteroverdacht gebruiken. Met een copy mechanism worden waarden in en uit een abstractie gekopieerd. Men kan value, result en value/result parameters onderscheiden. Dit komt overeen met de input, output en input/output parameters van Ada. Bij het copy mechanism is de formele parameter een lokale variabele. Bij binnenkomst en/of verlaten van de procedureabstractie wordt de waarde van deze lokale variabele van/naar de actuele parameter gekopieerd.

Bij het definitional mechanism wordt bij aanroep van de procedure een link tussen de formele[14] en de actuele parameter gelegd. De formele parameter is geen aparte lokale variabele. Elke wijziging van de formele parameter

[13] Een systeem dat deze vorm van variabeletoegang mogelijk maakt is een logisch gemeenschappelijk geheugen systeem (pagina 45). Vanwege de communicatieoverhead stellen logical shared-memory systemen dusdanige semantische beperkingen aan het gebruik van gemeenschappelijke variabelen dat ze niet als gemeenschappelijk geheugen beschouwd kunnen worden.

[14] De parameter die de aanroeper meegeeft is de *actuele* parameter; de parameter in de definitie van de aangeroepene de *formele* parameter [Van der Sluis].

vindt plaats in de actuele parameter. De formele parameter wordt voor de duur van de aanroep gesubstitueerd door de actuele parameter. Reference parameters zijn een voorbeeld van dit mechanisme.

Het voordeel van het copy mechanism is een eenvoudige semantiek. Een nadeel is dat het op toekenning van waarden aan variabelen (assignment) gebaseerd is. Het is ongeschikt voor variabeletypen zonder toekenning.[15] Een ander nadeel is dat het kopieëren van grote samengestelde waarden (arrays) inefficiënt is.

Het voordeel van het definitional mechanism is dat het geschikt is voor alle variabeletypen. Het is gebaseerd op indirecte toegang tot waarden—via pointers of adressen van variabelen. Dit is in het algemeen efficiënter dan het kopieëren van gegevens. Een nadeel is de mogelijkheid van aliassen. In het voorbeeld is `y` een alias van `x` op het moment dat beide de link naar `a` leggen. Volgens [Watt 1, p. 98] maken aliassen programma's ingewikkelder en slecht beredeneerbaar.

Een referentie naar een variabele is het geheugenadres van die variabele. Daar aanroeper en aangeroepene in gedistribueerde systemen een verschillende adresruimte hebben kan het RPC systeem niet dit geheugenadres doorgeven. In plaats van de referentie moet de waarde worden doorgegeven. Bij de simulatie van call by reference moet deze waarde zowel heen als terug gekopieerd worden. Dit wordt *call by copy/restore* genoemd. In bepaalde situaties is de semantiek van call by copy/restore anders dan bij call by reference. In [Tanenbaum 3] wordt hiervan het volgende voorbeeld gegeven.

```
program test(output);
var a: integer;

procedure dubbelverhoog(var x, y: integer);
begin
   x := x + 1;
   y := y + 1;
end;

begin
   a := 0;
   dubbelverhoog(a, a);
   writeln(a);
end.
```

Als `dubbelverhoog` een lokale procedure is refereren `x` en `y` beide aan `a`. `a` wordt dan achtereenvolgens van 0 naar 1 en van 1 naar 2 opgehoogd.

Als `dubbelverhoog` als RPC wordt behandeld dan zal het RPC-subsysteem bij aanroep de waarden van de twee call by copy/restore parameters,

[15] In Pascal mogen filevariabelen niet aan elkaar worden toegekend vanwege de overhead van het kopiëren van files. Dit moet men met lees- en schrijfopdrachten per record uitvoeren.

tweemaal 0, verzenden aan de remote procedure. Deze hoogt de eerste parameter op van 0 naar 1, en hij hoogt y op van 0 naar 1. De waarden worden teruggezonden en a blijkt 1 te bevatten.

De semantiek van call by reference wordt slechts benaderd met call by copy/restore. In de meeste gevallen zal deze benadering voldoende zijn. In de uitzonderlijke gevallen zullen dit soort fouten moelijk te vinden zijn.

In gedistribueerde omgevingen zijn geheugenreferenties (pointers) niet efficiënt. Veel RPC-implementaties vermijden dit soort problemen door pointers en referenceparameters niet te ondersteunen. Applicatieprogrammeurs moeten hiervoor zelf oplossingen programmeren. De semantiek van een RPC ligt dan wel erg ver van die van een lokale procedure aanroep.

d. faal-semantiek

Bij het ontwerpen van computersystemen speelt de mate van fouttolerantie die het systeem gaat ondersteunen een belangrijke rol. In het kader van RPC-systemen spitst deze kwestie zich toe op „de exacte betekenis van een aanroep in het licht van het falen van machine en communicatie" [Birrell].

De faal-semantiek van een *lokale* aanroep valt in twee categorieën uiteen: in systemen *met* en systemen *zonder* transactiemechanismen. Er zijn systemen die met behulp van transactiemechanismen fouttolerantie bieden [Date]. De toestand van de applicatie wordt op foutvrij achtergrondgeheugen (*stable storage*) bewaard. Wanneer een procedure halverwege de uitvoering onderbroken wordt kan bij herstarten van het systeem de situatie van voor de onderbreking gereconstrueerd worden, waarna de procedure of het proces de operaties alsnog gaat uitvoeren.

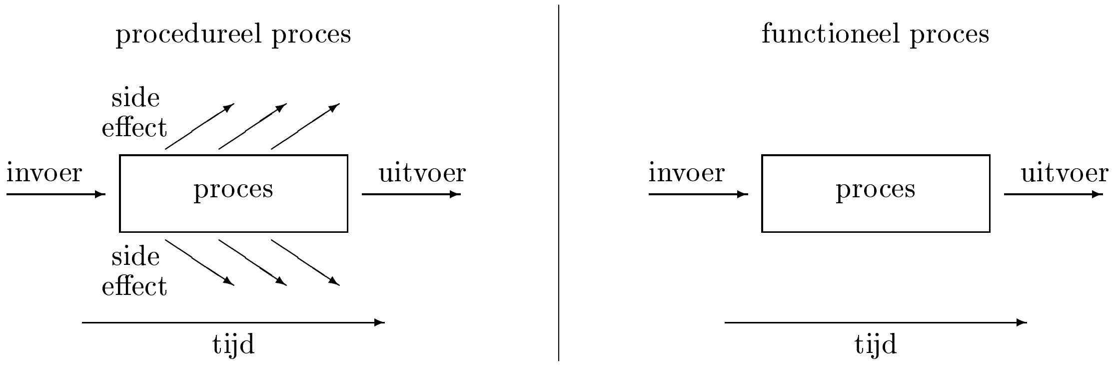
Een sleutelbegrip in deze problematiek is het zogenaamde *side-effect*. Een *effect* is een toestandsverandering van een virtuele machine of *omgeving*. *Side-effects* zijn toestandsveranderingen die naast het einddoel van de procedure plaatsvinden. Zie figuur 3.2. Side-effects zijn gebeurtenissen die, voordat de procedure is afgelopen, reeds de toestand van het systeem veranderen.

De oorspronkelijke betekenis van een procedure—of een proces—is een entiteit die invoer accepteert, bewerkingen erop uitvoert, en uitvoer produceert. Deze strikte betekenis wordt naar analogie van het wiskundige functiebegrip van functionele programmeertalen in [Van Renesse] het *functional processing model* genoemd. Een functioneel proces heeft geen interactie met zijn omgeving anders dan door zijn uitvoer. Een functioneel proces is deterministisch,[16] de uitvoer wordt volledig door de invoer bepaald.

Na een storing wordt het proces door de omgeving opnieuw opgestart met dezelfde invoer. Het systeem garandeert zo dezelfde uitkomst. De uitvoer kan uiteindelijk leiden tot 'side'-effects als bestandsbijwerkingen of creatie van een ander functioneel proces. Communicatie vindt plaats doordat een proces de uitvoer van een ander proces als zijn invoer kan accepteren.

[16] Volgens het woordenboek: Leer volgens welke de handelingen van een mens afhangen van voorgaande zaken, waarop hij zelf geen invloed heeft.

Figuur 3.2: Side-effects



In conventionele, procedurele talen kan een proces wèl side-effects hebben. Omgevingen die fouttolerantie moeten ondersteunen moeten tot de terminering van een proces alle side-effects bijhouden om in het geval van een calamiteit de oorspronkelijke toestand van het systeem te kunnen reconstrueren.

Zero-or-one semantiek houdt in dat àlle toestandsveranderingen—ook de side-effects—in hun geheel optreden. Òf àlle toestandsveranderingen vinden plaats, òf geen enkele. De toestandsveranderingen die optreden totdat een procedure halverwege onderbroken wordt, worden bij zero-or-one semantiek achteraf ongedaan gemaakt. Pas als de laatste operatie van de procedure heeft plaatsgevonden wordt de gehele toestandsverandering doorgevoerd. De procedure (transactie) wordt dan ge-'commit'. Deze acties vinden transparant voor de applicatieprogrammeur plaats. De herstelwerkzaamheden worden door de virtuele machine uitgevoerd.

Procedures zonder side-effects zijn *idempotent*: wanneer ze worden aangeroepen met dezelfde parameters geven ze steeds hetzelfde resultaat. Een procedure met uitvoerparameter i die $i := 2 + 2$ berekent is idempotent. Een procedure die $i := j + 1$ uitrekent waarbij j geen parameter is, niet. De uitkomst hangt niet van de invoerparameters af maar van andere omgevingsfactoren—in dit geval van globale variabele j.

Systemen die geen transactieconcept ondersteunen bieden *last-one* semantiek. Wanneer de processor faalt wordt bij herstarten van de applicatie uiteindelijk de onderbroken procedure opnieuw uitgevoerd. Eventuele side-effects treden wellicht opnieuw op. De aanroeper gebruikt het eindresultaat van de laatste procedureaanroep. Voor het uitvoeren van idempotente berekeningen is last-one semantiek voldoende. Systemen die last-one semantiek bieden missen de overhead van transactiesystemen. Ze zijn sneller (pagina 51).

RPC-systemen kunnen verschillende strategieën toepassen bij het optreden van systeemfouten. De literatuur onderscheidt twee categorieën fouten:

communicatiefouten en processorfouten [Bal]. Het optreden van *communicatiefouten* wordt bij connection-oriented communicatieprotocollen (als TCP en OSI) door de netwerksoftwarelaag gesignaleerd. Het uitvallen van een *processor* kan door het RPC-subsysteem worden gesignaleerd.

Wanneer het RPC-systeem een fout signaleert kan het de fout rapporteren aan de aanroeper, of proberen zelf de fout te herstellen. Bij idempotente procedures kan het RPC-systeem de aanroep blijven herhalen tot er een antwoord komt. De procedure wordt dan minstens een keer uitgevoerd. Dit heet *at-least-once* semantiek.

Wanneer de procedure side-effects heeft leidt deze werkwijze tot ongewenste resultaten. Een oplossing is om zero-or-one semantiek via fouttransparantie (§ 3.3.3) aan te bieden. Het is ook mogelijk om de programmeur bouwstenen aan te bieden (atomaire objecten met transacties) waarmee men zelf zero-or-one semantiek kan programmeren.

Een derde mogelijkheid is om een foutcode aan de aanroeper terug te geven of een exception te signaleren. De programmeur moet dan zelf zorgen voor code die de fout afhandelt. In dit laatste geval biedt de RPC-implementatie *at-most-once* semantiek. Het RPC-systeem (de virtuele RPC machine) garandeert dat bij een foutmelding de aangeroepen procedure maximaal één keer is uitgevoerd. Side-effects kunnen maximaal één keer optreden. Als alleen het antwoord verloren is gegaan is de procedure verder in zijn geheel uitgevoerd. Bij een processorfout halverwege is de procedure slechts ten dele uitgevoerd. Het RPC-systeem onderscheidt deze twee mogelijkheden niet en voert geen herstelwerkzaamheden uit. Dit wordt aan de programmeur overgelaten.

In de tabel wordt de meest geschikte vorm van semantiek voor verschillende applicaties en aanroepen samengevat. Er zijn idempotente toepassingen en toepassingen met side-effects. Een procedureaanroep kan lokaal of remote zijn. Voor elk van de vier combinaties is de semantiek die de level 5 virtuele machine biedt gegeven.

Semantiek

	lokale aanroep	*RPC*
idempotente procedure	last one	at least once
procedure met side-effects	zero or one	at most once *of* zero or one

At-least-once semantiek, het opnieuw verzenden van een aanroep door het RPC-systeem, is alleen geschikt voor idempotente berekeningen. At-most-once semantiek biedt de programmeur weinig ondersteuning. Voor toepassingen die een beperkte mate van bedrijfszekerheid vereisen is at-most-once semantiek geschikt. Het RPC-systeem is eenvoudiger en aanmerkelijk sneller. Eventuele fouttolerantie wordt moet programmeur geheel zelf programmeren (§ 3.3.3). De semantiek van een at-most-once procedureaanroep wijkt af van die van een lokale procedureaanroep.

Voor toepassingen die een zekere mate van bedrijfszekerheid vereisen zijn zero-or-one RPC-systemen beter geschikt.

e. foutcode/exceptions

Het optreden van fouten kan op verschillende manieren worden gesignaleerd. Sommige talen, bijvoorbeeld Ada, stellen de programmeur in staat voor foutcondities die het run-time systeem van de taal signaleert exception handlers te schrijven. Bij het optreden van een fout wordt de code van de desbetreffende exceptionhandler uitgevoerd. Dit gebeurt los van het gewone programma. In het gewone programma wordt de code van het algoritme niet meer onderbroken door code voor het testen op foutcondities.

Sommige talen verwachten van de programmeur dat deze na elke procedureaanroep op zulke foutcondities test. C is hiervan een goed voorbeeld [Kernighan 2]. Veel fouten die pas na de invoering van een programma aan het licht komen zijn te wijten aan het vergeten van dit soort fouttesten [Rochkind]. In talen die exceptions ondersteunen is het niet nodig op elke plaats op een fout te testen. Voor elke soort fout hoeft bij deze talen maar één keer de afhandelingscode geschreven te worden [Coulouris]. De aanroep van deze code wordt door het run-time systeem van de taal verzorgd.

Het opsporen van fouten bij gedistribueerd programmeren is van groot belang (zie ook pagina 53). Het schrijven van foutafhandelingscode is in talen die exceptions implementeren minder werk [Coulouris, Bal].

In RPC-systemen die zero-or-one semantiek ondersteunen worden fouten van lagere virtuele machines door het RPC-subsysteem afgehandeld. In at-most-once systemen kunnen meer soorten fouten voorkomen. Hier is een adequate foutafhandeling heel belangrijk. Van de programmeur wordt hier veel discipline verwacht. Talen die exceptions ondersteunen geven hiervoor een zekere ondersteuning.

f. integratie in programmeertaal

Het doel van het RPC concept is gedistribueerd programmeren te vereenvoudigen door een bekende abstractie uit procedurele talen te implementeren [Birrell]. RPC-primitieven worden vaak aan reeds bestaande sequentiële procedurele talen toegevoegd. De semantiek van de RPC moet dan zo veel mogelijk overeenkomen met concepten uit die taal.

RPC-primitieven kunnen als system-calls aan de taal worden toegevoegd. Deze system-calls zijn moeilijk in het gebruik [Birrell, Sun 1]. Om het schrijven van toepassingsprogramma's voor deze omgevingen te vereenvoudigen wordt soms een codegenerator aan de omgeving toegevoegd. Deze codegenerator produceert op basis van een *interfacedefinitie* (zie punt g) code die de programmeur anders zelf zou moeten schrijven. Met de codegenerator probeert men de programmeur zo veel mogelijk van ingewikkelde system-calls af te schermen.

Hiervoor zijn de problemen van RPC's met betrekking tot globale variabelen, pointers, parameters en faal-semantiek beschreven. Deze problemen komen voort uit de wens om een gedistribueerd concept een conventioneel (gecentraliseerd) uiterlijk te geven.

RPC's worden behalve in sequentiële talen ook in gedistribueerde talen gebruikt. Bij het ontwerp van zo'n gedistribueerde taal kiest men alleen concepten die voor een gedistribueerde omgeving geschikt zijn, zodat er geen integratieproblemen met concepten voor verschillende architecturen zijn (zie § 3.2.1).

In Argus, een taal die atomaire transacties ondersteunt, worden RPC's anders gezien dan Birrell & Nelson [Birrell] dat doen. Argus is niet ontworpen met als primair doel sequentiële procedureabstracties te implementeren. [Liskov, p. 387] stelt, dat een RPC-systeem de message-passing details van de programmeur moet afschermen, maar de mogelijkheid van langere wachttijden en processorfouten *niet* voor de aanroeper moet verbergen. De aanroeper moet fouten afhankelijk van de eisen van de applicatie kunnen afhandelen. De RPC moet gestopt kunnen worden en in dat geval geen effecten hebben (zero-or-one semantiek). De mogelijkheid dat clients RPC's kunnen stoppen heeft gevolgen voor het ontwerp van de server. De server moet in staat zijn de toestand van het systeem van voor de RPC te restaureren.

Door een andere semantiek aan de (sequentiële) procedureabstractie bij RPC's toe te kennen, kan men een taal te maken die een grotere conceptuele consistentie biedt dan een taal met sequentiële èn gedistribueerde concepten. De transparantie van zo'n taal is door deze semantiekverandering verminderd.

Birrell & Nelson streven dezelfde semantiek na (transparantie), al bereiken ze die niet. Liskov streeft niet dezelfde semantiek na. Zij legt meer de nadruk op flexibiliteit door de programmeur expliciete primitieven voor foutafhandeling te geven.

g. interfacedefinitie

Veel moderne imperatieve talen als Ada, Modula-2 en sommige Pascal dialecten kennen het begrip *module*. Een module wordt ook wel 'package' of 'unit' genoemd. Een module is een hogere vorm van abstractie dan de procedureabstractie in de zin dat een moduleabstractie meerdere procedureabstracties omvat. In figuur 2.3 is de structuur van een programma weergegeven als bestaande uit meerdere modules die elk weer uit meerdere procedures bestaan.

Net als de procedure kent de module een interfacebeschrijving. Bij de procedure is het de declaratie waarin naam en parameters beschreven worden. Een module interface bevat de componenten (procedures en variabelen) die zichtbaar zijn voor de buitenwereld. Door de scheiding van interfacebeschrijving en implementatie van de componenten biedt de module de mogelijkheid voor zijn gebruikers om van de implementatiedetails te abstraheren. Zo ondersteunt de programmeertaal een vergaande structurering van de opbouw van een programma.

Modules kunnen apart gecompileerd worden. De compiler kan met behulp van de interfacebeschrijving nagaan of er geen inconsistentie tussen gebruik en implementatie van een module bestaat. De voor de compilatievolgorde van de modules onderlinge afhankelijkheden worden door de compiler aan de hand van de interfacebeschrijvingen bijgehouden.

In tegenstelling tot deze *aparte* compilatie kennen C en FORTRAN *gescheiden* compilatie. Hier worden procedureaanroepen niet op type-inconsistenties gecontroleerd. De compilatievolgorde wordt niet door de compiler bijgehouden [Watt 1].

Door in ANSI C extra proceduredeclaraties vóór de procedureaanroep te vermelden wordt de compiler in staat gesteld toch typechecks uit te voeren op de parameters van een procedure. Deze voorafgaande proceduredeclaraties worden in ANSI C 'prototypes' genoemd. Ze zijn te vergelijken met forward declaraties van procedures in Pascal.

Client/server applicaties bestaan uit clientmodules en servermodules. Wanneer de communicatie tussen deze modules met RPC's wordt uitgevoerd moet er een mechanisme bestaan waarmee consistentie tussen procedureaanroep en -definitie—bij gescheiden compilatie—kan worden gewaarborgd. RPC-systemen bevatten voor dit doel vaak een speciale interfacetaal. Als het RPC-subsysteem geïntegreerd is in een gedistribueerde taal is dit niet nodig.

Met een interfacetaal worden de interfaces van remote procedures gespecificeerd. In interfacetalen kunnen de naam van een procedure, de datatypen van zijn parameters en de 'richting' van deze parameters—input of output—worden beschreven.

Deze interface wordt door een interfacecompiler naar een module in de gewenste programmeertaal (C, Pascal, FORTRAN, etc.) vertaald. Deze module maakt typechecks van de compiler mogelijk en verzorgt details van het RPC mechanisme die niet door de bibliotheekroutines worden afgehandeld als *marshalling* en *dispatching* (zie pagina 80).

De belangrijkste taak van de interfacedefinitie is het mogelijk maken van typechecks van aanroep en definitie van remote procedures. Op deze wijze wordt een stap naar typesecurity gezet (pagina 53).

h. gegevensoverdrachtprotocol

Een gedistribueerde omgeving kan uit systemen van verschillende leveranciers bestaan. De respectievelijke virtuele machines kunnen gegevens elk op hun eigen manier representeren. Het is ook mogelijk dat *complexe datastructuren* in verschillende omgevingen (programmeertaal, DBMS) anders worden opgeslagen [Tanenbaum 3]. Om toch gegevensuitwisseling mogelijk te maken moeten de partijen, de virtuele RPC machines, een gemeenschappelijke representatiestandaard met elkaar overeen komen.

In het OSI-referentiemodel worden deze problemen in de presentatielaag, laag 6, opgelost. Eventuele conversie tussen machines die gegevens op een andere wijze intern representeren vindt hier plaats. Wanneer het RPC-systeem op protocollen volgens de OSI-structuur gebaseerd is levert een verschillende representatie voor het RPC-systeem geen problemen op. De conversie wordt dan door de protocollen gedaan.

Eén van de voornaamste ontwerpeisen die aan RPC-implementaties gesteld worden is snelheid. Een lokale procedureaanroep is in vergelijking met netwerkcommunicatie snel. Het OSI-referentiemodel is uit zeven lagen opgebouwd. Een implementatie met zo'n sterk gelaagde opbouw levert voor RPC-toepassingen teveel overhead op. RPC-implementaties worden vanwege de snelheid zo min mogelijk gelaagd gestructureerd [Tanenbaum 3].

Als de gebruikte netwerkprotocollen hier niet in voorzien moet het RPC-subsysteem zelf voor conversie zorg dragen wanneer men applicaties in een heterogene omgeving wil kunnen gebruiken.

Binnen netwerkprotocollen kan men twee groepen onderscheiden op basis van de functionaliteit die de gebruiker geboden wordt: *connection oriented* protocollen en *connection-less* (of datagram) protocollen [Tanenbaum 3]. Connection-less communicatie bestaat uit enkelvoudige netwerkpackets die niet noodzakelijk op volgorde worden afgeleverd. Connection oriented communicatie garandeert dat verzonden berichten in volgorde van verzending aankomen. Het implementeren van connection oriented protocollen is ingewikkelder dan het implementeren van connection-less protocollen. De implementatie van connection-less functionaliteit kent minder overhead en is sneller.

RPC-systemen kunnen *at-least-once* semantiek implementeren met datagramcommunicatie door na het verstrijken van een bepaalde tijdsduur een verzoek opnieuw te verzenden. Bij *at-most-once* semantiek moet de implementatie het optreden van fouten eenduidig kunnen vast stellen. In plaats van hier zelf mechanismen op basis van datagramcommunicatie voor te bouwen is het efficiënter voor ontwerpers van RPC-systemen om betrouwbare (connection oriented) protocollen te gebruiken.[17] Deze protocollen signaleren zelf al het uitvallen van een verbinding. Ook voor zero-or-one semantiek zijn connection oriented protocollen beter geschikt dan datagramprotocollen.

Uiteindelijk wordt de protocolkeuze bepaald door het soort applicatie dat men programmeert. Idempotente procedures zijn sneller met een datagramprotocol. Voor procedures met side-effects zijn connection oriented protocollen beter geschikt [Coulouris, Sun 1].

i. netwerkadressering

In een gedistribueerd systeem bevinden de processen van een applicatie zich op verschillende virtuele machines. In een RPC applicatie moet de aanroeper op de een of andere wijze aangeven *welke* remote procedure hij aanroept. Het proces dat de remote procedure bevat moet aan het systeem bekend maken welke diensten het aanbiedt. Dit kan in beginsel at run-time en at compile-time. Compile-time *binding* is te inflexibel. Bij elke verandering van de gebruikte netwerkconfiguratie zou de applicatie gewijzigd en gecompileerd moeten worden.

[17] Birrell & Nelson [Birrell] hebben dit niet gedaan. Zij implementeerden at-most-once semantiek op basis van datagramverkeer met zelf geschreven sterk geoptimaliseerde RPC communicatie protocollen.

Bij binding at run-time worden bij het compileren van de applicatie bepaalde identifiers (namen of nummers) aan aanroeper en aangeroepene toegekend. De plaats—het netwerkadres—waar de delen van de applicatie zich zullen gaan bevinden hoeft op dit moment nog niet vast te staan. Pas bij de daadwerkelijke aanroep, at run-time, wordt het netwerkadres vastgesteld. Op deze manier is het mogelijk dat servers van plaats veranderen zonder dat de applicatie opnieuw gecompileerd behoeft te worden.

Zo gauw een serverproces is opgestart maakt het zich bekend bij een centraal RPC meldpunt. Bij een aanroep zoekt het RPC-systeem bij dit centrale meldpunt uit op welk netwerkadres de gevraagde procedure zich bevindt. Zodra het netwerkadres is vastgesteld gaat de communicatie direct—zonder hulp van het centrale meldpunt. Zo wordt een flexibele en efficiënte oplossing geboden voor de adresseringsproblematiek [Birrell].

j. beveiliging

Voor het beveiligen van gedistribueerde systemen wordt gebruik gemaakt van twee methodes:. *versleuteling* van berichten en *identificatie* van vragers van een bepaalde dienst. Om ongeauthoriseerd gebruik van een remote procedure te voorkomen worden beide technieken toegepast. Bij elke aanroep moet worden vastgesteld of de aanroeper de procedure mag uitvoeren.

In gecentraliseerde systemen worden gebruikers met behulp van een centraal wachtwoordbestand geïdentificeerd. De gebruiksrechten zijn op één plek opgeslagen. In gedistribueerde systemen worden gebruiksrechten vaak op een meer gedistribueerde manier bijgehouden. Processen kunnen zelf een soort sleutel krijgen die een gebruiksrecht vertegenwoordigd. Door deze sleutel—of 'capability'—met een verzoek aan een ander proces mee te zenden kan deze direct vast stellen of het verzoek gehonoreerd mag worden. Dit voorkomt dat de server eerst zelf de beheerder van het wachtwoordenbestand moet raadplegen bij een aanroep van een client. Een alternatief is dat de servers zelf bijhouden wie het recht tot gebruik heeft met behulp van zogenaamde 'access control lists' [Mullender 2].

Omdat servers bij het gebruik van capabilities minder toestandsinformatie over hun clients behoeven bij te houden wordt deze methode in veel gedistribueerde systemen toegepast [Coulouris].

Door het afluisteren van communicatielijnen zouden derden gegevens of wachtwoorden en capabilities in handen kunnen krijgen. Om dit te voorkomen kan de communicatie versleuteld worden. Er bestaan veel cryptografische technieken. In [Tanenbaum 3] worden er een aantal beschreven. De Data Encryption Standard (DES) van de Amerikaanse overheid wordt veel toegepast. Om een snelle versleuteling te bereiken is DES in hardware is geïmplementeerd.

Voor een adequate beveiliging van gedistribueerde systemen is de wijze waarop gebruikers omgaan met de mogelijkheden die het systeem hiervoor biedt van groot belang.

k. parallellisme

Communicatie in RPC-systemen verloopt synchroon. Nadat de aanroeper een RPC heeft uitgevoerd wacht hij op antwoord. De mate van parallellisme van een applicatie is bij synchrone communicatie minder dan bij asynchrone communicatie. Bij asynchrone communicatie gaan client en server beide door met de uitvoering van hun proces.

Wanneer de *client* op antwoord wacht kan het operating system andere processen schedulen. Aan de client kant kan de processor steeds nuttig werk verrichten door verschillende processen pseudo-parallel uit te voeren. Aan de *server* kant kunnen zich meer problemen voordoen. Veel gedistribueerde applicaties worden zodanig gestructureerd dat voor bepaalde diensten specifieke servers worden aangewezen (zie ook § 3.4.4). Deze servers kunnen met verzoeken van meerdere clients tegelijk te maken krijgen.

Als er een RPC aankomt bij een server terwijl een vorige RPC nog niet is afgelopen kan het RPC-systeem verschillende wegen bewandelen. Verzoeken kunnen in FIFO volgorde worden afgehandeld. Het nadeel hiervan is dat er lange wachttijden kunnen ontstaan voor nieuwe clients. Daarnaast moeten applicatieontwerpers met deadlock rekening houden. Wanneer server A een RPC doet naar een andere serverprocedure die als client net een RPC naar A heeft gedaan treedt er deadlock op. Vaak zal een deadlock door een timeout van het RPC-systeem worden opgeheven. Om inefficiënte verwerking te voorkomen is een zorgvuldig ontwerp van applicaties noodzakelijk.

Voor ontwerpers van RPC-systemen zijn FIFO servers eenvoudig te ontwerpen. Om tot een betere respons van het systeem te komen kan het operating system ook voor elke RPC een nieuw serverproces opstarten. Veel operating systems kennen alleen heavy-weight processen. Voor het opstarten en pseudo-parallel uitvoeren van heavy-weight processen moet het operating system steeds een contextswitch uitvoeren. Dit is inefficiënt [Coulouris].

Wanneer de RPC een omvangrijke procedure betreft zou het nieuwe proces naar een andere processor verplaatst kunnen worden. Om zo'n beslissing te kunnen nemen moet het operating system weten of de tijd, die het verplaatsen van een proces kost, opweegt tegen de verkorting van de responstijd. In § 3.4.3 wordt op pagina 73 nog een variant beschreven waarbij meerdere inactieve processen stand-by worden gehouden.

Wanneer het operating system light-weight processen ondersteunt kan voor elke RPC ook een nieuwe thread-of-control binnen het al bestaande serverproces worden opgestart. De overhead van een contextswitch ontbreekt hier. Voor serverprocedures die zwaar belast zullen gaan worden kan de applicatieprogrammeur ook hier replicatie toepassen.

Volgens [Coulouris] leveren servers die op light-weight processen gebaseerd zijn duidelijk betere prestaties dan servers in heavy-weight proces operating systems als UNIX.

3.4.3 Structuur van een implementatie

Hiervoor zijn verschillende aspecten die bij het ontwerp van RPC implementaties een rol spelen beschreven. Om de manier waarop deze aspecten samenhangen bij de implementatie van een compleet RPC systeem duidelijk te maken wordt in deze paragraaf de structuur van het RPC-systeem dat Birrell & Nelson [Birrell] hebben ontworpen beschreven.

Het RPC systeem maakt onderdeel uit van het Cedar gedistribueerde operating system, dat in het Xerox PARC is ontwikkeld. Dit operating system kent een RPC-subsysteem dat via system-calls RPC en andere GOS-primitieven aanbiedt. De level 5 probleemgeoriënteerde talen die het systeem ondersteunt zijn BCPL, Interlisp, Smalltalk, C en Mesa. Voor deze talen is een interfacecompiler, Lupine, beschikbaar.

interface

Voor applicaties moet een interfacemodule geschreven worden. Deze bestaat uit een lijst procedurenamen met hun parameter- en resultaattypen. Op basis van deze informatie kunnen compile-time typechecks (gescheiden) voor client en server worden uitgevoerd. Een module die procedures implementeert *exporteert* de interface. Een module die procedures aanroept *importeert* de interface.

Op basis van de interfacemodule genereert Lupine hulpprocedures die 'stubs' worden genoemd. Een stub is een procedure die op de lokale machine de contactpersoon (agent) is van de eigenlijke remote procedure of aanroeper (zie figuur 3.3). De client roept de stub aan als een gewoon lokale procedure. Het *is* een gewoon lokale procedure. De stub regelt details die met het netwerk te maken hebben. Dit zijn zaken als het verzendklaar maken van parameters en het aanroepen van de system calls voor het uitvoeren van de RPC.

De stub aan de server kant wordt na ontvangst van de boodschap door het operating system aangeroepen en roept op zijn plaats de door de applicatieprogrammeur geschreven functie aan.

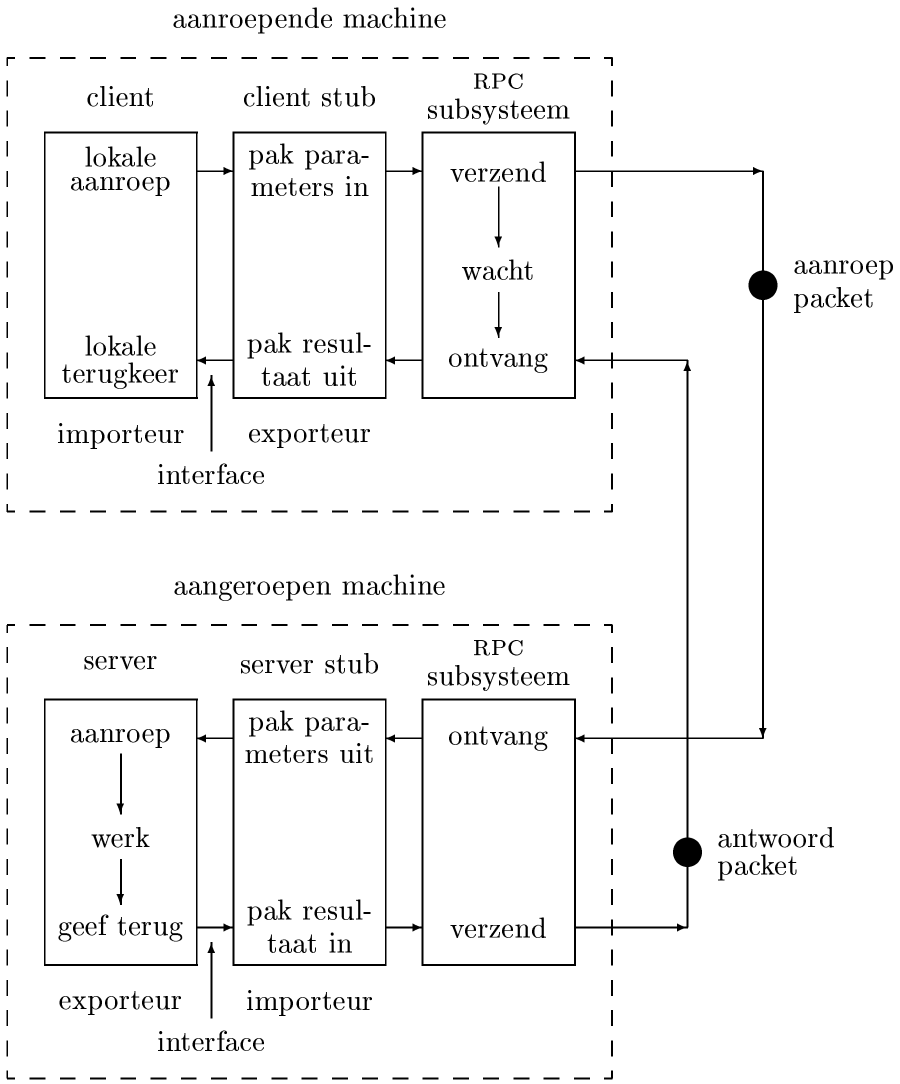
Op deze wijze behoeven programmeurs geen gedetailleerde communicatiecode te schrijven. Ze moeten alleen interface-, client- en servermodules schrijven. Lupine zorgt voor het genereren van code voor het in- en uitpakken van parameters (marshalling) en het aanroepen (dispatchen) van de juiste procedure voor een binnengekomen aanroep (zie pagina 80). Lupine gaat ook na of de programmeur geen parameters gebruikt die van een gemeenschappelijke adresruimte uitgaan.

Het Cedar RPC systeem gebruikt de Grapevine gedistribueerde database [Coulouris] als centraal meldpunt voor netwerkadresbepaling van remote procedures.

communicatieprotocol

Als communicatieprotocol is een speciale vorm van datagramcommunicatie toegepast. Het RPC-subsysteem implementeert 'light-weight connection

Figuur 3.3: Stubs [Birrell]



management' met als primair doel een snelle communicatie te waarborgen. De lichtgewicht verbinding wordt bereikt door bij een snelle opeenvolging van RPC's een volgend RPC-bericht als acknowledge van de vorige op te vatten. Op deze manier is het niet nodig dat voor elke boodschap een bevestiging van goede ontvangst wordt verstuurd.

In dit protocol wordt een verbinding (connection) gezien als de gemeenschappelijke toestand van de aanroeper die een RPC doet en de server die deze aanroep accepteert. Het RPC-subsysteem gaat om de hogere communicatieprotocollagen van het operating system heen om een zo snel mogelijke respons te verwerkelijken.

efficiëntie

De taal Mesa ondersteunt exceptions en parallelle processen. Deze processen zijn heavy-weight. Om dure contextswitches te voorkomen wordt op elke machine een aantal serverprocessen door het RPC-subsysteem stand-by gehouden. Een RPC kan zo worden uitgevoerd zonder dat er telkens een extra proces gecreëerd moet worden. Deze manier van werken is transparant voor de programmeur.

Voor de beveiliging van het systeem biedt het Cedar RPC systeem de Grapevine database voor identificatie doelen en de DES versleutelmethode voor communicatie.

Birrel & Nelson schrijven in [Birrell, p. 42]: „An important issue in design is resolving the tension between powerful semantics and efficiency." Deze afweging speelt niet alleen bij het ontwerp van RPC-systemen een rol, maar is bij het ontwerp van gedistribueerde applicatieontwikkelomgevingen in het algemeen van belang (§ 2.3 en § 3.3). Bij de semantiek van een lokale procedure aanroep hoort, naast de in de vorige paragraaf beschreven punten, ook een snelle respons. Met de interfacecompiler Lupine is getracht het gebruik van een RPC te vereenvoudigen. Met het optimaliseren van communicatieprotocol en serverprocesstructuur is een snelle respons nagestreefd.

3.4.4 Client/server

Een belangrijk model bij applicaties die om reden 3 en 4 van § 2.1.4 op gedistribueerde systemen worden geïmplementeerd is het client/server model. Bij de bespreking van loosely-coupled gedistribueerde systemen wordt dit vaak gebruikt [Birrell, Coulouris]. De beschrijving van gedistribueerde systemen in abstracte zin leent zich goed tot gebruik van de termen 'client' en 'server'.

De client/server benadering houdt in dat een proces een ander proces verzoekt een bepaalde taak voor hem te verrichten. Het kunnen verzoeken zijn als: „Haal alle namen op van saldo's die negatief zijn", of „Zet een zachtrode vijfhoek linksboven op het scherm", of „Open bestand '/usr/mail/berg001' voor schrijven", of „Geef de beste zet die de koningin op e4 kan doen".

De client/server benadering is, dat taken moeten worden toegekend aan processoren die daar het meest geschikt voor zijn. Bijvoorbeeld: een databasemachine is geschikt voor bestandsmanipulatie, en een grafisch werkstation (of een PC) voor interactie met de gebruiker. Het werkstation stuurt als client queries aan de server. De client geeft de antwoorden grafisch weer aan de gebruiker.

Gebruik van een client/server architectuur kan leiden tot:[18]

- Minimaliseren van de netwerkbelasting door efficiënte verdeling van taken.
- Een zo snel mogelijke respons door taken op de meest geschikte processor uit te voeren.
- Efficiënt gebruik van de beschikbare processoren.
- Structurering van programma's (vergelijk de monolitische monitor van § 2.1.1).
- Door parallelle uitvoering betrouwbaarheid of rekensnelheid verhogen.

Het is de bedoeling dat de boodschappen die uitgewisseld worden zo klein mogelijk zijn om zo het netwerk, vaak de traagste component in het systeem, weinig te belasten [Coulouris].

Methoden om tot een goede partitionering te komen zijn bekend uit de systeemanalyse [DeMarco] en -ontwerp [Page-Jones] en software engineering [Van Katwijk]. Hier wordt getracht een partitionering te vinden die de interfaces (gegevensstromen) tussen verschillende subsystemen van een programma minimaliseert. De modules die het resultaat zijn van deze partitionering vormen bij monolitische programma's één proces. Voor grotere applicaties kan deze aanpak tot beheersproblemen aanleiding geven. De complexiteit van de applicatie wordt ondanks de partitionering te groot. Door nu modules van een programma in meerdere autonome eenheden te groeperen—over verschillende processen te verdelen—wordt het partitioneringsprincipe op een hoger niveau nogmaals toegepast. Het programma wordt relatief losstaande eenheden opgesplitst. Deze processen bezitten geen gemeenschappelijke globale variabelen. Ze moeten van expliciete communicatie gebruik maken.

Bij het partitioneren voor client/server applicaties is de bepalende factor het zoeken naar de beste verdeling over mogelijk heterogene processoren.

Client/server partitionering levert modules in verschillende processen op. De interproces-communicatie die hierdoor nodig wordt wordt met behulp van message-passing geïmplementeerd. De client stuurt een boodschap aan een server die een bepaalde service aanbiedt. De server stuurt in veel gevallen een antwoord terug aan de client. `Send`- en `receive`-primitieven zijn in veel gedistribueerde operating systems (level 3) geïmplementeerd. Er is geen standaard voor interproces-communicatie op level 5 niveau, al wordt het RPC paradigma veelvuldig toegepast. De ontwikkeling van multiproces applicaties ondervindt hiervan hinder [Rochkind].

[18] Nadelen van het client/server model zijn in § 3.4.5 beschreven.

3.4.5 Nadelen van het RPC model

Het remote procedure call paradigma wordt in veel gedistribueerde systemen gebruikt. RPC heeft de status van een heilige koe gekregen [Tanenbaum 4]. Het RPC-model geeft een elegant concept om het schrijven van gedistribueerde programma's te vereenvoudigen. Helaas zijn sommige problemen moeizaam met behulp een RPC te implementeren.

Een alternatief voor het *high level* RPC model is het *low level* virtueel circuit model van OSI. Dit streeft geen transparantie na zoals het RPC-model.[19] Het virtueel circuit model bestaat uit asynchrone point-to-point communicatie—`send` en `receive`.

Onder de volgende kopjes worden drie soorten problemen van het RPC-model beschreven.

concept (1)

Het RPC-model gaat uit van twee partijen: de client en de server. De client roept de server aan waarop de server antwoordt. Client en server zijn nooit tegelijkertijd actief. Een RPC is een synchrone communicatievorm. Dit levert voor veel toepassingen problemen op. Vaak is het wenselijk dat bij client of server een bepaalde mate van asynchrone verwerking mogelijk blijft.

Ter illustratie van dit probleem dienen twee voorbeelden. Het eerste is een applicatie die bestaat uit een database-server en een gebruikersinterface-client. Wanneer de gebruiker een vraag geformuleerd heeft pleegt de client een RPC naar de database. De server verwerkt de query en geeft als resultaat de records die de gebruiker gevraagd heeft. De client presenteert deze records aan de gebruiker.

Wanneer de gebruiker een query geformuleerd heeft die veel verwerkingstijd vraagt kan het gebeuren dat de gebruiker zich bedenkt en het verzoek wil onderbreken. In het RPC model is het niet mogelijk dat een client voor hij antwoord heeft gekregen nog acties onderneemt. Voor deze situatie moet een oplossing gezocht worden zoals bijvoorbeeld een tweede client die het tweede verzoek—het afbreekverzoek—aan de server stuurt. De server moet de boodschap accepteren en de uitvoering van de vorige RPC onderbreken. Het in eerste instantie heldere model—vraag/antwoord—wordt hierdoor aanmerkelijk gecompliceerd.

Een andere oplossing zou kunnen zijn om zo gauw een record gevonden is dat aan de criteria van de client voldoet dit door de database server te laten verzenden. De gebruiker zal een snellere responstijd ervaren, hoewel de totale verwerkingstijd niet bekort is. Om steeds deelresultaten te kunnen zenden moet volgens het RPC model de client na ontvangst van elk record een RPC voor een volgend record plegen. Efficiëntere oplossingen zijn met het virtuele circuit model zeker denkbaar [Tanenbaum 4].

Het tweede voorbeeld betreft een branch-and-bound optimalisatie algoritme. De branch-and-bound methode bouwt een boom van mogelijke oplossingen op. Een voorbeeld van een probleem dat hiermee kan worden

[19] Het poogt niet van het gebruik van expliciete communicatieprimitieven te abstraheren.

opgelost is het Travelling Salesman Probleem, TSP. De parallelle versie van dit algoritme zoekt delen van de boom parallel af. Elke processor verwerkt een deel van de boom. De toewijzing van deelbomen aan serverprocessoren geschiedt door een proces een RPC naar een vrije processor te laten doen. Om te voorkomen dat de clientprocessoren na het plegen van een RPC onbenut blijven moet er naast het oorspronkelijk clientproces voor elke serverprocessor een *agentproces* zijn. Voordat de client een remote procedure aanroept splitst hij eerst een agentproces af dat de RPC pleegt en op het antwoord blijft wachten. Op de processor zijn een aantal pseudo-parallelle processen aanwezig waarvan de agentprocessen wachten op antwoord.

Naast deze vrij gecompliceerde client structuur is er het probleem dat er bij TSP een globale kortste pad variabele moet worden bijgewerkt door de processoren. In het RPC-model is het niet mogelijk om de uitvoering van een RPC te onderbreken om een nieuw kortste pad door te geven aan alle servers. Hierdoor wordt er in een RPC-implementatie wat werk teveel gedaan, de *search overhead*. Een asynchrone multicast zou dit probleem kunnen verhelpen [Bal].

In het database voorbeeld was sprake van een server die een tweede RPC kan accepteren terwijl een andere RPC wordt uitgevoerd. Niet elke server is hiertoe in staat. Wanneer het een FIFO- en geen multi-threaded server is zal de tweede RPC moeten wachten tot de eerste klaar is indien RPC het enige interproces-communicatieparadigma is. Op pagina 70 wordt deze problematiek ook besproken.

prestaties (2)

Een van de bestaansredenen van gedistribueerde systemen is snelle verwerking van programma's—reden 1 van § 2.1.4. In het RPC model moet de client op de server wachten. Om parallellisme te bereiken moet een client zelf eerst een extra proces opstarten dat asynchroon aan de RPC uitgevoerd kan worden. De synchrone RPC is een eenvoudig, maar inefficiënt paradigma. Om hoge prestaties te bereiken moet meer werk verricht worden dan bij het virtueel circuit model. Aan de serverkant geldt een soortgelijke problematiek.

De RPC lijkt op een gewone procedure aanroep. Veel programmeurs structureren hun programma zo dat veel kleine procedures veelgebruikte taken uitvoeren. In een sorteermodule kan men bijvoorbeeld een procedure die twee waarden omwisselt aantreffen. Wanneer zo'n kleine, veelgebruikte procedure per ongeluk een remote procedure wordt zal de verwerkingstijd van het programma door de communicatieoverhead sterk toenemen.

Bij het gebruik van een niet-transparant communicatiemodel als het virtueel circuit model is zo'n vergissing niet mogelijk [Tanenbaum 4].

transparantie (3)

Een RPC moet transparant zijn voor de applicatieprogrammeur. Sommige eigenschappen van de shared-memory procedureabstractie zijn in gedistribueerde systemen niet of moeizaam te implementeren. Dit zijn globale variabelen, pointervariabelen, referenceparameters, typechecking van parameters en faal-semantiek. Deze punten zijn in § 3.4.2 behandeld. Het RPC-model suggereert volledige transparantie, terwijl deze niet wordt gerealiseerd.

conclusie

Systemen die alleen de RPC als primaire communicatie primitieven aanbieden schieten voor sommige toepassingen te kort. RPC is niet geschikt als *algemeen* communicatieparadigma. Alternatieven als point-to-point message-passing moeten voorhanden blijven.

Hoofdstuk 4

Beoordeling Sun omgeving

In dit hoofdstuk wordt de gedistribueerde applicatieontwikkelomgeving van Sun besproken.

4.1 Beschrijving

4.1.1 Gedistribueerd systeem

Een Sun systeem is opgebouwd uit een verzameling werkstations die in een LAN met elkaar verbonden zijn. De Sun werkstations zijn computers die geschikt zijn voor grafische toepassingen. Ze hebben een beeldscherm met een hoge resolutie. Het operating system presenteert zich met een GUI aan de gebruiker. In de typologie van § 2.3.1 valt de Sun omgeving in de categorie werkstation/server model.

Als LAN wordt Ethernet [Matthijsen, Tanenbaum 3] toegepast. De meeste werkstations binnen een Sun systeem bevatten geen secundair geheugen. Het zijn *diskless* werkplekken. In het netwerk zijn een paar fileservers met genoeg opslagcapaciteit voor alle gebruikers opgenomen. De bestanden van alle fileservers tezamen zijn in één virtueel file systeem opgenomen. Voor gebruikers en applicatieprogrammeurs lijkt het alsof er een gecentraliseerd file systeem aanwezig is.

Deze transparantie wordt gerealiseerd door een component van het operating system met de naam NFS (Network File System). Het operating system heet SunOS. Het is gebaseerd op UNIX variant BSD 4.3. Net als UNIX kent SunOS alleen heavy-weight processen.

Het SunOS is een compromis tussen een netwerk operating system en een gedistribueerd operating system. Op het gebied van bestandsbeheer biedt NFS GOS-primitieven aan. Wat betreft geheugen- en procesbeheer zijn de primitieven niet transparant. Het zijn NOS-primitieven. Eventuele transparantie wordt hier aan de level 5 vertalers overgelaten.

Aan de bestaande mechanismen voor interproces-communicatie van BSD UNIX [Rochkind] heeft Sun de RPC toegevoegd. NFS is met behulp van RPC's geschreven. Er bestaan hulpmiddelen waarmee applicatieprogrammeurs de RPC-primitieven voor hun programma's kunnen gebruiken.

4.1.2 Applicatieontwikkeling

De ontwikkeling van toepassingsprogramma's is gebaseerd op conventionele systeemontwikkelhulpmiddelen voor sequentiële UNIX operating systems. De meeste ontwikkeltools zijn op ontwikkeling in C georiënteerd.

De Sun omgeving is ontworpen voor het ontwikkelen van client/server applicaties (§ 3.4.4). Dit zijn applicaties die om reden 3 en 4 van § 2.1.4 op gedistribueerde systemen geïmplementeerd worden.

UNIX

De basis van UNIX-applicatieontwikkeling is de C compiler, `cc`. De C compiler kan weinig typechecks uitvoeren omdat C veel impliciete typeconversies toestaat [Kernighan 2]. Voor het opsporen van dubieuze constructies bestaat het C syntax analyseprogramma `lint`. Voor C programma's die uit meerdere modules bestaan kan de compilatievolgorde bewaakt worden met `make`—gescheiden compilatie (pagina 67). Voor een programma kan de programmeur afhankelijkheden tussen modules en declaratiefiles (ook wel `#include`-files of `.h`-files genoemd) opstellen. Bij Ada worden deze afhankelijkheden door de compiler zelf bewaakt.

Om verschillende versies van programmamodules efficiënt te beheren bestaat het pakket `sccs` (Source Code Control System). Het achterhalen van semantische fouten at run-time kan met verschillende debuggers als `adb`, `sdb` en `xdb` geschieden.

RPC

Voor de ontwikkeling van gedistribueerde applicaties is geen nieuwe standaard programmeertaal ontwikkeld. Het operating system bevat RPC-primitieven die via bibliotheekroutines kunnen worden aangeroepen. Applicaties worden in C geschreven. Het is een omgeving volgens § 3.2.1.

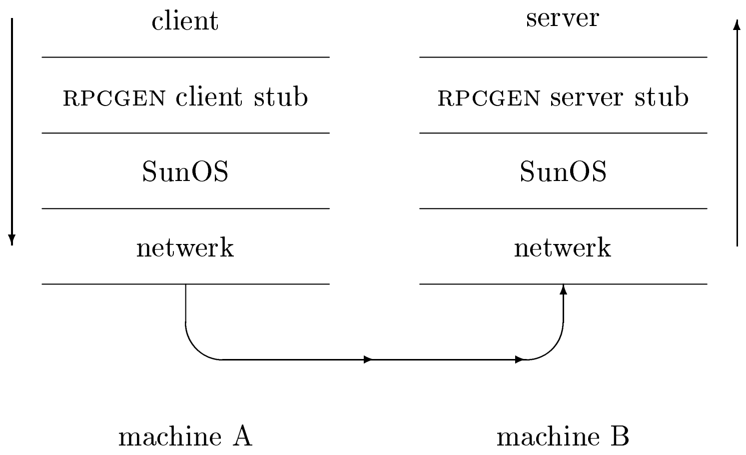
Niet elke programmeertaal is voor het programmeren van elk soort toepassing geschikt [Hoare]. Voor gedistribueerde applicatieontwikkeling ondersteunt Sun alleen C. De taalelementen (concepten) die de virtuele C machine zijn gebruiker aanreikt zijn primair op aansluiting op de level 3 laag gericht. Er worden weinig high level primitieven geboden. C is een machinegerichte programmeertaal [Kernighan 2, Rochkind, Bal]. Voor het implementeren van sommige toepassingen kan men beter voor een andere taal kiezen, mocht deze keuze bestaan.

Sun ondersteunt een standaard voor de uitwisseling van complexe datastructuren in heterogene gedistribueerde systemen. Met behulp van deze standaard, de XDR of External Data Representation, is het mogelijk applicaties te schrijven die gegevens uitwisselen met computers van andere leveranciers. Het is mogelijk te communiceren met onder andere SUN werkstations, DEC VAX-en, IBM PC's en Cray supercomputers [Sun 1].

Programmeren met RPC-primitieven uit bibliotheekroutines wordt in [Sun 1] 'low level' genoemd. Vanwege de complexiteit wordt afgeraden de RPC-primitieven *direct* te gebruiken voor het schrijven van programma's.

Naast deze low level primitieven biedt SunOS een stubgenerator aan, RPCGEN. Voor deze generator moet een interfacemodule geschreven worden. Deze interfacemodule wordt door RPCGEN gebruikt om client- en serverstubs te genereren. De stubs vervullen dezelfde functies als in het Cedar systeem van Birrell & Nelson (§ 3.4.3, [Birrell]). In figuur 4.1 is conceptueel de plaats van stubs en SunOS weergegeven. De figuur geeft aan hoe een applicatieprogrammeur de verschillende virtuele machines ziet.

Figuur 4.1: Sun RPC omgeving



In de interfacemodule worden de datatypes van parameters van remote procedures gespecificeerd. Dit gebeurt in de XDR-taal. De interfacemodule wordt ook wel `.x`-file genoemd. De stubs die RPCGEN genereert bevatten aanroepen van XDR-primitieven waarmee de parametermarshalling plaatsvindt. Ze bevatten ook de aanroepen van RPC-primitieven voor verzending en ontvangst van parameters, en het dispatchen van de RPC. RPCGEN genereert daarnaast een *headerfile* met declaraties die nodig zijn om de stubs aan te roepen.

Marshalling is het verzendklaar maken van parameters door de verzender en het weer uitpakken door de ontvanger. Gecompliceerde datastructuren als gelinkte lijsten en bomen moeten geconverteerd worden in een vorm die het operating system kan verwerken. Bij overdracht tussen verschillende typen operating systems moeten de gegevens in een standaardrepresentatie gecodeerd worden.

Dispatching vindt bij de server plaats. Elke remote procedure heeft een uniek nummer. In verzoeken van client aan server wordt met dit nummer aangegeven welke serverprocedure aangeroepen wordt. Servers bestaan uit een dispatcher die op basis van het verzoek de gevraagde procedure aanroept.

Door het gebruik van RPCGEN wordt het aantal system-calls dat programmeurs zelf moeten specificeren sterk teruggedrongen.

werkwijze RPCGEN

In [Chin, p. 6] is samengevat hoe een RPCGEN-applicatie wordt gebouwd:

1. Maak de file met interfacedefinities (*naam*`.x`).
2. Genereer de stubs met RPCGEN (*naam*`_clnt.c` en *naam*`_svc.c`).
3. Schrijf de client- en servermodules.
4. Include de headerfile (*naam*`.h`) in client en server en compileer beide gescheiden.
5. Compileer de serverstub (*naam*`_svc.c`) en de file met de datatransportroutines (*naam*`_xdr.c`). Link deze met de servermodule.
6. Compileer de clientstub (*naam*`_clnt.c`) en de file met de datatransportroutines (*naam*`_xdr.c`). Link deze met de clientmodule.
7. Start de server op de servermachine.
8. Start de client op de clientmachine.

In figuur 4.2 zijn de verschillende files die bij dit proces een rol spelen weergegeven.[1] In de figuur wordt als voorbeeld aan de hand van een fictieve applicatie beschreven welke files door welke compiler of generator gebruikt worden. De applicatie bestaat uit twee (sets) files: spreadsheet.c en dbms.c. De client kant is een gebruikersinterface in de vorm van een spreadsheet. Aan de server kant zit een database back-end dat queries en updates van de client uitvoert.

Om deze applicatie te compileren wordt eerst de interfacemodule geschreven. RPCGEN genereert twee stubfiles en een dataconversiefile. Vervolgens worden twee uitvoerbare programma's gecompileerd: de client en de server. In dit voorbeeld is uitgegaan van één client en één server. Het is mogelijk om complexere configuraties te programmeren. Zie bijvoorbeeld [Chin].

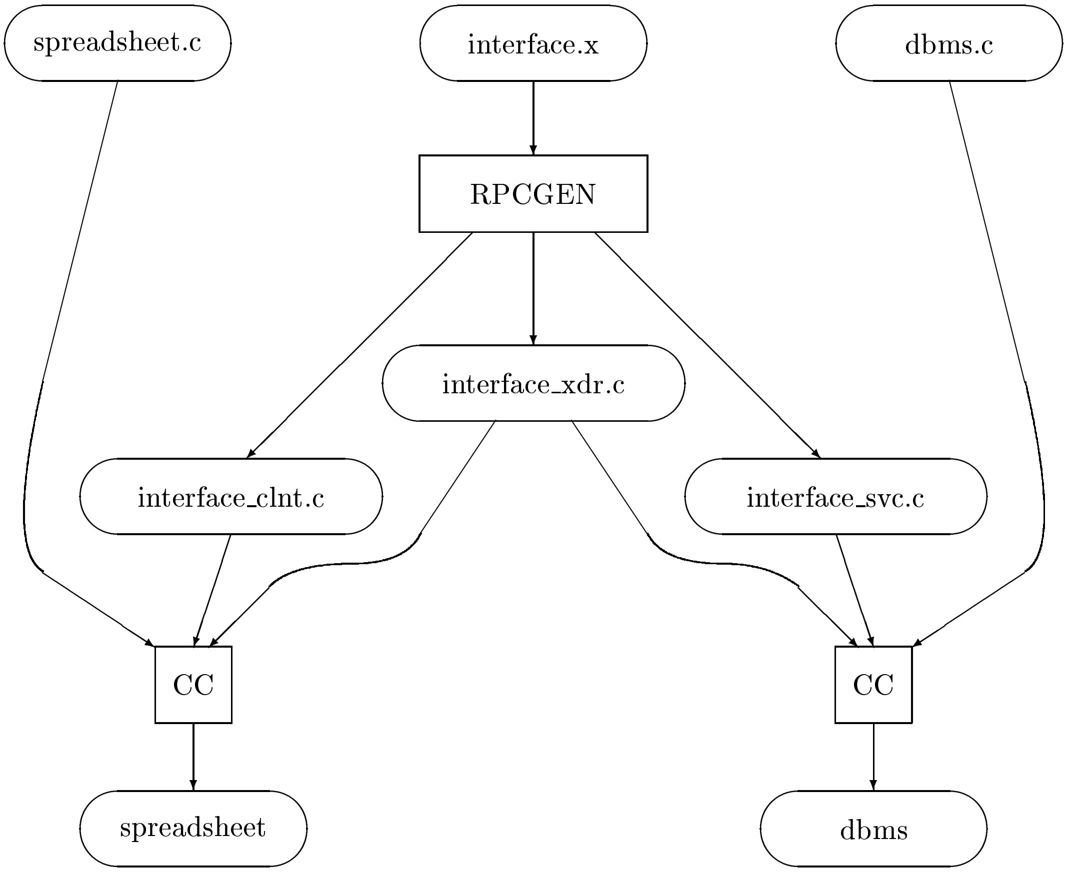
RPCGEN

De stubgenerator heeft een definitiefile nodig waar de applicatieprogrammeur in declareert welke remote procedures de applicatie bevat en wat de parametertypen zijn. De interfacedefinitie heet bij RPCGEN een `.x` of XDR file. De voornaamste taak is er voor zorgen dat client en server van dezelfde procedure declaratie uitgaan. Vergelijk de prototypes die ANSI aan C heeft toegevoegd (pagina 67).

RPCGEN RPC's kunnen maar één input parameter en één output parameter hebben. Wanneer er toch meerdere parameters nodig zijn moeten deze in een structure geplaatst worden. Deze structure is dan de ene parameter van de RPCGEN stubs. In het voorbeeld hierna heeft functie 'bereken' één input structure en één output structure. De remote procedure zou als `bereken_uit = bereken(bereken_in);` aangeroepen moeten worden.

[1] Uitgezonderd de headerfile.

Figuur 4.2: RPCGEN files




```
/*
** interface.x
 ** XDR file met de interface definitie van een
** client-server applicatie met 1 remote procedure
*/

/*
** Function 'bereken' -- input parametertypen
*/

struct bereken_IN_t {           type van de input parameter
   int nb;                      XDR integer declaratie
   float b[10];                 XDR array declaratie
   int na;
   float a[10];
```

```
};                                   in totaal vier input parameters

/*
** Function 'bereken' -- output parametertypen
*/

struct bereken_UIT_t {               type van de output parameter
   int resultaat;                    returnwaarde van de remote functie
   int nc;
   float c[10];
};                                   in totaal drie output parameters

/*
** Versionnumbers for program and remote routines
*/

program PROG {
   version VERS {
      bereken_UIT_t bereken(bereken_IN_t) = 1;
                                     volgnummer van de procedure: 1
   } = 13;                           versienummer van het programma
} = 0x2000076;                       programmanummer volgens Sun norm
```

Voordat een stub aangeroepen mag worden moet eerst een initialisatie functie worden aangeroepen. Deze kijkt of het opgegeven serverprogramma in het systeem aanwezig is. Deze functie heet `clnt_create()`. De applicatieprogrammeur moet eventuele foutcodes van de functie zelf afhandelen. De beslissing of het programma moet stoppen na een fout of het nog eens moet proberen wordt aan de programmeur over gelaten.

Bij de `clnt_create()` initialisatie kan de programmeur opgeven via welk netwerkprotocol de boodschappen verzonden moeten worden. Men kan UDP kiezen voor datagram verkeer. Na een time-out volgt retransmission. Deze optie is geschikt voor applicaties die idempotent zijn (pagina 62). Wanneer gekozen wordt voor TCP wordt een connection oriented communicatieprotocol gebruikt. Deze optie biedt at-most-once semantiek. Zij is geschikt voor applicaties die procedures met side-effects hebben.

De client stub geeft een pointer naar de output parameter terug. Voordat deze parameter gebruikt mag worden moet de programmeur eerst kijken of deze misschien `NULL` is. Dit geeft aan dat er een RPC systeemfout is opgetreden. Pas nadat er op een RPC-fout getest is mag de eigenlijke interpretatie van het functieresultaat plaatsvinden.

4.2 Beoordeling ontwikkelomgeving

In hoofdstuk 3 zijn twee groepen criteria aangelegd. De eerste groep van acht criteria dient voor de beoordeling van gedistribueerde applicatieontwikkelomgevingen in het algemeen. De tweede groep van elf criteria is gericht op specifieke kenmerken van systemen die op de remote procedure call zijn gebaseerd.

Hierbij staat + voor overeenstemming met deze punten, ∘ voor redelijke overeenstemming, en − voor twijfelachtig.

4.2.1 Gedistribueerdheid

De Sun omgeving kent geen speciale gedistribueerde taal. De compiler voor gedistribueerde applicaties is de standaard UNIX C compiler. Gedistribueerde primitieven worden met behulp van bibliotheekroutines aangeroepen. Er bestaat een stub-generator die analoog aan § 3.4.3 het gebruik van de bibliotheekroutines vereenvoudigt.

parallellisme (1)

Gecentraliseerde sequentiële talen kennen het *proces* als eenheid van uitvoering. De Sun omgeving is op C, een sequentiële, procedurele taal gebaseerd. De omgeving kent alleen expliciet parallellisme. De eenheid van parallellisme is het proces. Elk C programma vormt een proces. In een client/server applicatie moet men minstens twee programma's schrijven. Een clientprogramma en een serverprogramma.

Processen worden lokaal impliciet gescheduled door uitvoering van een RPC. Het RPC-model is gericht op procedures, niet op processen. Meerdere processen kunnen met de standaard UNIX `fork` en `exec` primitieven expliciet gecreëerd worden. Voor applicatieontwikkeling zijn dit vrij ingewikkelde primitieven. De Sun omgeving kent alleen heavy-weight processen.

Het aantal processen wordt at compile-time bepaald. De globale scheduling wordt door de programmeur zelf verzorgt door client en server expliciet op een bepaalde machine op te starten. Lokale scheduling zou men met behulp van UNIX system calls kunnen beïnvloeden.

Expliciet parallellisme met proces als eenheid van parallellisme is geschikt voor het ontwikkelen van client/server applicaties. De scheduling moet door de programmeur verricht worden. De Sun omgeving biedt een laag abstractieniveau—flexibiliteit ten koste van transparantie. Dit past bij de programmeertaal C [Kernighan 2, p. 1]. Het punt parallellisme is bevredigend uitgewerkt: ∘

interproces-communicatie (2)

Het Sun operating system biedt message-passing in twee vormen aan: datagram en connection oriented. Het datagramprotocol is het Unreliable Data Protocol of UDP. Het connection oriented protocol is het Transmission Control Protocol of TCP. Beide maken—in de Sun omgeving—gebruik van Ethernet LAN's. Point-to-point messages-passing met UDP of TCP verloopt synchroon.

De RPC-communicatie kan van beide protocollen gebruik maken. Dit is van belang voor de faal-semantiek (zie pagina 67). De Sun omgeving implementeert geen vormen van gemeenschappelijke gegevens. RPC is het enige *high level* communicatieparadigma van de Sun omgeving (zie § 3.4.5 voor de nadelen hiervan).

Het RPC-paradigma past bij de vorm van parallellisme. Er wordt een grote verscheidenheid aan primitieven ondersteund: + Helaas is RPC het enige high level paradigma: +/∘

fouttolerantie (3)

Het Sun operating system noch de taal C is fouttransparant. Ze bieden ook geen bouwstenen als atomaire objecten. De taal C kent geen exceptions. De bibliotheekroutines van de RPC-primitieven retourneren een foutcode om een foutconditie te signaleren. De applicatieprogrammeur moet telkens wanneer een procedure wordt aangeroepen het resultaat op fouten controleren. Men moet zelf de gewenste mate van fouttolerantie programmeren.

De Sun omgeving geeft een slechte ondersteuning voor een van de twee bestaansredenen van gedistribueerde systemen: parallellisme en fouttolerantie (§ 2.3.3). Oordeel: −

integratie semantiek (4)

De taal C staat het gebruik van globale variabelen toe. Er wordt veel gebruik gemaakt van pointervariabelen, met name voor de overdracht van referenceparameters. Deze methodes zijn niet bruikbaar in gedistribueerde applicaties. Het is makkelijk om hier fouten mee te maken. De Sun omgeving biedt geen ondersteuning bij het opsporen van dit soort fouten.

De sequentiële taal C is niet zo geschikt voor een gedistribueerde omgeving: − Door de interfacecompiler RPCGEN worden een aantal van deze problemen opgelost (zie punt 5 en punt b). Oordeel: ∘/−

complexiteit (5)

De combinatie van C en RPC-system-calls is complex. Om het gebruik te vereenvoudigen is een interface-compiler aan het ontwikkeltraject toegevoegd. Hierdoor kan de programmeur voor eenvoudige applicaties de meeste bibliotheekroutines vermijden.

De RPC omgeving bestaat uit een grote verzameling bibliotheekroutines. Het doel dat Birrell & Nelson op pagina 8 formuleerden was ervoor zorgen dat niet alleen specialisten maar ook gewone applicatieprogrammeurs programma's voor gedistribueerde systemen zouden kunnen schrijven.

Voor applicaties die enigszins andere eisen stellen dan een standaard synchrone RPC biedt de Sun omgeving een slechte ondersteuning. De primitieven zijn aanwezig[2] maar het gebruik ervan is ingewikkeld. De Sun omgeving biedt buiten RPCGEN slechts low level primitieven. Deze zijn voor systeemprogrammeurs, niet voor applicatieprogrammeurs.

[2]Bijvoorbeeld voor asynchrone clients, asynchrone servers en broadcast RPC.

RPCGEN verminderd de complexiteit van de C/system-call combinatie aanmerkelijk, er kleven echter ook nadelen aan RPCGEN (zie punt c en f). Oordeel: $\circ$

typesecurity (6)

C is niet typesecure. De introductie van prototypes in ANSI C heeft parameter type-checks mogelijk gemaakt. Desondanks zijn er mogelijkheden te over om fouten te maken. Met name het gebruik van pointers en arrays geeft aanleiding tot veel fouten.

Het programma `lint` kan een aantal syntactisch juiste maar mogelijk incorrecte constructies onderkennen. Run-time checks moet de programmeur zelf programmeren.

De interface-compiler RPCGEN waarborgt consistentie van parameters tussen aanroep en definitie van remote procedures.

C is niet typesecure: $-$

efficiëntie (7)

Het UNIX operating system is een monolitische monitor. Een contextswitch kost relatief veel tijd. Bij het uitvoeren van een RPC doorloopt de aanroep een groot aantal software lagen. De parameters van een aanroep worden veelvuldig gekopieerd: van client naar stub, van stub naar kernel en van kernel naar netwerkbuffer. Micro-kernel operating systems zijn beter geschikt voor snelle communicatie dan het UNIX operatingsystem [Coulouris].

Voor de bouw van applicaties biedt SunOS alleen de RPC aan als eenvoudig te gebruiken primitieve. In § 3.4.5 zijn een aantal redenen genoemd waarom het RPC paradigma een efficiënte implementatie van sommige problemen in de weg staat. In uitzonderingsgevallen kan men de low level UNIX socket-interface gebruiken. Deze implementeert `send` en `receive` primitieven. Helaas zijn de low level communicatie primitieven niet afgestemd op gebruik samen met de high level primitieven. In de Sun documentatie wordt geadviseerd om uitsluitend de high level RPC primitieven samen met de interface-compiler te gebruiken.

De efficiëntie van UNIX is voor een RPC-systeem niet optimaal. Het RPC-paradigma is een redelijk efficiënt communicatieparadigma (zie pagina 45). Oordeel: $\circ$

overdraagbaarheid (8)

SunOS ondersteunt de XDR standaard voor gegevensuitwisseling in heterogene netwerken. De XDR standaard is volgens [Sun 1] vrijwel gelijk aan de Abstract Syntax Notation van het ISO OSI referentiemodel, X.409. Op dit punt is het een zogenaamd 'open systeem'.

Wat betreft de overdraagbaarheid van de programmatuur geldt de restrictie dat het platform een UNIX operating system moet zijn dat UDP of TCP als communicatieprotocol ondersteunt. Om de applicatie te kunnen compileren moet de omgeving RPCGEN en de door Sun gebruikte RPC-primitieven ondersteunen. Doordat sommige leveranciers dit doen is het mogelijk applicaties te schrijven die ook op andere systemen dan Sun werkstations kunnen opereren.

Er bestaat een grote variëteit tussen implementaties van RPC-primitieven in verschillende gedistribueerde operating systems [Coulouris]. Voor het overdragen van een RPCGEN-applicatie naar een omgeving die niet de Sun RPC-primitieven ondersteunt zal de programmeur de communicatiemodules van het programma moeten aanpassen. Er zijn nog geen standaarden op dit gebied.

De Sun omgeving conformeert zich aan relevante standaarden: +

4.2.2 RPC

In deze paragraaf worden specifieke RPC aspecten van de Sun applicatieontwikkelomgeving beoordeeld. In § 3.4.2 zijn per punt een aantal mogelijkheden besproken. Op basis van deze mogelijkheden worden hier de keuzes die de Sun ontwerpers gemaakt hebben besproken.

a. globale variabelen

Probleem: *interproces globale variabelen hebben geen betekenis in RPC systemen.*

C staat het gebruik van globale variabelen voor communicatie tussen twee procedures toe. In de Sun omgeving bestaat elk proces uit een apart programma. Elk programma wordt gescheiden gecompileerd door de C compiler. De interfacecompiler compileert alleen de `.x`-file. Het programma `lint` werkt op een per-programma basis. Het biedt in dit geval geen oplossing. In het ontwikkelproces wordt het gebruik van dezelfde naam voor een globale variabelen in verschillende programma's niet gesignaleerd. De programmeur wordt niet gewaarschuwd tegen dit soort fouten.

Bij het ontwikkelen van conventionele pseudo-parallelle multiproces applicaties voor uniprocessoren onder UNIX doet zich dit probleem ook voor. Programmeurs die met de UNIX omgeving bekend zijn worden niet met een onbekend fenomeen geconfronteerd.

Pogingen tot foutief gebruik van interproces globale variabelen worden niet onderkend: −

b. pointervariabelen

Probleem: *pointervariabelen in een andere adresruimte—een ander proces—hebben geen zin.*

Het is niet mogelijk om in de XDR taal in de interfacedefinitie pointervariabelen te declareren. Pogingen hiertoe worden door RPCGEN niet geaccepteerd. In C worden vectoren (arrays) en recursieve datastructuren met behulp van pointers opgebouwd. Voor het gebruik van onder andere bomen, gelinkte lijsten en grafen als parameter kan RPCGEN marshalling-code genereren waarmee deze structuren ontleed kunnen worden voor verzending en weer opgebouwd bij ontvangst. Deze code maakt deel uit van de RPCGEN-stubs.

Wanneer de programmeur de datastructuren eenmaal in de interfacedefinitie gedeclareerd heeft is dit proces verder transparant.

Pogingen tot foutief gebruik van interproces pointervariabelen worden wel onderkend, en recursieve datastructuren zijn mogelijk: +

c. parameters

Probleem: *de semantiek van reference parameters is moeilijk te verwezenlijken in RPC systemen.*

Sun remote procedures kunnen hooguit één input parameter accepteren en één resultaatwaarde afleveren. Wanneer de programmeur meerdere parameters wil gebruiken moet deze ze in *structures* (tuples) groeperen. Zie punt f.

De programmeur specificeert in de interfacedefinitie zelf welke parameters input, en welke output zijn. Het begrip *call by reference* bestaat niet bij Sun RPC's. De programmeur moet expliciet de richting van de overdracht declareren. Sun remote procedures zijn op dit punt niet transparant voor de gebruiker.

Parameteroverdracht in C verschilt fundamenteel van parameteroverdracht van RPCGEN RPC's: −

d. faal-semantiek

Probleem: *de semantiek van lokale procedures is voor remote procedures bij uitvallen van systeemcomponenten moeilijk te ondersteunen.*

De Sun omgeving kent het conventionele proces begrip (in tegenstelling tot het functionele, zie pagina 62). Processen en procedures kunnen side-effects hebben. De Sun omgeving is niet fouttransparant en kent geen bouwstenen om zero-or-one semantiek mee te implementeren.

Het is mogelijk om voor elke remote procedure het communicatieprotocol te specificeren en zo tussen datagram of connection oriented verkeer te kiezen. Men moet specificeren of het UDP of het TCP gebruikt moet worden. Op pagina 62 en 67 is aangegeven dat op deze wijze voor at-least-once of at-most-once semantiek gekozen kan worden. Afhankelijk van het type procedure—idempotent of side-effects—kan de programmeur de gewenste semantiek kiezen.

Het herstel van de toestand van de omgeving van een halverwege onderbroken procedure wordt bij at-most-once semantiek aan de programmeur overgelaten. Net als bij de last-one semantiek van een lokale procedure kan dit voor applicaties die een hoge mate van fouttolerantie vereisen problematisch worden. Voor de implementatie van een gedistribueerd vluchtreserveringssysteem of een gedistribueerde banktoepassing is deze vorm van semantiek minder geschikt [Liskov].

Of een toepassing met side-effects zero-or-one omgevingen als Argus [Coulouris, Liskov] nodig heeft of in een at-most-once omgeving geïmplementeerd kan worden komt neer op een afweging tussen fouttolerantie en efficiëntie (§ 3.3.3).

Gegeven de het feit dat de Sun omgeving geen faciliteiten voor de implementatie van fouttolerantie applicaties biedt (punt 3), is de keuze voor at-most-once of at-least-once semantiek van een RPC een waardevolle mogelijkheid voor de programmeur: +

e. foutcode/exceptions

Probleem: *op welke wijze moet de programmeur de foutafhandeling in de applicatie formuleren?*

De taal C kent geen exceptions. Applicatieprogrammeurs moeten bij elke RPC het resultaat op fouten controleren. Pas wanneer er geen RPC-subsysteemfouten gesignaleerd worden kan men het resultaat gebruiken. Daarbij moet de programmeur zelf de gewenste mate van fouttolerantie programmeren.

RPC applicaties bevatten meer foutafhandelingscode dan gecentraliseerde applicaties. De programmeur wordt niet door de Sun omgeving tot het schrijven van deze code verplicht. Veel run-time fouten treden op doordat programmeurs vergeten uitputtende foutafhandelingscode te schrijven [Chin]. Het feit dat C niet typesecure is, is ook bij gecentraliseerde applicatie een probleem [Rochkind]. In een gedistribueerde context is dit probleem groter omdat parallelle applicaties vaak non-deterministisch gedrag vertonen [Bal].

Voor de test- en foutherstel fase in het ontwikkelproces van een applicatie moet in de Sun omgeving veel tijd worden gereserveerd.

De Sun omgeving is niet typesecure en kent geen exception-mechanisme: –

f. integratie in programmeertaal

Probleem: *gedistribueerde en sequentiële primitieven samen leveren niet altijd een consistente programmeeromgeving op.*

In het ontwikkeltraject wordt de standaard UNIX C compiler gebruikt. De RPC-primitieven worden via de bibliotheek aangeroepen. In [Sun 1] wordt aangegeven dat gebruik van deze primitieven erg ingewikkeld is. Het wordt afgeraden ze direct te gebruiken. Hiertoe biedt men RPCGEN aan. Om RPCGEN te gebruiken moet de programmeur een interfacedefinitie schrijven.

RPCGEN genereert stubs. Deze stubs zijn omslachtig in het gebruik [Chin]. Het probleem is dat ze maximaal één input- en één outputparameter kennen. Voor elke aanroep van een stub moet de programmeur eerst de parameters in een structure plaatsen. De serverprocedure moet deze structure weer uitpakken om de parameters te kunnen gebruiken. Voor het verzenden van het resultaat moet de serverprocedure de outputparameters in de outputstructure plaatsen. De aanroeper moet deze, na controle op RPC-systeemfouten, dan weer uitpakken.

Een bijkomend probleem is dat bij dit proces gebruik wordt gemaakt van pointers naar parameters. Het C run-time systeem voert geen validi-teitscontroles uit op de waarden van pointerverwijzingen. Hier worden veel fouten mee gemaakt [Chin]. Verder moet de programmeur er aan denken op het juiste moment geheugen dat XDR RPC-primitieven gealloceerd kunnen hebben vrij te geven [Sun 1].

Het gebruik van RPC-primitieven is door de RPCGEN-stubs vereenvoudigd, het gebruik van de stubs zelf vereist de nodige voorzichtigheid. De programmeur moet bij een RPC in vergelijking met een lokale procedure-aanroep veel handelingen verrichten. Hierbij kunnen veel fouten gemaakt worden. [Hoare]'s uitgangspunt dat een taal alle afwijkingen van de taalregels moet signaleren is hier niet toegepast.

Birrell & Nelson (§ 3.4.3 hebben stubs geïntroduceerd om een remote aanroep te vertalen in een lokale aanroep teneinde de transparantie van het RPC-concept te verhogen. Vanwege de wijze van parameteroverdracht zijn RPCGEN-stubs niet transparant voor de programmeur.

De integratie van RPCGEN-stubs en C is slecht: −

g. interfacedefinitie

Probleem: *op welke manier moet de programmeur de interface tussen client en servermodules specificeren?*

Veel RPC systemen kennen een interfacecompiler [Coulouris]. In de Sun omgeving worden in de interfacedefinitie remote procedures en de datatypen voor de parameters gespecificeerd. Ook nummers ter identificatie van programma, versie en procedure moeten worden gegeven. De programmeur moet deze zelf kiezen. Het is mogelijk dat twee programmeurs in het systeem hetzelfde nummer kiezen. Sun microsystems treedt in deze op als centraal coördinatiepunt.

De specificatiesyntax voor de parameters in de `.x`-file is de XDR taal. Deze lijkt sterk op C. In § 5.4.3 worden uitzonderingen beschreven.

De XDR-syntax sluit goed aan op de taal C: +

h. gegevensoverdrachtprotocol

Probleem: *(a) het RPC systeem moet een communicatieprotocol aanbieden dat voor de toepassing geschikt is (snel of betrouwbaar); (b) het RPC systeem moet representatieproblemen in heterogene netwerken oplossen.*

Bij het eerste punt kan de programmeur zelf voor datagram—snel/idempotent—of connection oriented—betrouwbaar/side-effects—kiezen. Een negatieve bijkomstigheid is dat de pakketgrootte van een UDP-datagram niet meer dan 8 kilobytes mag zijn. Voor vectoren of andere samengestelde datastructuren kan dit te klein zijn.

In punt d is het belang van de mogelijkheid om afhankelijk van de toepassing de gewenste semantiek te kunnen kiezen beschreven. De Sun omgeving biedt hier adequate mogelijkheden toe.

Het tweede punt betreft gegevensrepresentatie. Het Sun RPC-subsysteem maakt geen gebruik van volgens het OSI-model gestructureerde communicatieprotocollen. Het moet representatieproblemen zelf oplossen. Hiertoe biedt SunOS XDR-primitieven in de vorm van bibliotheekroutines. RPCGEN-stubs maken gebruik van deze primitieven. Behalve door specificatie van parametertypen in de interfacedefinitie, hoeft de programmeur zich hier niet mee bezig te houden.

De programmeur kan kiezen tussen UDP en TCP, en voor de gegevensconversie gebruikt Sun de XDR standaard: +

i. netwerkadressering

Probleem: *op welke wijze worden aanroeper en remote procedure in het netwerk bekend gemaakt?*

Op pagina 68 is een methode beschreven waarmee *name-binding*—het vaststellen van netwerkadressen—at run-time kan plaatsvinden. De server meldt zich aan bij een centraal punt dat de client bij de eerste RPC raadpleegt. Deze vorm van binding paart flexibiliteit aan efficiëntie [Birrell].

De Sun omgeving past een vergelijkbare methode toe. Op elke machine is een zogenaamde 'portmapper' aanwezig. Deze vervult de functie van centraal punt dat de netwerkadressen van servers aan clients doorgeeft.

De portmapper van de Sun omgeving werkt goed: +

j. beveiliging

Probleem: *biedt het RPC systeem een efficiënte en effectieve beveiliging?*

De programmeur kan in beperkte mate de gewenste vorm van beveiliging kiezen die bij client/server communicatie gebruikt moet worden. Het Sun RPC systeem ondersteunt (a) *geen beveiliging*, (b) *standaard UNIX identificatie* en (c) *DES identificatie.* Het RPC systeem implementeert alleen de identificatie van clients. De programmeur van de server moet zelf op basis van het identificatieresultaat besluiten welke actie wordt ondernomen. Dit kan bijvoorbeeld het niet uitvoeren van de RPC en het terugsturen van een statuscode met de reden van weigering zijn.

Het Sun RPC-subsysteem kent geen primitieven voor versleuteling van berichten.

De Sun omgeving kent alleen voorzieningen voor identificatie, niet voor versleuteling: ∘

k. parallellisme

Probleem: *het is moeilijk met synchrone communicatieprimitieven een hoge mate van parallellisme te bereiken in een applicatie.*

De Sun omgeving ondersteunt single-threaded FIFO servers. Wanneer een RPC aankomt bij een server terwijl deze bezig een andere RPC uit te voeren wordt de laatste in een wachtrij geplaatst. Op pagina 70 zijn technieken als multi-threaded servers en stand-by processen beschreven om de mate van parallellisme van gedistribueerde applicaties te verhogen. Deze worden niet toegepast.

Een belangrijke beperking die hierin een rol speelt is dat SunOS een UNIX operating system is dat alleen heavy-weight processen ondersteunt. Het operating system van het systeem van Birrell & Nelson (§ 3.4.3) is ook een omgeving met alleen heavy-weight processen. Zij passen een techniek toe waarbij een aantal processen stand-by worden gehouden. Het RPC-subsysteem zorgt hier zelf voor creatie van een nieuwe server.

Een hieraan verwante oplossing is in [Chin] voor de Sun omgeving geïmplementeerd. Hierbij is twijfel gerezen over de wenselijkheid van parallelle servers in de Sun omgeving. Met `connect` en `disconnect` primitieven creëerde men voor elke RPC-transactie een apart serverproces. Hiermee wordt extra overhead geïntroduceerd voor elke transactie. Het creëren van een nieuw (heavyweight) serverproces bleek met name bij zwaar belaste machines lang te duren. Er zijn geen prestatiemetingen vermeld. Volgens [Coulouris] leveren operating systems die lightweight processen (multithreaded servers) ondersteunen betere prestaties.

De Sun omgeving kent alleen FIFO servers: $-$

4.2.3 Samenvatting

Hieronder wordt samengevat in welke mate de Sun gedistribueerde applicatieontwikkelomgeving aan de behandelde criteria voldoet. In deze tabel worden veel aspecten van een uitgebreide en complexe omgeving in een paar symbolen samengevat. Zo'n beoordeling kan de werkelijke waarde van de Sun omgeving nooit adequaat beschrijven. De tabel is niet meer dan een geforceerd beknopte samenvatting waarin veel belangrijke aspecten ongenoemd blijven. Om te voorkomen dat een verkeerd beeld ontstaat moet de beschrijving bij de criteria hiervoor niet uit het oog verloren worden.

Allereerst wordt de overeenstemming met de algemene aspecten van de Sun omgeving samengevat. Deze aspecten zijn beoordeeld op geschiktheid voor het ontwikkelen van toepassingsprogramma's. Hierbij staat + voor overeenstemming met deze punten, $\circ$ voor redelijke overeenstemming, en $-$ voor twijfelachtig.

Gedistribueerde omgeving

criterium	*kern van de uitkomst*	*score*
1. parallellisme	proces, scheduling: compile-time	$\circ$
2. communicatie	RPC goed, point-to-point ingewikkeld	$+/\circ$
3. fouttolerantie	geen bouwstenen	$-$
4. integratie	C niet zo geschikt voor RPC	$\circ/-$
5. complexiteit	RPCGEN is redelijk	$\circ$
6. typesecurity	C is niet typesecure	$-$
7. efficiëntie	UNIX overhead, alleen RPC paradigma	$\circ$
8. overdraagbaarheid	XDR, UNIX	+

De volgende tabel vat de overeenkomst voor de RPC aspecten samen.

RPC aspecten

criterium		*kern van de uitkomst*	*score*
a.	globale variabelen	geen waarschuwing	−
b.	pointers	niet mogelijk als parameter	+
c.	parameters	één in, één uit	−
d.	faal-semantiek	keuze mogelijk	+
e.	foutcode	niet typesecure, geen exceptions	−
f.	integratie RPC	RPCGEN-stubs: in- en uitpakken	−
g.	interfacedefinitie	XDR, typechecks	+
h.	protocol	XDR, keuze UDP/TCP	+
i.	netwerkadressering	'portmapper'	+
j.	beveiliging	alleen identificatie	∘
k.	parallellisme	FIFO-server	−

Uit de beoordeling komt naar voren dat de volgende punten verbeterd kunnen worden:

1. *Fouttolerantie* Bouwstenen voor het implementeren van fouttolerante toepassingen ontbreken. Hier wordt onderzoek naar verricht. Camelot [Spector] is een bibliotheek voor de taal C die transactieprimitieven implementeert.

2. *Integratie* De integratie van gedistribueerde primitieven in de sequentiële taal C is matig. Een andere taal is hier wellicht beter geschikt voor.

3. *Programmeerfouten* De Sun omgeving is niet typesecure. Een andere taal die wel typesecure is kan dit probleem verhelpen.

4. *Transparantie* Verkeerd gebruik van globale variabelen wordt niet onderkent. Een oplossing zou het uitbreiden van `lint` voor multiproces programma's zijn.

5. *Transparantie* De parametersemantiek van de lokale procedureabstractie wordt niet ondersteund. De programmeur moet zelf de parameters in input of output groeperen.

6. *Transparantie* De programmeur moet veel foutcontroles programmeren. Wanneer dit wordt vergeten signaleert het systeem dit niet.

7. *Transparantie* Het gebruik van de RPCGEN-stubs is moeizaam. De programmeur moet bij elke RPC parameter in- en uitpakcode schrijven. Hierbij maakt men snel fouten in het gebruik van pointers.

8. *Parallellisme* De structuur van Sun RPC servers is FIFO. Dit is niet bevorderlijk voor de mate van parallellisme van applicaties.

In het volgende hoofdstuk wordt een verbetering voor de punten 5, 6 en 7 gepresenteerd. Een oplossing voor deze punten lijkt een minder ingrijpende verandering van de Sun omgeving te vereisen dan bij de andere punten. Deze drie punten hebben alle betrekking op de mate van transparantie van de remote procedure call (probleemstelling). In hoeverre deze verhoogd kan worden is het onderwerp van de volgende twee hoofdstukken.

Hoofdstuk 5

Verbeteringen

In dit hoofdstuk wordt beschreven hoe het ontwikkelen van toepassingsprogramma's voor de Sun omgeving vereenvoudigd kan worden. Hier wordt het model beschreven, in § 6.3 en appendix A het prototype.

5.1 Problemen Sun

De Sun omgeving voldoet op een aantal punten niet aan de hiervoor beschreven criteria voor gedistribueerde applicatieontwikkelomgevingen. De Sun omgeving is niet voor alle soorten toepassingen even geschikt. Zo zijn applicaties die een hoge mate van bedrijfszekerheid vereisen eenvoudiger op systemen die zero-or-one semantiek bieden te implementeren (pagina 62). Daarnaast is het RPC paradigma bij sommige toepassingen niet geschikt voor een efficiënte implementatie (§ 3.4.5). RPC's worden door Sun RPC-servers in FIFO volgorde uitgevoerd. Voor applicaties met een hoge mate van parallellisme is dit geen goede oplossing.

Het opsporen van fouten in gedistribueerde programma's wordt niet optimaal door de Sun omgeving ondersteund. Door de complexe wijze van parameteroverdracht worden hier snel fouten bij gemaakt. Het gebruik van remote procedures is niet transparant voor de programmeur.

De probleemstelling van dit onderzoek stelt *vereenvoudiging van de Sun omgeving* voor de programmeur centraal. In § 1.3 is als doelstelling de ondersteuning van standaard C programma's geformuleerd. Uit de beoordeling van de Sun omgeving zijn vier punten naar voren gekomen waarbij de Sun RPC niet transparant is voor de programmeur. Dit zijn op pagina 93 punt 4, 5, 6 en 7.

In punt 4 wordt beschreven dat de Sun omgeving pogingen tot het gebruik van interproces globale variabelen niet onderkent. Om dit op te lossen kan men de functionaliteit van bestaande syntax-analyse hulpmiddelen als `lint` uitbreiden voor gedistribueerde omgevingen. Voor dit onderzoek is dit, ondanks de uitgesproken mening van [Hoare] in deze, niet gedaan. De scope van globale variabelen is in een sequentiële omgeving het *programma*. Bij het ontwikkelen van een gedistribueerde applicatie moet de programmeur voor elk benodigd proces een apart programma creëren—elk met intern een eigen `main()` procedure. Doordat de programmeur expliciet een multi-*programma* applicatie schrijft ligt het minder voor de hand dat men een onjuiste semantiek van interproces globale variabelen veronderstelt. In [Chin] is dit punt niet als probleem genoemd.

5.1.1 RPCGEN-stubs

Punt 5, 6 en 7 zijn problemen die optreden bij het gebruik van door RPCGEN gegenereerde stubs. Deze zijn ook in [Chin] genoemd. RPCGEN-stubs hebben één input en één output parameter. Om meerdere parameters te gebruiken moet de programmeur gebruik maken van complexe structures. De opbouw van deze structures moet vooraf in de `.x`-file gespecificeerd worden. Bij de aanroep van een remote procedure—RPCGEN-stub—moeten meer foutcontroles worden uitgevoerd dan bij een lokale procedureaanroep.

Het gebruik van RPCGEN-stubs is niet transparant voor programmeurs. Het concept 'stub' is door Birrell & Nelson geïntroduceerd om netwerkcommunicatiedetails af te schermen voor de programmeur (§ 3.4.3). De stubs zijn te beschouwen als een schil om de details heen. Deze schil of vertaalslag biedt alleen high level RPC-communicatieprimitieven aan de programmeur aan. De *low-level communicatieprimitieven* zijn niet meer zichtbaar.

Door nu een extra schil om de gebruiksonvriendelijke RPCGEN-stubs heen te leggen kunnen de details van parameterstructuregebruik voor de programmeur verhuld worden. Deze extra vertaalslag heeft tot doel een procedureabstractie te bieden die meer overeen komt met de conventionele lokale procedure-abstractie van de taal C. Volgens de eis van transparantie zou een stub net zo moeten kunnen worden aangeroepen als een gewone lokale procedure. De extra schil bestaat uit stubs die op de gewone wijze worden aangeroepen en dan de gebruiksonvriendelijke RPCGEN-stubs aanroepen.

De stubs van de Sun omgeving worden door een codegenerator geproduceerd. De extra stubs van de extra schil worden door een nieuwe codegenerator geproduceerd. Dit kan door op basis van standaard C programma's input files voor RPCGEN te genereren. Een programma dat dit doet vertaalt C procedure declaraties naar de RPCGEN taal. Het is een preprocessor voor RPCGEN. Deze preprocessor wordt verder 'Pre RPCGEN Processor' of PRP genoemd. Door PRP hoeft de applicatieprogrammeur zich niet meer te bekommeren om Sun-specifieke details als `.x`-files, `clnt_create()`'s of RPC systeemfoutafhandeling.

Het PRP-model beschrijft een programma dat een software laag tussen de applicatiecode en RPCGEN genereert. PRP leest in principe standaard C.[1] Zie voor een beschrijving van C [Kernighan 2].[2] De output van PRP wordt door RPCGEN gelezen. Van groot belang voor PRP is hoe de input van

[1] In § 6.1 zijn PRP's afwijkingen van de C syntax beschreven.

[2] Hierbij zij aangetekend dat ondersteuning van ANSI C het doel is, niet het oorspronkelijke K&R C. In [Kernighan 2, tweede druk] wordt aangeraden C programma's—indien de compiler dit ondersteunt—volgens de ANSI norm te formuleren. Dit omdat met ANSI C de compiler meer fouten kan opsporen dan in de oorspronkelijke C versie. In ANSI C worden functiedeclaraties en -definities met behulp van zogenaamde 'prototypes' op consistentie gecheckt. Voorheen was dit een bron van veel fouten.

RPCGEN—de output van PRP—er uit moet zien. Zie voor een beschrijving van de RPCGEN syntax [Sun 1] en § 4.1.2.

globale eisen PRP

De syntactische verschillen tussen de RPCGEN input en standaard C zijn de volgende [Kernighan 2, Sun 1]:

- De interface tussen client en server moet voor RPCGEN in een `.x`-file worden gespecificeerd. Bij een lokale procedureaanroep is een proceduredeclaratie of -prototype voldoende. *Alle informatie van de parametertypedeclaraties uit de interfacedefinitie is ook in de gewone C programmacode te vinden.*

- Meerdere parameters moeten gegroepeerd naar input of output in structures worden gedeclareerd. In C is wordt niet het onderscheidt input/output maar value/reference gehanteerd. Op pagina 61 is beschreven hoe call by reference in RPC-systemen past.

- De programmeur moet voor gebruik van een server eerst contact maken met de servermachine door middel van een `clnt_create()` aanroep. Men moet de netwerknaam van de machine specificeren.

- Na terugkomst moet het resultaat van de RPC op RPC-systeemfouten getest worden voordat de gebruikelijke foutcontrole van de procedure mag plaatsvinden.

PRP moet een vertaalslag genereren voor deze punten. De inputspecificatie voor PRP is in [Kernighan 2] beschreven. De outputspecificatie is op in [Sun 1] beschreven (zie pagina 81 en 149 voor voorbeelden).

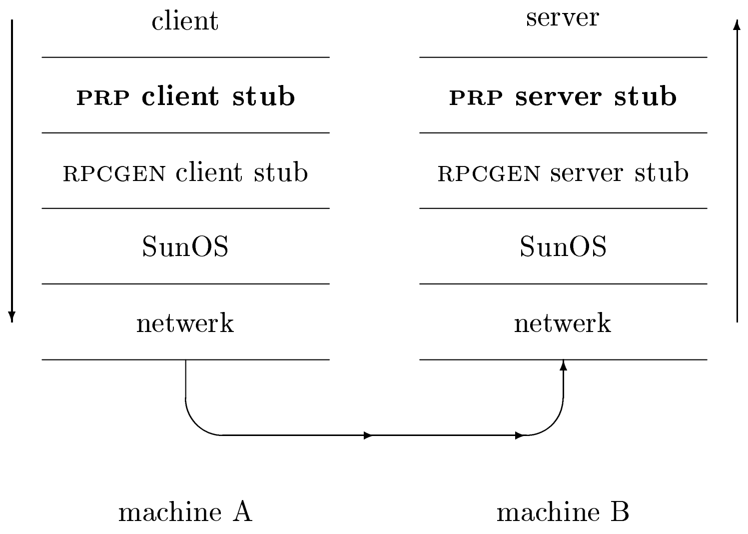
5.1.2 Plaatsbepaling van PRP

In figuur 5.1 is de PRP-laag aan de SUN RPC omgeving (figuur 4.1) toegevoegd. De 'client' en de 'server' file—de applicatieprogrammatuur—zijn hier idealiter *standaard C files.* In de RPCGEN situatie van figuur 4.1 moet de programmeur in de code van client en server RPC's anders behandelen dan lokale procedureaanroepen.

In figuur 5.1 is aangegeven dat de door PRP genereerde softwarelaag uit stubs bestaat. De files met de extra stubs bevatten voor elke remote procedure de vertaling van standaard C procedure aanroepen naar RPCGEN-stub procedure aanroepen. Daarnaast bevat de client stub initialisatie- en RPC-systeemfout afhandelingscode.

In figuur 5.1 wordt conceptueel de plaats van de PRP softwarelaag weergegeven. In deze figuur is met pijlen de control flow van de procedure aanroep weergegeven. Deze is gecompliceerder dan in een situatie met één verzameling stubs (figuur 3.3 en 4.1). De client roept volgens de standaard C syntax de PRP-client-stub aan. De PRP-client-stub roept volgens de RPCGEN

Figuur 5.1: PRP in de RPC omgeving



regels de RPCGEN-client-stub aan. Deze verzendt een opdracht via het RPC-systeem aan de RPCGEN-server-stub. De RPCGEN-server-stub roept de PRP-server-stub aan. Deze roept volgens de standaard C syntax de gebruikers serverfunctie aan.

Als de serverfunctie klaar is wordt deze weg in omgekeerde volgorde doorlopen om de resultaten terug te sturen naar de aanroeper.

Voor de programmeur is deze complexe gang van zaken niet zichtbaar. Deze roept een PRP-stub op dezelfde wijze aan als een lokale procedure. Met de interne werking van de stubs behoeft men zich niet bezig te houden. De mate van transparantie van een RPC èn de complexiteit van het systeem zijn door het gebruik van de extra stubs vergroot.

5.1.3 Output files

De softwarelaag die PRP genereert bestaat naast de al genoemde `.x` file uit:

- de bewerkte applicatiefiles (zie § 5.2.2)
- twee extra stub files

In figuur 5.2 wordt een voorbeeld gegeven van de files waaruit een eenvoudige gedistribueerde applicatie bestaat. Er is uit af te lezen hoe de software laag die PRP genereert past in de compilatie volgorde van RPCGEN. Als voorbeeld wordt aan de hand van een fictieve applicatie beschreven welke files door welke processor gebruikt worden. De applicatie bestaat uit twee files: `spreadsheet.c` en `dbms.c`. De client kant is een gebruikersinterface in de vorm van een spreadsheet. Aan de server kant zit een database back-end dat queries en updates in opdracht van de client uitvoert.

Om deze applicatie te compileren worden de twee files eerst door PRP bewerkt. PRP analyseert de parameter declaraties van client en server en genereert een XDR file (de `.x`-file `PRP_intf.x`). In de client en server PRP-stub staan de parameter in- en uitpak- en initialisatie- en foutafhandelingscode voor de RPCGEN-stubs (`PRP_c_stub.c` en `PRP_s_stub.c`). Parameters worden tweemaal gemarshalled: de PRP-stub zet ze in de RPCGEN input structure, de RPCGEN-stub maakt ze verzendklaar. De applicatie wordt verder als een gewone RPCGEN applicatie gecompileerd.

Dit plaatje is bedoeld om de plaats van PRP binnen de applicatieontwikkelomgeving van Sun weer te geven, met name het verband met RPCGEN. Bij de daadwerkelijke ontwikkeling van gedistribueerde applicaties kan het compilatieproces door het UNIX commando `make` worden geautomatiseerd zodat de programmeur zich niet met dit soort details hoeft bezig te houden.

5.2 Opbouw van PRP

Hiervoor zijn taken en plaats van het PRP-model globaal omschreven. De volgende paragrafen gaan over de werking van PRP zèlf. Niet over hoe de stubs werken, maar hoe het *genereren* van de stubs werkt. Eerst wordt globaal uiteengezet hoe de vertaalslag door PRP gegenereerd wordt. § 5.4 en appendix A gaan op de details in.

5.2.1 Systeemontwerp PRP

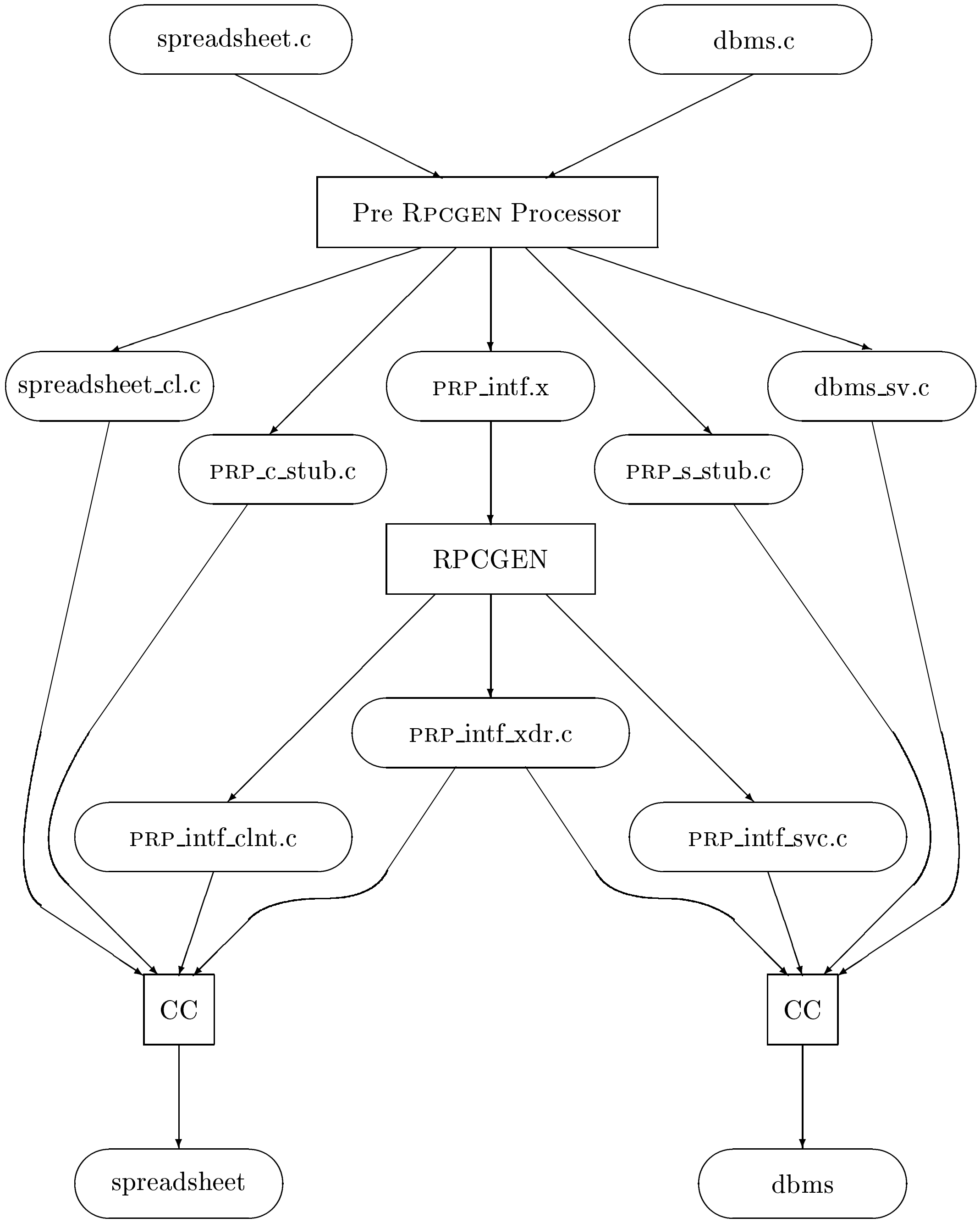
De analyse en het ontwerp van het PRP programma zijn vanuit de output die het moet genereren gestart (figuur 5.3). Door de aard van de problematiek is een gegevensgeoriënteerd analyseproces toegepast [DeMarco, Van 't Veld]. De aard van de problematiek is: het enige dat *vast* ligt is de *output*specificatie. Er is een *sterke wens* om als *input* ongewijzigd-C te ondersteunen.

De output van PRP is gedetailleerd omschreven in [Sun 1]. Wijzigingen in de output specificatie van het programma leiden tot disfunctioneren van PRP. Bij het vaststellen van de input bestaat wel enige ruimte om eventuele problemen op te lossen. In dat geval wordt de inputsyntax veranderd. De transparantie van het geheel wordt door elke afwijking van C minder. In § 6.1 zijn de afwijkingen ten opzichte van ANSI C beschreven.

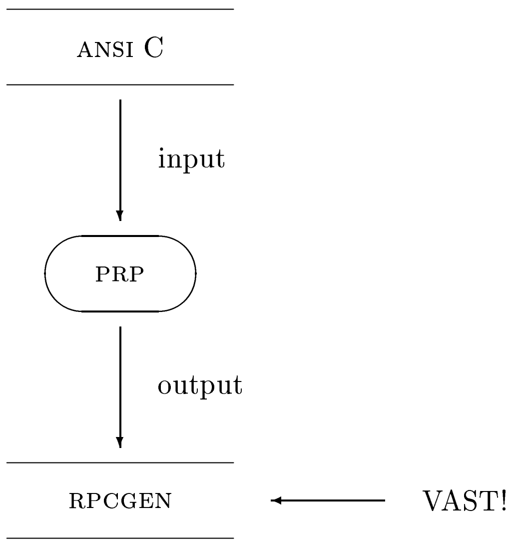
Op basis van de gedetailleerde outputspecificatie is een globaler schema ontworpen waarin (a) de verschillende parametertypen en (b) de categorieën te genereren code met elkaar in verband worden gebracht (§ A.2.2). Dit is een bottom-up benadering.

Nadat de gegevens (output, input en globaal schema) vast stonden zijn de modules ontworpen. Hier werd eerst globaal vastgesteld welke functionaliteit een vertaalslaggenerator als PRP moet bieden (figuur 5.4). PRP kent een eenvoudige module structuur. Later zijn op basis van de detail schema's van

Figuur 5.2: PRP files



Figuur 5.3: Input en output van PRP



§ A.2.2 en appendix A de modules ingevuld en uitgeprogrammeerd. Dit is een top-down benadering.

Resumerend: analyse en ontwerp van PRP zijn vanuit de outputspecificatie begonnen. Er is een gegevensgericht analyseproces toegepast. De gegevensanalyse is bottom-up geweest. Hierna zijn de programmamodules ontworpen. De modules zijn top-down ontworpen.

Vervolgens is getracht een werkend programma met de minimaal benodigde functionaliteit te schrijven. Het programma is daarna via de methode van stapsgewijze verfijning verder ontwikkeld. Door het toevoegen van ondersteuning van meer taalelementen zoals complexere parameters is het doel standaard C als input te accepteren steeds dichter benaderd (zie § 6.3).

5.2.2 Functies PRP-laag

De uitgangspunten van het systeemontwerp worden gevormd door de uitvoer die het programma moet genereren. In deze paragraaf worden de verschillende categorieën te genereren code beschreven. § A.3.1 behandelt de invulling hiervan en detail.

PRP leest door de applicatieprogrammeur geschreven files in. De uitvoer van PRP is een `.x` file, een tweetal files met de PRP stubs en de aangepaste applicatiefiles (zie figuur 5.2). De functies die deze uitvoer vervult wordt hieronder puntsgewijs behandeld.

- De `.x` file bevat allereerst de parameter declaraties voor elke remote procedure. Daarnaast worden de remote procedures in het versienummergedeelte opgesomd. Zie pagina 82 voor een voorbeeld.

- De PRP-stubs vervullen meerdere taken. Elke remote procedure heeft een eigen stel stubs. De clientstub is de grootste stub. De functies van de clientstub zijn achtereenvolgens:

  - initialiseer het RPC systeem: bepaal waar de server procedure zich bevindt en voer de `clnt_create()` aanroep uit.
  - vul de RPCGEN inputparameter structure met de eigenlijke parameters.
  - roep de RPCGEN-clientstub aan.
  - ga na of het resultaat de `NULL`-pointer is. Zo ja, geef een foutmelding.
  - pak de RPCGEN outputparameterstructure uit en vul de eigenlijke uitvoer parameters alsmede de returnwaarde van de procedure.

  De PRP-serverstub is eenvoudiger. Deze kent de volgende functies:

  - pak de RPCGEN inputparameterstructure uit en vul de inputparametervariabelen van de serverprocedure die de programmeur geschreven heeft.
  - roep de gebruikers serverprocedure aan volgens de standaard C syntax.
  - vul de RPCGEN outputparameterstructure met de output parameters en de returnwaarde van de gebruikersprocedure.

- De applicatieprogrammafiles van de gebruiker moeten op een paar punten gewijzigd worden. De namen van de oorspronkelijke gebruikers procedures moeten veranderd worden in de namen van de desbetreffende PRP stubs. Mocht de gebruiker de netwerknaam van een servermachine bij een RPC specificeren, dan moet deze aan de PRP client stub worden doorgegeven voor de initialisatie.

5.2.3 Structuur PRP

De structuur van PRP is eenvoudig (zie figuur 5.4). Het programma leest de invoer en genereert de uitvoer. De invoer is standaard C. De uitvoer is een RPCGEN applicatie die door RPCGEN en de C compiler verder verwerkt worden.

Eerst worden de input files geanalyseerd. Op basis van deze analyse worden de `.x`-file en de PRP-stubs gegenereerd. Dan worden de namen van de PRP-stubs in de gebruikersfiles gezet. Tenslotte wordt een `makefile` gemaakt (zie ook pagina 79) om de verzameling files tot een applicatie te compileren.

datastructuren

Tijdens de analyse worden de gegevens over de remote procedures in datastructuren opgeslagen. PRP bevat drie lijsten: een lijst met de filenamen waar de applicatie uit bestaat, een lijst met de namen van de remote procedures en een lijst met parameters die bij die remote procedures horen. In deze lijsten zijn alle gegevens opgenomen die nodig zijn om de output te genereren.

De *filelijst* bevat voor elke inputfile de oorspronkelijke filenaam, de naam van tussentijdse versies van de file, en de output filenaam.

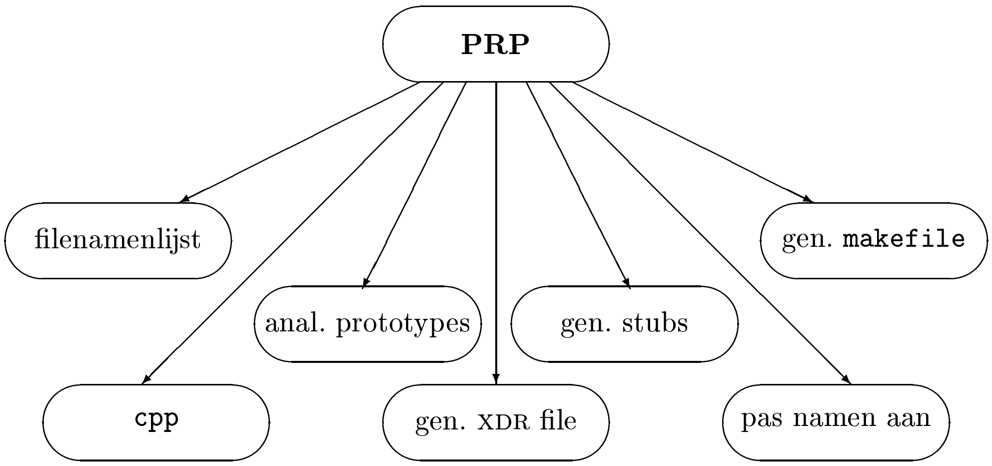
De *remote procedure lijst* bevat voor elke routine de naam, het type van de returnwaarde en een intern volgnummer.

De *parameterlijst* tenslotte bevat voor elke parameter de naam van de routine waar hij bij hoort, de parameternaam zelf, het type, het level-of-indirection (is het een pointer?), of het een input of output parameter is, eventueel de array omvang en een intern volgnummer.

procesgang

De uitvoervolgorde van de programmamodulen is als volgt:

Figuur 5.4: Structuur van het PRP programma



1. Sla in de filenaamlijst op uit welke files de applicatie bestaat. De headerfiles worden niet als aparte files, maar als onderdeel van de `.c` files beschouwd.

2. Preprocess de files met de C Preprocessor (`cpp`). Er staan nu geen `#include` en `#define` directives meer in de applicatiefiles. De headerfiles zijn nu daadwerkelijk onderdeel van de `.c` files.

3. Analyseer de prototypes van de remote procedures. De procedurelijst en de parameterlijst worden hier gevuld. De analyse vindt plaats met YACC, de UNIX parser [Kernighan 1].

4. Genereer de `.x`-file voor RPCGEN.

5. Genereer de PRP-stubs. Zie § 5.2.2 voor een beschrijving van hun functionaliteit.

6. Pas de gebruikerfiles aan aan de veranderde functienamen.

7. Maak een `makefile`.

In figuur 5.4 is de samenhang van de PRP programma modulen grafisch weergegeven.

YACC

Vaak wordt bij programmatuurontwikkeling voor UNIX systemen gebruik gemaakt van formele taalanalyse. Een *parser* is een programma dat de structuur van zijn invoer analyseert om bepaalde acties uit te voeren. Parsing van een rij symbolen van een bepaalde grammatica bestaat uit het herkennen van deze symbolen en het samenstellen van de *syntax tree* voor déze rij symbolen [Watt 2]. De syntax tree kan door de rest van het programma gebruikt worden, bijvoorbeeld om een taal voor een lagere virtuele machine te genereren.

Het programma `yacc` genereert op basis van een grammatica een parser in C [Kernighan 1]. YACC converteert een context-vrije grammatica (in Backus-Naur Form) in een verzameling tabellen voor een eenvoudige automaat die een LR(1)[3] parsing algoritme uitvoert [Sun 2, Johnsonbaugh]. Deze automaat, een C procedure, kan de programmeur in het programma gebruiken om de syntax van een programmeertaal te analyseren.

In [Kernighan 2] is de grammatica van C gegeven. Met behulp van YACC kan de syntaxanalysemodule van een compiler of van een programma als `lint` gegenereerd worden. PRP gebruikt een YACC-parser om procedure- en parameterdeclaraties te ontleden en in lijsten op te slaan.

5.3 Ontwerpproblemen

Na de omgeving en structuur van PRP globaal behandeld te hebben wordt hier op de uitwerking van het raamwerk ingegaan. PRP is ontworpen met als doel de transparantie voor de programmeur van een Sun RPC op een aantal specifieke punten (punt 5, 6 en 7 van pagina 93) te verhogen. De syntax van programmamodules voor gedistribueerde applicaties moet zo min mogelijk af wijken van de taalconstructies van sequentiële applicaties. Een RPC moet net zo geformuleerd kunnen worden als een lokale procedureaanroep—met

[3] Een veelgebruikte klasse syntaxanalyse-algoritmes. LR(1) betekent: „Lees het meest **L**inkse symbool, produceer het meest **R**echtse symbool, en lees **1** symbool teveel vooruit."

evenveel parameters en zonder extra foutcontroles of parameter in- en uitpakcode.

5.3.1 Beperkingen taal

De ontwerpproblemen die optreden komen voort uit het doel standaard C als input syntax te ondersteunen. De initiële eis is:

$$\text{syntax}(\textsc{rpc}) = \text{syntax}(\text{lokale aanroep})$$

De syntax die PRP accepteert moet zo weinig mogelijk van ANSI C afwijken [Kernighan 2].

Een aantal elementen van de inputsyntax zijn niet eenduidig naar de outputsyntax te vertalen. De RPCGEN omgeving staat het niet toe om de gehele syntax van de taal C te ondersteunen. Hieronder volgt een opsomming van de restricties die PRP de programmeur oplegt. In de volgende paragraaf wordt de uitwerking van onderstaande problemen voor PRP gepresenteerd.

variabele parameterlijst

In C is het mogelijk om een functie met een variabel aantal parameters te declareren. De `printf(fmt, ...);` proceduredeclaratie is hier een voorbeeld van. De drie puntjes geven aan dat het aantal parameters bij elke aanroep anders kan zijn. Ze geven aan dat vanaf dat punt een at compile-time onbekend aantal parameters doorgegeven gaat worden.

Voor RPCGEN moeten alle parameters in een structure gedeclareerd worden. Er is hier geen mogelijkheid om een variabel aantal structure members te declareren. De parameters van RPCGEN-stubs staan at compile-time vast. Bij gebruik van RPCGEN (en PRP) zijn remote procedures met een variabel aantal parameters niet toegestaan.

remote of lokaal

Een RPC wordt anders behandeld door het operating system dan een lokale procedure aanroep. Voor een RPC moeten communicatieprimitieven aangeroepen worden, voor een lokale aanroep niet. Het systeem moet tijdens de compilatie weten welke procedure lokaal is en welke remote. Alleen voor remote procedures moet RPCGEN code gegenereerd worden. PRP moet op de een of andere manier weten welke procedures remote zijn en welke niet. De programmeur moet dit specificeren.

servernaam

De client zal op een of andere wijze de server moeten identificeren zodat het communicatie subsysteem het verzoek kan versturen. Voor sommige toepassingen kan anonieme adressering—broadcast-RPC—worden toegepast. Voor complexere applicaties die uit meerdere servers bestaan moet een individuele server geïdentificeerd kunnen worden. Wanneer men geen gebruik kan maken van broadcast-RPC moet de naam van de servermachine—het netwerkadres—in de RPC syntax worden opgenomen.

call by value/reference

Er is een verschil in de semantiek van *by value* en *by reference* doorgegeven parameters. Een *value* parameter wordt alleen bij aanroep gebruikt om de formele parameter te initialiseren. Bij een *reference* parameter worden de bewerkingen van de procedure op de actuele parameter zelf uitgevoerd [Van der Sluis].

In RPC termen wordt de waarde van een reference parameter na afloop van de functie teruggegeven aan de actuele parameter. In de literatuur wordt dit *call by copy/restore* genoemd [Tanenbaum 3, Coulouris, Watt 1] (zie ook punt c in § 3.4.2).

Voor een RPC systeem is het noodzakelijk om te weten welke parameters door de programmeur als value of als reference gebruikt worden. Van een reference parameter moet de waarde teruggegeven worden aan de aanroeper, bij een value parameter niet.

C kent alleen value parameters. Call by reference wordt in C uitgevoerd door een pointer naar de waarde by value door te geven. De procedure kan door *dereferencing* van de pointer zo de actuele parameter bereiken. Hieronder worden een aantal ideeën genoemd hoe PRP value van reference parameters zou kunnen onderscheiden.

- *At run-time* In C worden parameters *by reference* doorgegeven door hun geheugen*adres* in plaats van hun waarde aan de procedure door te geven (een pointer naar de waarde). Soms echter worden pointers ook als echte valueparameter gebruikt. Het verschil is dat de waarde van een value parameter kan veranderen tijdens de loop van de procedure. De pointer die bij een reference parameter gebruikt wordt blijft onveranderd.

  Door tijdens de uitvoering van het programma te kijken of een formele parameter verandert, is vast te stellen of een parameter kennelijk als value of als reference gebruikt wordt. Als de waarde van een pointerparameter onveranderd is moet de waarde waar hij naar wijst door het RPC-systeem aan de aanroeper terug worden gegeven.

  Om dit at run-time vast te stellen zou extra code aan het einde van een procedure moeten worden uitgevoerd. Een nadeel is dat er meer overhead gecreëerd wordt. Het programma wordt enigszins trager en groter.

  Deze oplossing wordt gecompliceerd door het feit dat er een *alias* van de formele parameter gebruikt kan worden. Een alias is een nieuwe pointervariabele die naar dezelfde geheugenlokatie wijst. Door syntactische analyse moeten alle aliassen van de formele parameter geïdentificeerd worden. Ze moeten dan allemaal at run-time op verandering gecheckt worden.

Deze werkwijze druist in tegen het grondbeginsel van statically typed languages [Watt 1]. Bij deze groep talen, waar C deel van uitmaakt, worden typechecks *at compile-time* uitgevoerd. Zo worden fouten zo vroeg mogelijk opgespoord. Er wordt zo min mogelijk overhead at run-time gecreëerd.

- *At compile-time* De waarde van een pointer parameter verandert niet als hij als reference parameter gebruikt wordt. Uitgaande van dit gegeven kan men een syntactische analyse uitvoeren, die voor elke pointer parameter onderzoekt of er in de functie ook een waarde aan toegewezen wordt (assign expressions).

  Ook bij deze oplossing moet aan aliassen gedacht worden. Wanneer een formele parameter zelf als actuele parameter voor een andere functie fungeert zal ook die functie geanalyseerd moeten worden. Anders kan niet met zekerheid vastgesteld worden of de waarde veranderd kan worden of niet.

  Deze weg loopt dood wanneer de programmeur gebruik maakt van bibliotheekroutines waarvan de source code niet beschikbaar is.

- *'in' en 'out'* Een oplossing is om bij de declaratie van remote procedures de programmeur te laten vermelden of het om een input, output of input/output parameter gaat.

  Sommige C programmeurs vermelden in het commentaar van hun programma bij elke parameter of het een input, output of input/output parameter is. In de taal Pascal wordt expliciet syntactisch onderscheid gemaakt tussen value en reference parameters. In de taal Ada wordt onderscheid gemaakt tussen input en output parameters [Watt 1]. Het onderscheid tussen 'in' en 'out' is in de syntax doorgevoerd. Een *value* parameter is een *input* parameter. Een *reference* parameter komt overeen met een *input/output* parameter.

  Het voordeel van deze oplossing is dat het een eenvoudige en afdoende oplossing is, nadeel is dat wordt afgeweken van standaard C.

5.3.2 Ontwerpkeuzes

Bij de hiervoor genoemde problemen zijn de volgende keuzes gemaakt.[4]

remote of lokaal

Het onderscheid tussen remote en lokale procedures wordt bij RPCGEN in de definitiefile aangegeven. Elke remote procedure moet door de programmeur in de interfacedefinitiefile worden opgesomd. Voor de taal C is een elegantere oplossing denkbaar. Analoog aan het C keyword `extern` kan het keyword `remote` worden ingevoerd. `Extern` staat onder andere voor de proceduredeclaratie in de file waar de procedure wordt aangeroepen, maar

[4] In § A.2.2 wordt in een tweetal tabellen de PRP-syntax gedetailleerd behandeld.

niet gedefinieerd. `Remote` is syntactisch equivalent aan `extern`, zij het dat `extern` ook bij variabelen mogelijk is en `remote` alleen bij proceduredeclaraties.

Een remote procedure is intuïtief te beschouwen als een `extern` procedure die een slag externer is. De procedure bevindt zich niet alleen in een andere module, maar in een ander programma op een andere machine.

PRP onderscheidt *client* van *server* files door te kijken of de remote procedures in de file worden *aangeroepen* of *gedefinieerd*. `Remote` wordt door PRP gebruikt om te signaleren voor welke proceduredefinities (server) en welke aanroepen (client) code gegenereerd moet worden.

Deze oplossing is uit het oogpunt van transparantie voor de programmeur beter dan een aparte interfacedefinitie. De werkwijze sluit aan bij de in C gebruikelijke manier van werken.

servernaam

Het probleem van de serveradressering heeft twee varianten:

1. De client heeft *één* server per remote procedure. De identificatie zou door het systeem zelf kunnen worden uitgevoerd volgens het algoritme: „Als er een server actief is in het netwerk met de gewenste procedure, gebruik deze dan. Zo niet, geef een foutmelding." Dit kan met een broadcast-call geïmplementeerd worden. Broadcasts werken in verband met congestie vraagstukken alleen op het lokale (gedeelte van een) netwerk—LAN [Sun 2].

   RPC's zullen om redenen van performance vaak in een LAN omgeving geïmplementeerd worden. Broadcast-calls zijn geschikt voor gebruik in gedistribueerde applicaties in een LAN.

   Bij een broadcast-call specificeert de programmeur geen netwerknaam. De syntax wijkt op dit punt niet af van een lokale aanroep.

2. De client gebruikt *meerdere* servers. Als het niet nodig is een specifieke server te selecteren voor een aanroep kan men een broadcast-RPC gebruiken. De server die het eerst de aanroep beantwoord wordt dan gebruikt. Voor sommige toepassingen kan het nodig zijn dat de programmeur per RPC aangeeft welke server de RPC moet uitvoeren.[5]

De aanroepen zouden er als `functie(para1, para2)@netwerknaam;` kunnen uitzien:

- `bereken(matrix1, matrix2)@"sus.eur.nl";`
- `bereken(matrix1, matrix2)@argv[1];`
- (broadcast) `bereken(matrix1, matrix2);`

[5] Zie bijvoorbeeld [Chin] voor een bankapplicatie met meerdere filialen.

Het van server wisselen voor een procedure is mogelijk door een andere netwerknaam te specificeren.[6]

Bij PRP betekent het afwezig zijn van een `@netwerknaam`-achtige constructie bij de aanroep van een remote procedure het in werking treden van het broadcast mechanisme. De snelst reagerende server wordt gekozen.

Naar analogie van Argus [Liskov] is voor het 'at' (@) symbool gekozen, niet voor een extra parameter voor de RPC. Het @ symbool is nieuw voor de taal C-syntax. Voor procedures is het concept *plaats* nieuw. @ kan niet verward worden met bestaande C constructies of concepten. Een extra parameter zou gemakkelijk aangezien kunnen worden voor een reguliere parameter. Voor een nieuw concept wordt een nieuw syntaxelement geïntroduceerd.

call by value/reference

Om de parameterrichting aan te geven is voor de laatste optie gekozen: toevoeging van 'in' en 'out' bij de procedure declaratie. Deze oplossing levert een heldere syntax op die voor programmeurs eenvoudig te begrijpen is. Om aan te sluiten bij de gangbare werkwijze in C [Kernighan 2] worden, wanneer de programmeur geen `in` of `out` vermeldt, pointer parameters als input/output beschouwd en andere parameters als input. Daar call by value van pointerparameters in een RPC omgeving zinloos is, is een ondubbelzinnige interpretatie van pointerdeclaraties mogelijk.

De andere opties brengen implementatieproblemen met zich mee. Optie drie is eenvoudig te implementeren omdat RPCGEN ook het onderscheid tussen input- en outputparameter kent. Andere procedurele talen (Ada) maken ook onderscheid in input en outputparameters. Een laatste argument is dat implementatie in een RPC omgeving van call by reference semantische problemen met zich mee brengt, terwijl de semantiek van input- en outputparameters goed aansluit bij gescheiden geheugen-systemen.

5.3.3 Nieuwe taalelementen

Voor eenvoudige toepassingen is het mogelijk een gedistribueerde applicatie met PRP te schrijven waarbij de enige afwijking van standaard C het keyword `remote` voor de prototypes van remote procedures is.

Deze applicatie maakt dan gebruik van broadcast-RPC en voor de richting van parameteroverdracht worden de defaultregels gebruikt. De defaultregels zijn in § A.2.2 beschreven.

Wanneer de programmeur meer controle over serverkeuze en parameteroverdracht nodig heeft moet men bij PRP meer nieuwe syntaxelementen gebruiken: '@' en 'in' en 'out'.

[6] In het huidige prototype is dit nog niet mogelijk. Verder moet bij het prototype een tweede @ ter afsluiting van de servernaam gespecificeerd worden. Zie appendix A.

5.4 Details PRP

Deze paragraaf gaat in op de details van de vertaalslag C → RPCGEN. Het meest complexe vraagstuk van het PRP ontwerp is de grote diversiteit die de taal C biedt voor het declareren van parameters. Bij het genereren van de initialisatie van de RPCGEN-stubs met broadcast en `clnt_create()` en de foutafhandeling zijn er veel minder keuzemogelijkheden.

Om de parametertypedeclararties van de `.x` file en de PRP-stubs te kunnen genereren moet PRP de declaraties van `remote` procedures gedetailleerd analyseren. Deze syntaxanalyse wordt door een YACC-parser op basis van de grammatica van C uit [Kernighan 2] uitgevoerd. Na de analyse moeten de verschillende outputfiles gegenereerd worden. Hieronder wordt ingegaan op de analyse van de proceduredeclaraties en de mogelijkheden bij de code generatie.

5.4.1 Analyse prototypes

De programmeur moet de prototypes van de remote procedures in de applicatieprogrammamodules van het keyword `remote` voorzien. De syntaxanalysemodule van PRP zoekt met behulp van de door YACC gegenereerde parser [Kernighan 1] die prototypes op.

Op basis van de grammatica van de taal C worden de prototypes geanalyseerd. De procedurelijst en de parameterlijst (zie § 5.2.3) worden tijdens dit proces gevuld. Uit de prototypes is alle informatie af te leiden die PRP nodig heeft—zoals typedefs en structure-declaraties—om de `.x`-file en de PRP-stubs te kunnen genereren. In de `.x`-file worden de RPCGEN input- en outputstructure gedeclareerd.

Voor deze declaratie moet PRP de datatypes van de parameters kennen. Om te weten in welke RPCGEN parameterstructure de parameters gedeclareerd moeten worden, moet PRP de *richting* van de parameter kennen. Deze wordt òf door de programmeur gespecificeerd, òf uit het *level of indirection*—aantal sterretjes—in de declaratie afgeleid. In § A.2.2 worden deze regels behandeld, in paragraaf 5.3.1 en 5.3.2 de theoretische achtergrond.

Door de parser worden niet alleen procedure*declaraties* herkend. Ook de *definitie* van een procedure die al eerder als remote was gedeclareerd wordt herkend. Op dat moment wordt bij de filenaam in de filenaamlijst aangetekend dat deze file een serverdefinitie bevat. Dit wordt gebruikt om vast te stellen welke naamswijzigingen in de respectievelijke gebruikersfiles moeten plaatsvinden. In *client*files moet de aanroep van de PRP-clientstub in de plaats komen van de gebruikers serverprocedureaanroep. In *server*files verdwijnt het oude prototype, en moet er een prototype van de gebruikers serverprocedure gegenereerd worden op basis van de aanroep in de client file. Op deze manier kunnen inconsistenties tussen aanroep en definitie van de serverprocedure door de C compiler aan de applicatieprogrammeur gerapporteerd worden. Zo blijft de foutopsporingsfunctie die de interfacedefinitie (zie § 3.4.3) van RPCGEN vervult bij PRP behouden.

5.4.2 Parameteroverdracht

Na de analyse van de proceduredeclaratie—de input—wordt nu de codegeneratie—de output—van PRP besproken.

Reference parameters worden via een call by copy/restore mechanisme gesimuleerd. De parameters moeten voor RPCGEN in input en output structures worden gegroepeerd. Bij de declaratie van het `remote` prototype moet (mag) de programmeur aangeven of het `in`, `out`, of `inout` parameters betreft. Laat men dit achterwege dan treedt een default mechanisme in werking. Op basis van datatype en level of indirection wordt zelf gekozen. Dit mechanisme is gebaseerd op [Kernighan 2]. Bij arrays en strings gaat [Kernighan 2] altijd uit van call by reference, bij andere datatypes hangt de keuze van het level of indirection af.

Parameters kunnen van een ster voorzien worden om, net als in standaard C, call by reference in plaats van call by value aan te geven. In een RPC omgeving is het doorgeven van pointers betekenisloos. Bij een RPC declaratie kan een sterretje niets anders dan 'by reference' aanduiden. De oorspronkelijke (standaard C) betekenis als declaratie van een pointerparameter *by value* bestaat niet meer omdat interprocespointers betekenisloos zijn in de Sun RPC omgeving. Bij PRP duidt een pointerdeclaratie aan dat de *actuele* parameter gewijzigd kan worden door de remote procedure.

Wanneer de programmeur niet accoord gaat met de defaultinterpretatie kan deze door het expliciet aanbrengen van `in`, `out`, en `inout` de precieze parameteroverdracht tot in detail bepalen. Bij een `in` parameter wordt de *restore* van de *call by copy/restore* niet uitgevoerd. Zo kan bij PRP een array by value worden doorgeven: de remote procedure kan het origineel niet wijzigen. Deze mogelijkheid is in standaard C niet aanwezig. Met `in`, `out`, en `inout` kan de efficiëntie van een RPC verhoogd worden door onnodige parameteroverdracht tussen procedures te voorkomen. In appendix A wordt hierop dieper ingegaan.

De Sun omgeving ondersteunt met RPCGEN het copy mechanism. Reference parameters kan men met input/output parameters benaderen.

5.4.3 RPCGEN datatypes

Binnen PRP worden simple-, string-, array- en structure-parametersoorten onderscheiden. Simple types zijn integers, characters en floats. Strings zijn `NULL`-terminated-character-arrays. Arrays en structures zijn in [Kernighan 2] beschreven.

Sun RPC gebruikt de External Data Representation (XDR) voor de overdracht van gegevens tussen client en server. Het omvormen van een bepaalde machine-afhankelijke representatie naar XDR vorm wordt *serializing* genoemd, het decoderen en lokaal opbouwen van de datastructuur *deserializing*. Als parameters over het netwerk gezonden worden moet de zender ze *serializen*. De ontvanger moet ze *deserializen*. RPCGEN genereert voor de RPCGEN-stubs de hiervoor benodigde aanroepen van XDR system-calls. Om de juiste aanroepen te kunnen genereren moet bij ingewikkelde datastructuren in de `.x`-file de opbouw van die datastructuur gespecificeerd worden. RPCGEN maakt dan procedures die de datastructuur ontleden in zijn samenstellende delen, die door de XDR bibliotheekroutines verwerkt kunnen worden.

De XDR-datatypes sluiten helaas niet perfect aan op die van C. Sommige typedeclaraties in de `.x`- file worden foutief geïnterpreteerd door RPCGEN [Sun 1]. De declaratie van een pointer-to-character wordt als een pointer naar één karakter opgevat. In het algemeen zal een C programmeur een `NULL`-terminated-string bedoeld hebben [Kernighan 2]. RPCGEN heeft dit opgelost door het keyword `string` in te voeren. Als de programmeur een variabele als `string` declareert wordt de XDR functie voor strings aangeroepen door de RPCGEN-stubs. In de rest van de C programma's van de applicatie moet gewoon `char *` gebruikt worden. Op deze problematiek wordt verder in appendix A ingegaan.

Een tweede verschil treedt op bij unions. XDR unions zien er anders uit dan C unions. De programmeur mag voor een RPCGEN RPC alleen de XDR variant gebruiken. RPCGEN vertaalt het voor C nieuwe datatype in een structure die de C union omsluit. De programmeur moet goed opletten dat men niet per ongeluk de union op de C manier gebruikt. Een voorbeeld [Sun 1, p. 61]:
In de `.x`-file staat:

```
union read_result switch (int errno) {
case 0:
    char data[1024];
default:
    void;
};
```

RPCGEN vertaalt dit voor de rest van de applicatie in:

```
struct read_result {
    int errno;
    union {
       char data[1024];
    } read_result_u;
};
```

Arrays zijn er in XDR in twee soorten: constante lengte en variabele lengte. Constante lengte arrays komen met statische C arrays overeen, variabele lengte kunnen voor dynamische—`malloc()`—arrays gebruikt worden. Variabele lengte arrays worden door RPCGEN vertaald naar een structure met een extra lengte member. Dit laatste veld moet door de programmeur at run-time gevuld worden. Een voorbeeld [Sun 1, p. 60]:
In de `.x`-file staat:

```
int heights<>;
```

RPCGEN vertaalt dit voor de rest van de applicatie in:

```
struct {
    unsigned int heights_len;
    int *heights_val;
} heights;
```

De te genereren code is van twee criteria afhankelijk. Het ene is *datatype*, het andere *plaats*.

Voor de verschillende groepen datatypes moet door PRP verschillende code gegenereerd worden. PRP maakt qua datatype onderscheid naar simple, string, array en structure. De code is qua *plaats* verschillend voor (a) de `.x`-file, (b) de PRP-clientstub en (c) de PRP-serverstub.

Binnen de PRP-*client*stub verschilt de code voor de proceduredeclaratie, het vullen van de RPCGEN inputparameter en het uitpakken van de RPCGEN outputparameter.

Binnen de PRP-*server*stub verschilt de code voor de declaratie van de lokale variabelen, het uitpakken van de RPCGEN inputparameter, het aanroepen van de gebruikers serverprocedure en het vullen van de RPCGEN outputparameter.

In appendix A wordt de code die bij deze verschillende typen en plaatsen hoort behandeld en met tabellen verduidelijkt. In de appendices zijn verder nog twee voorbeeldprogramma's opgenomen. Deze zijn bedoeld om het verschil tussen een RPCGEN-applicatie en een PRP applicatie te illustreren.

Het volgende hoofdstuk geeft een beoordeling van het PRP-model. Hier wordt bekeken in hoeverre de transparantie van de Sun omgeving verhoogd is door PRP.

Hoofdstuk 6

Beoordeling PRP

6.1 Kritiek

PRP moet een vertaalslag van ANSI C naar RPCGEN maken. De interface aan de outputkant wordt gevormd door RPCGEN en de C compiler. De output van PRP mag niet afwijken van de syntaxbeschrijvingen omdat het compilatieproces dan stokt. De inputkant van PRP is de applicatieprogrammeur. Deze is wel in staat om bij syntaxwijzigingen te blijven functioneren. De syntax die PRP accepteert wijkt op een aantal punten af van de C syntax. Correcte C programmafiles (`.c`-files) moeten aangepast worden voordat PRP ze kan verwerken.

Bij RPCGEN applicaties moet de programmeur een aparte interfacedefinitiefile schrijven. De `.c`-files bevatten geen nieuwe syntaxelementen. Ze worden direct door de C compiler verwerkt en niet eerst door een aparte processor—zoals bij PRP. Bij RPCGEN applicaties gebruikt de programmeur geen nieuwe *syntax*elementen maar nieuwe taal*constructies*. Een RPC wordt met bestaande syntaxelementen op een ongebruikelijke manier aangeroepen (zie pagina 93).

Bij PRP applicaties wordt een RPC—na declaratie met nieuwe syntaxelementen—op de gebruikelijke manier aangeroepen.

afwijkingen van standaard C

Een doelstelling van het PRP-ontwerp is om geen nieuwe syntaxelementen en geen nieuwe taalconstructies te introduceren—de input moet uit standaard C files bestaan. In de volgende lijst zijn de punten opgesomd waarop van deze doelstelling wordt afgeweken.

- *Remote* PRP onderkent de te bewerken procedures op basis van het keyword `remote` dat voor het prototype geplaatst moet worden. C kent het keyword `extern` dat aangeeft dat een procedure in een andere module gedefinieerd wordt. `Remote` wordt op dezelfde wijze gebruikt als `extern`. De gedachte achter `remote` is dat dit aangeeft dat een procedure niet in een andere module maar in een ander programma—op een andere machine—gedefinieerd wordt.

In C applicaties zal voor de desbetreffende procedures `extern` door `remote` moeten worden vervangen.

- *Servernaam* Remote procedures die niet met een broadcast-call aangeroepen kunnen worden moeten van een plaatsaanduiding voorzien worden. In RPCGEN applicaties is dit een extra parameter voor de RPC. PRP kent in navolging van sommige andere systemen de @-notatie. In § 5.3.2 is een argumentatie gegeven voor de introductie van dit nieuwe syntaxelement.

  Wanneer de programmeur de adresseringsmogelijkheid niet nodig heeft is het niet nodig de @-notatie te gebruiken. Heeft men voor de toepassing wel behoefte aan de mogelijkheid de adressering te kunnen beheersen, dan kan dit. Op dit punt is de PRP oplossing zowel transparant (indien gewenst) als flexibel (indien gewenst). Dit gaat ten koste van de efficiëntie: er wordt één extra RPC uitgevoerd om de servernaam te bepalen.

- *'in' en 'out'* Het default parametercopymechanisme—de richting van de parameters—dat PRP uit standaard C constructies afleidt is niet altijd de gewenste of meest efficiënte oplossing voor een bepaalde toepassing. In deze gevallen kan de programmeur gebruik maken van nieuwe syntaxelementen—`in`, `out` en `inout`—om het exacte gedrag van de applicatie te beheersen.

  Het copy mechanism sluit qua efficiëntie beter aan op gescheidengeheugen paradigma's dan het definitional mechanism (pagina 61). Net als RPCGEN vertaalt PRP de definitional mechanism aspecten van C—call by pointervalue—naar een copy mechanism.

Naast nieuwe syntaxelementen introduceert het PRP-model meer beperkingen voor de applicatieprogrammeur ten opzichte van C. RPC's mogen geen variabel aantal parameters hebben. Dit omdat de laag waarop PRP gebouwd is, RPCGEN, een constant aantal parameters veronderstelt.

In § 6.3.2 wordt uiteengezet welke problemen zich voordoen omdat C *weakly typed* is. Binnen het PRP-model is het niet mogelijk het datatype van een union, of het aantal elementen waar een pointer naar wijst vast te stellen op basis van de C-parameterdeclaraties.

Het union-probleem is op te lossen door te eisen dat in plaats van een union-parameter de programmeur een structure gebruikt—daarvan staat het type wel at compile-time vast. Het pointer-probleem heeft meer facetten (zie ook § A.4). Een oplossing hiervoor is om het maximale aantal elementen—net als in de XDR-file—tussen scherpe haken aan te laten geven. Op het gebruik van pointers voor RPC-parameters worden veel beperkingen gelegd. Door het ontbreken van een gemeenschappelijke adresruimte neemt de bruikbaarheid van pointervariabelen af (zie appendix A).

Applicaties die gebruik maken van het RPC's in de Sun omgeving kunnen geen gebruik maken van bepaalde mogelijkheden die in conventionele monolitische programma's vanzelf spreken. Het gebruik van interproces globale variabelen en interproces geheugenreferenties is vanwege de inefficiëntie niet geïmplementeerd. PRP applicaties kennen wat dit betreft dezelfde beperkingen als RPCGEN applicaties.

Omdat PRP een hogere mate van transparantie nastreeft dan RPCGEN vallen deze beperkingen bij PRP wellicht meer op dan bij RPCGEN. Om deze beperkingen op te heffen zou PRP gemeenschappelijk geheugen paradigma's moeten implementeren. Binnen de in deze scriptie beschreven opzet van PRP als eenvoudige preprocessor is dit niet mogelijk, nog afgezien van de efficiëntie problematiek die in § 3.3.2 en 46 beschreven is.

6.2 Beoordeling model

Hieronder wordt het PRP-model beoordeeld op de twee verzamelingen criteria voor gedistribueerde applicatieontwikkelomgevingen.

6.2.1 Gedistribueerdheid

PRP introduceert geen speciale gedistribueerde taal. PRP bouwt voort op de Sun RPCGEN omgeving. PRP is een extra stub- en interfacedefinitie generator.

parallellisme (1)

De eenheid van parallellisme en de manier van scheduling zijn ongewijzigd. Oordeel: $\circ$

interprocescommunicatie (2)

PRP bouwt voort op RPCGEN. PRP applicaties maken gebruik van het RPC-paradigma. Het is in principe mogelijk om de point-to-point messagespassing primitieven van de diepere UNIX lagen te gebruiken. Het door elkaar gebruiken van *high level* RPC-primitieven en *low level* message-passing wordt door [Sun 1] ontraden. Het leidt tot complexe applicaties. Bij de ontwikkeling van dit soort applicaties zullen de beperkingen van transparantie verhogende gereedschappen als RPCGEN en PRP de voordelen overtreffen.

PRP vergroot het aantal paradigma's niet (§ 3.4.5). Onveranderd oordeel: $+/\circ$

fouttolerantie (3)

PRP voegt geen elementen voor het verhogen van de bedrijfszekerheid van applicaties aan de Sun omgeving toe. Het aantal foutcontroles dat de programmeur moet uitvoeren wordt verminderd doordat de PRP-stubs foutcodes van de RPCGEN-stubs afvangen. De programmatekst van een applicatie wordt hierdoor overzichtelijker.

Onveranderd oordeel: $-$

integratie semantiek (4)

Het gebruik van globale variabelen en pointers is in PRP applicaties niet anders dan in RPCGEN applicaties. Hier blijft een groot verschil tussen gedistribueerde applicaties en monolitische programma's.

PRP doet geen pogingen de ondersteuning bij het opsporen van fouten die hiermee gemaakt worden te verbeteren.

Onveranderd oordeel: $\circ/-$[1]

complexiteit (5)

Voor een PRP applicatie hoeft de programmeur geen aparte interfacedefinitie te schrijven. Het is niet meer nodig om ingewikkelde parameterstructures samen te stellen en uiteen te rafelen zoals bij RPCGEN RPC's. Het aantal foutcontroles dat de programmeur moet uitvoeren is teruggebracht.

De complexiteit van de programmacode is verminderd. De transparantie van een RPC in de Sun omgeving is verhoogd. Het doel dat Birrell & Nelson op pagina 8 formuleerden was ervoor zorgen dat niet alleen specialisten maar ook gewone applicatieprogrammeurs programma's voor gedistribueerde systemen zouden kunnen schrijven. Dit doel wordt met PRP beter gerealiseerd dan met RPCGEN.

Het compilatieproces is door de introductie van een nieuwe processor complexer geworden. Vergelijk hiervoor de figuren 4.2 en 5.2.

Voor applicaties die andere eisen stellen dan een synchrone RPC biedt de Sun omgeving een slechte ondersteuning. PRP introduceert geen nieuwe communicatieparadigma's. Dit punt is ongewijzigd.

De complexiteit van de programmacode is verminderd. Oordeel: van $\circ$ naar $+$

typesecurity (6)

In PRP applicaties hoeft de programmeur geen pointers te gebruiken om parameterwaarden te benaderen. De code hiervoor wordt door PRP gegenereerd. Hier kan men geen fouten meer mee maken.

Prototypes van procedures maken parameter type-checks door de compiler mogelijk. PRP vereist van de programmeur dat deze prototypes in het programma gebruikt voor remote procedures. De consistentie-waarborgende functie van de interfacecompiler is door PRP overgenomen.

Run-time checks om typesecurity te waarborgen worden in PRP applicaties niet uitgevoerd.

De kans op het maken van fouten is enigszins afgenomen. Oordeel: van $-$ naar $\circ/-$

[1] De transparantie van een PRP RPC is groter dan van een RPCGEN RPC. Dit RPC-specifieke criterium is bij punt f op pagina 120 beschreven. Zie verder § A.4 voor een beschrijving van problemen die de taal C met zich mee brengt.

efficiëntie (7)

De efficiëntie van operating system en RPC-subsysteem zijn niet gewijzigd. Een PRP applicatie kent een extra stublaag bij de uitvoering van een RPC. Een PRP applicatie kent een grotere overhead dan een vergelijkbare RPCGEN applicatie. Deze overhead is klein in relatie tot de netwerkoverhead en de XDR conversie.

PRP biedt alleen het RPC-paradigma voor interprocescommunicatie aan, waardoor sommige toepassingen niet efficiënt geïmplementeerd kunnen worden.

De efficiëntie van het systeem is niet wezenlijk gewijzigd. Oordeel: $\circ$

overdraagbaarheid (8)

Voor de overdraagbaarheid van gedistribueerde Sun applicaties moeten XDR en RPCGEN op andere systemen beschikbaar zijn. PRP is met behulp van standaard UNIX gereedschappen (`yacc`, `cc` en `make`) ontwikkeld. Het PRP programma is eenvoudig naar andere systemen over te dragen. Platforms die RPCGEN applicaties ondersteunen zijn ook geschikt voor PRP applicaties.

Onveranderd oordeel: $+$

6.2.2 RPC

In deze paragraaf worden de elf specifieke RPC aspecten van de PRP applicatieontwikkelomgeving beoordeeld. Op basis van de beoordeling van de Sun omgeving worden hier de verbeteringen besproken.

a. globale variabelen

Het probleem dat interproces globale variabelen niet gesignaleerd worden bestaat ook bij PRP. Er is niet gepoogd dit probleem op te lossen.

Onveranderd oordeel: $-$

b. pointervariabelen

Pointervariabelen tussen verschillende adresruimtes zijn betekenisloos. XDR onderschept pogingen om dit toch te doen. In een logisch gedistribueerde omgeving—zonder (logisch) gemeenschappelijk geheugen—zijn pointervariabelen zinloos. In geen enkele taal voor zulke omgevingen kan deze semantiek geboden worden.

Recursieve datastructuren worden door de XDR standaard ondersteund. In het PRP model worden C typedeclaraties uit de programmatekst in de XDR-taal vertaald.

Het pointerprobleem wordt zowel in de RPCGEN als in de PRP situatie adequaat opgelost.

Onveranderd oordeel: $+$

c. parameters

RPCGEN-stubs kennen hooguit één inputparameter en één resultaatwaarde. Het gebruik van deze parameters is ingewikkeld en foutgevoelig.

PRP-stubs zijn op dit punt transparant voor de programmeur. De parametersyntax van een RPC kan gelijk zijn aan die van een lokale procedureaanroep. De programmeur hoeft geen `.x`-file meer te schrijven. De kans op fouten bij het gebruik van parameters is kleiner geworden.

Voor gedistribueerde systemen is een copy mechanism beter geschikt voor de parameteroverdracht dan een definitional mechanism. Bij RPCGEN moet de programmeur zelf call by reference met input- en outputparameters nabootsen. PRP doet een poging veelgebruikte call by reference constructies volgens het copy mechanism te modelleren (§ 5.4.2). De semantiek van call by reference kan in gedistribueerde systemen slechts benaderd worden (pagina 61).

Met PRP zijn meerdere parameters mogelijk. Oordeel: van $-$ naar $+$

d. faal-semantiek

PRP biedt dezelfde semantiek van een RPC als RPCGEN. PRP voegt geen bouwstenen om zero-or-one semantiek voor remote procedures mee te bouwen toe. Net als bij RPCGEN wordt dit aan de programmeur overgelaten. Door de specificatie van het gegevensoverdrachtprotocol kan de programmeur in principe voor at-most-once of at-least-once semantiek kiezen. In PRP is dit op TCP gefixeerd (zie ook punt h).

De programmeur moet bij at-most-once semantiek zelf eventuele side-effects van procedures bewaken. Voor het uitvoeren van herstelwerkzaamheden moeten systeemfouten aan de programmeur gerapporteerd worden. In het PRP model worden de foutcodes van de RPCGEN-stubs door de PRP-stubs afgevangen en eventueel in een globale variabele aan de applicatieprogrammeur doorgegeven.

De transparantie is toegenomen, de flexibiliteit is afgenomen. Er is geen keuzemogelijkheid meer. De faal-semantiek is at-most-once. Oordeel: van $+$ naar $\circ$

e. foutcode/exceptions

Net als RPCGEN voegt PRP geen exceptions toe aan C. Foutcondities worden door foutcodes van procedures gesignaleerd. Wanneer de programmeur hierom gevraagd heeft, vangen de PRP-stubs de foutcodes die de aanroepen van RPCGEN-stubs retourneren af, en zetten deze in een globale variabele om. De applicatieprogrammeur kan deze raadplegen om zelf de nodige actie te ondernemen.

Een alternatief is om bij elke fout het proces te stoppen.[2] Bij aanroep van een RPCGEN RPC moet men foutcontroles schrijven die bij een lokale

[2] Het PRP-prototype kent alleen de mogelijkheid om bij elke fout het proces te stoppen.

aanroep niet nodig zijn. Bij aanroep van een PRP RPC zijn deze ook afwezig. Op dit punt is de transparantie van een RPC vergroot.

Voor bepaalde toepassingen kan het wenselijk zijn om bij een fout herstelwerkzaamheden uit te voeren. Door het mechanisme met de globale variabele die de foutcode weergeeft biedt PRP dezelfde mate van flexibiliteit (indien gewenst) en meer transparantie (indien gewenst).

De alternatieve actie die door een PRP-stub bij een systeemfout ondernomen wordt is beëindiging van het proces. De reden voor deze keuze zou kunnen zijn dat een systeemfout bij een lokale procedureaanroep ook afbreken van het proces tot gevolg heeft. De semantiek van een PRP RPC komt op dit punt overeen met die van een lokale aanroep.

Een nadeel van deze werkwijze is dat dit niet voor alle toepassingen wenselijk is. Wanneer vergroting van de bedrijfszekerheid door replicatie het doel is zijn systeemfouten juist geen aanleiding voor afbreken van een proces. Bij idempotente procedures kan het opnieuw verzenden van een RPC een betere reactie op een systeemfout zijn. Identieke semantiek van remote en lokale procedures is niet altijd de juiste oplossing.

De transparantie is toegenomen. Oordeel: van $-$ naar $\circ$

f. integratie in programmeertaal

RPCGEN-stubs zijn ingewikkeld in het gebruik door de restrictie op het aantal parameters en de extra foutafhandeling bij elke aanroep. De taalconstructies die hierdoor nodig zijn maken dat het gebruik van een RPCGEN RPC niet transparant is voor de programmeur.

PRP-stubs worden op dezelfde wijze als een lokale procedure aangeroepen. Met uitzondering van de drie afwijkingen van de C syntax van § 6.1 is de formulering van de aanroep van een remote procedure gelijk aan het lokale equivalent.

Voor *volledige* transparantie van een RPC voor de programmeur is meer nodig dan syntactische gelijkenis. De gehele semantiek van de sequentiële taal moet dan in de gedistribueerde omgeving identiek zijn. Op pagina 46 is beschreven dat de semantiek van globale variabelen en geheugenreferenties niet in gedistribueerde omgevingen op efficiënte wijze ondersteund kan worden. Volledige transparantie is niet op efficiënte wijze te bereiken voor alle elementen van de taal C.

De aanroepsyntax van een PRP RPC komt in hoge mate met een lokale C aanroep overeen. Oordeel: van $-$ naar $+$

g. interfacedefinitie

De interfacedefinitie wordt door PRP gegenereerd. Het enige dat de applicatieprogrammeur moet doen is een applicatienummer voor de applicatie kiezen (zie § A.1.3). Alle andere functies van de interfacedefinitie worden door PRP uit de programmatekst van de applicatie gegenereerd.

De definitie van de interface is transparant voor de programmeur. Onveranderd oordeel: $+$

h. gegevensoverdrachtprotocol

PRP maakt gebruik van de door RPCGEN aangeboden communicatiemechanismen. Voor de gegevensrepresentatie wordt XDR gebruikt. Als communicatieprotocol kan de programmeur in principe uit UDP en TCP kiezen. Een nadeel van UDP is dat parameters niet groter dan 8 kilobyte mogen zijn.

Gezien de nadruk die in het PRP-model ligt op het verhogen van de transparantie wordt als protocol door de PRP-stubs altijd TCP geselecteerd. Op pagina 67 is beschreven dat voor at-most-once semantiek TCP geschikt is. Voor idempotente procedures—waar at-least-once/UDP geschikter voor is—zijn de PRP-stubs niet optimaal geschikt.

De protocolkeuze is transparant voor de programmeur. Dit gaat ten koste van de flexibiliteit. Voor sommige toepassingen kan het wenselijk zijn om UDP als protocol te kunnen kiezen.

De transparantie is toegenomen, de flexibiliteit is afgenomen. Er is geen keuzemogelijkheid meer. Oordeel: van $+$ naar $\circ$

i. netwerkadressering

PRP maakt voor adressering gebruik van de diensten die de portmapper aan RPCGEN levert. Voor applicaties waarin het niet nodig is servers direct te adresseren kan de programmeur de servernaam weglaten bij een RPC. Dit verhoogt de transparantie en verlaagt de flexibiliteit. De programmeur kan in dit geval kiezen tussen grotere transparantie of grotere flexibiliteit.

Onveranderd oordeel: $+$

j. beveiliging

De Sun omgeving biedt bouwstenen voor identificatie van client aan server. Het is aan de programmeur om hier al of niet gebruik van te maken.

De PRP-stubs maken geen gebruik van identificatieprimitieven. Dit wordt aan de programmeur overgelaten. Het doel van dit onderzoek is transparantieverhoging. De taak van PRP is een oplossing voor punt 5, 6 en 7 van § 4.2.3 te bieden. Beveiliging maakt hier geen onderdeel van uit.

PRP maakt geen gebruik van de mogelijkheden voor beveiliging. Oordeel: van $\circ$ naar $-$

k. parallellisme

PRP maakt gebruik van standaard Sun servers. Vergroting van de mate van parallellisme van RPCGEN servers verhoogt de transparantie van een RPC niet. De taak van PRP is een oplossing voor punt 5, 6 en 7 van § 4.2.3 te bieden. Vergroting van de mate van parallellisme maakt hier geen onderdeel van uit.

Onveranderd oordeel: $-$

6.2.3 Samenvatting

Hieronder wordt samengevat in welke mate PRP voldoet aan de twee verzamelingen criteria.

Allereerst wordt de overeenstemming tussen de criteria voor applicatieontwikkeling en de PRP-Sun omgeving samengevat. In de middelste kolom is kort het verschil tussen PRP en RPCGEN samengevat.

Gedistribueerde omgeving (PRP)

criterium	*kern van de verandering*	*score*
1. parallellisme	onveranderd	$\circ$
2. communicatie	RPC enig uitgewerkte paradigma	$+/\circ$
3. fouttolerantie	geen elementen toegevoegd	$-$
4. integratie	onveranderd	$\circ/-$
5. complexiteit	PRP accepteert C	$+$
6. typesecurity	wat minder pointers (bij parameters)	$\circ/-$
7. efficiëntie	iets trager	$\circ$
8. overdraagbaarheid	onveranderd	$+$

De volgende tabel vat de overeenkomst voor de RPC-specifieke aspecten samen.

RPC aspecten van PRP

criterium	*kern van de verandering*	*score*
a. globale variabelen	onveranderd	$-$
b. pointers	onveranderd	$+$
c. parameters	transparant	$+$
d. faal-semantiek	syntactisch transparant maar inflexibel	$\circ$
e. foutcode	meer transparantie	$\circ$
f. integratie RPC	transparant	$+$
g. interfacedefinitie	transparant	$+$
h. protocol	syntactisch transparant, geen keuze	$\circ$
i. netwerkadressering	portmapper, broadcast	$+$
j. beveiliging	ongebruikt	$-$
k. parallellisme	onveranderd	$-$

6.3 Implementatie prototype

Het PRP model is bedoeld om na te gaan of de transparantie van RPCGEN RPC's met een preprocessor op een aantal punten kan worden verhoogd. In § 5.2.1 is beschreven dat het ontwerp van het PRP programma volgens de methode van stapsgewijze verfijning is opgezet. Dit is gedaan om de complexiteit van het programmaontwerp te kunnen beheersen [Kernighan 1]. In dit traject zijn een aantal fasen te onderscheiden. De fasen lopen van beperkte ondersteuning van de mogelijkheden van gedistribueerde C applicaties tot uitgebreide ondersteuning. De onderstaande lijst somt de verschillende ondersteuningsniveaus op.

1. Simple types (integers, floating point en characters)
2. Statische eendimensionale arrays (arrays waarbij de dimensie in de declaratie vermeld wordt)
3. Strings
4. Broadcast-calls
5. Meerdimensionale statische arrays
6. Geneste RPC's (een serverprocedure die zelf een RPC uitvoert)
7. Structures, unions, typedefs en enumerated types; recursieve datastructuren als gelinkte lijsten, bomen en grafen
8. Dynamisch gealloceerde arrays (arrays waarbij het geheugen at runtime gealloceerd wordt door aanroep van een `malloc()`-routine)

Om het PRP-model op uitvoerbaarheid te onderzoeken zijn achtereenvolgens fase 1, 2, 3 en 4 geïmplementeerd in een prototype. Appendix A bevat een handleiding voor dit prototype.

De werkwijze van PRP ziet er in grote lijnen als volgt uit. Allereerst worden de applicatiefiles geanalyseerd door de parser. Bij het parsen zijn twee criteria van belang (zie § A.3.1):

- *Datatype* Of de parameter een simple, string, array, of structure type is.
- *Plaats* Of de procedure *gedeclareerd* of *gedefinieerd* wordt.

Wanneer na het parsen de interne tabellen gevuld zijn, worden de outputfiles gegenereerd. Bij de codegeneratie zijn ook weer twee criteria van belang (zie pagina 113):

- *Datatype* Of het simple, string, array, of structures zijn.
- *Plaats* Of de code voor (a) de `.x`-file, (b) de PRP-clientstub, of (c) de PRP-serverstub gegenereerd moet worden.

Hieronder worden de probemen bij het uitwerken van fase 1–4 in het PRP-prototype beschreven.

6.3.1 Ervaringen

Bij het verhogen van de transparantie binnen de gedistribueerde Sun omgeving is het fundamentele probleem dat C sterk gericht is op manipulaties met adressen binnen één logische adresruimte. Het parameteroverdrachtmechanisme van C abstraheert niet van low level pointers. De taal C is in hoge mate een sequentiële taal.

defaultregels voor parameteroverdracht

PRP probeert op basis van bepaalde constructies met low level C primitieven de abstractere (hogere) bedoelingen van de applicatieprogrammeur te achterhalen. Dit is het geval bij de defaultregels voor parameteroverdracht, en de keuze om een characterarray als string te interpreteren. De keywords `in`, `out` en `inout` zijn een soort noodoplossing zodat de programmeur deze vrij grove interpretatie kan corrigeren. Dit gaat ten koste van de transparantie.

broadcast-calls

De PRP-stubs maken gebruik van broadcast-calls om, indien de programmeur geen adres heeft gespecificeerd, het netwerkadres van een servermodule te bepalen. De serverroutines hebben hiertoe een kleine hulproutine die tot taak heeft een broadcastcall te beantwoorden (zie § C.2). De primitieven voor broadcast-calls behoren volgens [Sun 1] tot de low level primitieven. Het is een ingewikkelde en matig gedocumenteerde materie. De Sun RPC-primitieven schermen de implementatie met behulp van sockets niet af.

Een complexe implementatie levert hier primitieven met een hoog abstractieniveau op:[3] de PRP-stubs bieden (indien gewenst) locatietransparantie.

arrays

Het veelvuldig kopiëren van omvangrijke parameters is nadelig voor de efficiëntie van het systeem. Bij array-parameters moeten de PRP-stubs bij voorkeur niet de waarden zelf, maar pointers ernaar doorgeven. In de PRP-client-stub is dit bij het uitpakken van de RPCGEN-parameterstructure niet mogelijk.

In C levert een statisch-array declaratie een constante pointervariabele op. Dit houdt in dat het adres waar deze variabele naar wijst niet gewijzigd kan worden.[4] Een door de applicatieprogrammeur gedeclareerd statisch array kan niet gevuld worden door deze pointervariabele naar een geheugenblok met de nieuwe waarden te laten wijzen. De waarden moeten door de PRP-stub één voor één van de RPCGEN-stub naar de gebruikersvariabele gekopieerd worden (zie ook § A.3.1).

Een dynamisch-array declaratie levert een gewone (volatile) pointervariabele op—een varieerbare variabele. De overdracht van waarden van RPCGEN-stub naar gebruikersmodule kan hier efficiënter plaatsvinden door niet de waarden zelf, maar het beginadres ervan, door te geven.

resultaten

Voor het testen van implementatiefase 4 is de voorbeeldapplicatie uit appendix C gebruikt. Deze test de implementatie van de syntaxanalyse volgens

[3] Zie ook figuur 2.5.

[4] Het volgende is incorrect: `int a[10]; int *p; a = p;`
Wèl correct is: `int a[10]; int *p; p = a;`

§ A.2.2 en de codegeneratie volgens § A.3.1. De applicatie maakt gebruik van simple types, statische arrays, strings, en broadcast-RPC. De mogelijke combinaties van datatypes en plaats—al dan niet volgens de defaultregels—worden niet uitputtend getest. Voordat PRP serieus gebruikt kan worden—wanneer fase 8 geïmplementeerd is—moeten alle mogelijke combinaties getest worden. Welke combinaties dit zijn is uit tabellen als in § A.2.2 af te leiden.

Naast correctheidstests zijn snelheidstests belangrijk. RPC's zijn trager dan lokale procedureaanroepen. Alleen al om redenen van transparantie moet een RPC zo snel mogelijk zijn. Het PRP-programma tot en met fase 4 is niet op snelheid getest.

Een punt dat de snelheid van een RPC-systeem sterk beïnvloedt is het kopiëren van parameters [Tanenbaum 5]. In het PRP-model worden parameters bij het in- en uitpakken door de PRP-stubs van en naar de RPCGEN-parameterstructures gekopieerd. Van arrays en strings wordt in principe alleen het beginadres gekopieerd. Het kopiëren van omvangrijke geheugenblokken wordt zoveel mogelijk vermeden.[5] De applicatieprogrammeur kan door het gebruik van de keywords `in` en `out` overbodige communicatie voorkomen.

Uit een test, die in [Tanenbaum 5, p. 439] is beschreven, is af te leiden dat parametermarshalling in de stubs op de meeste systemen waarschijnlijk maximaal 10 procent van de overdrachttijd kost. Context switches, kopiëren tussen interne buffers van het operating system, en netwerkoverhead nemen de rest van de tijd in beslag.

Het kopiëren van enkelvoudige waarden of geheugenadressen zal de performance van applicaties waarschijnlijk niet excessief doen afnemen. PRP's transparantieverhoging leidt vermoedelijk niet tot een grote afname van de efficiëntie. Het veelvuldig gebruik van grote statische arrays als RPC-parameter zou hierop een uitzondering kunnen zijn.

De eventuele juistheid van deze vermoedens kan alleen door snelheidsmetingen worden vastgesteld.

6.3.2 Uitbreiding PRP-prototype

Voor de implementatie van fase 5–8 moet de invulling van het PRP-model uitgebreid worden. De structuur van het PRP-model zelf behoeft hiervoor niet gewijzigd te worden.

parser

Door de grammatica van de parser uit te breiden volgens [Kernighan 2] kan PRP declaraties van structures, unions, typedefs, enumerated types en meerdimensionale arrays herkennen. Daar de C declaratiesyntax hiervoor vast ligt is het parsen van deze datatypen geen probleem.

[5]Hiervoor is beschreven waarom het uitpakken van een statisch array in de PRP-clientstub hierop een uitzondering vormt.

Echter, de RPCGEN-syntax—de output van PRP—komt hier alleen wat betreft structures en typedefs mee overeen. Unions, enumerated types en meerdimensionale arrays moeten door PRP vertaald worden in constructies die voor RPCGEN geschikt zijn. De `.x`-file moet het RPCGEN-equivalent bevatten, de PRP-clientstub moet conversiecode bevatten die de C types naar RPCGEN-types vertaalt, en de PRP-serverstub moet conversiecode bevatten die RPCGEN-types naar C vertaalt.

Enumerated types komen met `int`'s overeen.[6] Deze conversie is triviaal.

Meerdimensionale arrays kunnen door het nesten van structures met één array-dimensie worden samengesteld. De PRP-clientstub converteert het meerdimensionale array naar een structure die bestaat uit sub-structures met elk één array-dimensie. De XDR-routines van de RPCGEN-clientstub *serializen* deze structure zodat het RPC-subsysteem deze kan verzenden.

Bij C unions doet zich het probleem voor dat het type at run-time kan wijzigen. Bij het serializen moet het datatype bekend zijn—een 4-byte-integer moet wellicht anders geconverteerd worden dan een 4-byte-characterstring. RPCGEN-unions hebben hierom een member dat het type aangeeft (zie pagina 112). C unions hebben dat niet, waardoor onduidelijk is welke XDR-conversieroutines gebruikt moeten worden.

Er zijn verschillende mogelijkheden om dit probleem op te lossen. Eén manier is om de RPCGEN-declaratiesyntax over te nemen. Dit betekent een nieuwe afwijking van de C syntax. Een andere mogelijkheid is om niet de declaratie te wijzigen, maar de applicatieprogrammeur te verplichten bij de aanroep van de PRP-clientstub het type te specificeren. Beide oplossingen zijn niet transparant voor de programmeur. Een oplossing die wel transparant is, is om de union als ongewijzigde bytestring over te zenden—de struisvogelvariant. Dit werkt alleen wanneer client en server dezelfde gegevensrepresentatie hebben. Deze oplossing doet afbreuk aan de XDR-standaard en de voordelen van 'open systemen'. Een vierde mogelijkheid is om unions als RPC-parameter te verbieden en de programmeur te dwingen voor RPC-parameteroverdracht alternatieven als structures te gebruiken.

dynamische geheugenallocatie

C is een *weakly typed language*. Dit betekent dat alleen de applicatieprogrammeur weet wat het type van sommige variabelen is. De compiler heeft hier geen informatie over. Tijdens de marshalling van parameters is het niet mogelijk om de omvang of het type van een parameter met zekerheid vast te stellen [Tanenbaum 5] (zie ook § A.4.1).

Het ontbreken van type-informatie in C levert—naast de problemen bij typesecurity en union-overdracht—ook bij de overdracht van dynamische arrays problemen op. Ten eerste is uit een pointerdeclaratie niet af te leiden of het een pointer naar een enkelvoudige datastructuur is, of dat het een pointer naar het beginwaarde van een array is. Om dit te kunnen vaststellen moet een nieuw syntaxelement worden ingevoerd—bijvoorbeeld scherpe

[6] In K&R C (de eerste druk van [Kernighan 2]) werden enumerated types ook op deze wijze gesimuleerd.

haken, net als in de XDR-taal [Sun 1].

Ten tweede kent de compiler de omvang van een at run-time gealloceerd dynamisch array[7] niet. Het is voor PRP niet mogelijk om uit de (compile-time) declaratie[8] de actuele omvang af te leiden. Alleen door gebruik te maken van implementatie-specifieke eigenschappen van het C-run-time systeem is de omvang van het geheugenblok soms te achterhalen. Sommige bibliotheken van C compilers bevatten routines waarmee dit mogelijk is. Deze oplossing is transparant voor de applicatieprogrammeur, maar niet altijd overdraagbaar naar andere ontwikkelomgevingen.

Wanneer deze mogelijkheden ontbreken kan men vervangende routines voor alle geheugenallocatieroutines schrijven—zoals `malloc()`, `calloc()`, `realloc()` en `free()`—die voor elk gealloceerde geheugenblok de omvang op een voor PRP bereikbare locatie opslaan. Deze vervangende routines moeten een andere naam hebben dan de oorspronkelijke routines. De programmeur moet deze vervangende routines gebruiken in de programmacode.

Wanneer een applicatie gebruik maakt van modules, waarvan de sourcecode niet beschikbaar is, kan het vóórkomen dat op deze wijze de omvang niet is vast te stellen. De programmeur moet de gegevens dan naar een zelf gealloceerd blok kopiëren, voordat ze kunnen worden overgezonden. Deze laatste oplossingen zijn niet transparant voor de applicatieprogrammeur.

Een andere niet-transparante oplossing is om de programmeur de omvang van het geheugenblok bij de aanroep van de remote procedure op te laten geven—zoals bij unions de programmeur het actuele type zou moeten specificeren.

geneste RPC's

De voorgaande punten hebben betrekking op het datatype van parameters. Dit punt heeft betrekking op de *plaats* van de te genereren code.

Een gedistribueerde RPC-applicatie is samengesteld uit clientmodules en servermodules. De voorbeeldapplicatie uit appendix B en C bestaat uit clientroutines die een serverroutine aanroepen, waarna de server weer terugkeert naar de client. De structuur van de applicatie kan ook complexer worden. Het kan zijn dat een remote procedure in een servermodule zelf weer een RPC pleegt naar een andere server. Dit is een *geneste* RPC [Spector, Weihl 2].

PRP genereert tot implementatiefase 5 een file met voor elke remote procedure een clientstub en een andere file met voor elke remote procedure een serverstub. De eerste file wordt met de clientmodules gelinkt, en de tweede file met de servermodules. PRP gaat uit van modules die puur client òf puur server zijn. Geneste RPC's—waarbij een serverprocedure zelf de client van een RPC wordt—zijn zo niet mogelijk.

Een oplossing is om per module de stubs te groeperen en voor elke applicatiemodule een aparte stub-module te genereren met de voor die module benodigde stubs. Op deze wijze bevat elk executable image precies die stubs die nodig zijn. Dit kunnen zowel client- als serverstubs zijn.

[7] Bijvoorbeeld: `p = (int *)malloc(aantal * sizeof(int));`

[8] Bijvoorbeeld: `int *p;`

Als bijvoorbeeld serverprocedure B door procedure A aangeroepen is, en zelf een (geneste) RPC naar procedure C doet, dan bevat module B de *server*stub voor B en de *client*stub voor de aanroep van C.

Hoofdstuk 7

Bevindingen

Centraal in de architectuur van gedistribueerde systemen staat het concept 'gescheiden geheugen'. Dit heeft grote consequenties voor het schrijven van toepassingsprogramma's voor deze omgevingen. Door de verschillende (logische) adresruimten moet men gebruik maken van interproces-communicatie [Tanenbaum 5]. Om gebruik te maken van de voordelen van gedistribueerde omgevingen (zie § 2.1.4) moet de structuur van veel toepassingen worden herzien.

Een gedistribueerd systeem is gedefinieerd als een *loosely coupled* systeem: de processoren hebben elk een eigen geheugen, en communiceren met elkaar via een netwerk. Communicatie via een netwerk is aanmerkelijk trager dan via gemeenschappelijk geheugen. Toepassingen die frequent communiceren kunnen beter op systemen met gemeenschappelijk geheugen worden geïmplemeneteerd. Toepassingen die infrequent communiceren—applicaties met *large grain* parallellisme—zijn geschikt voor gedistribueerde systemen. Om teveel communicatie te voorkomen moet de eenheid van parallellisme van de programmeertaal voldoende groot worden gekozen. Voor loosely coupled systemen zijn het *proces* of het *object* als eenheid van parallellisme geschikt (zie pagina 39).

Een gedistribueerd systeem is opgebouwd uit meerdere min of meer gelijkvormige componenten, bijvoorbeeld werkstations. Door de integratie van meerdere complexe bouwstenen is de complexiteit (in de zin van ingewikkeldheid) van het geheel groot. Om zo'n systeem bruikbaar te maken wordt gestreefd naar *transparantie* van de gedistribueerdheid. Dit houdt in dat het systeem voor de gebruiker of de applicatieprogrammeur een virtuele uniprocessor is.

abstractie

Het abstractieniveau van de concepten van een gedistribueerd systeem moet even hoog zijn als van een conventioneel systeem—bij het openen van een file moet de gebruiker niet gedwongen worden om zelf de plaats ervan te kennen. Door de ingewikkelder structuur van een gedistribueerd systeem moet met de primitieven een grotere *semantic gap* overbrugd worden. De semantiek van primitieven hangt samen met de efficiëntie van hun implementatie: het is ingewikkeld om met low level primitieven krachtige high level concepten efficiënt te implementeren.

Gedistribueerde applicaties maken gebruik van interproces-communicatie. Om transparantie aan applicatieprogrammeurs te kunnen bieden moet de semantiek van interproces-communicatieprimitieven zo krachtig mogelijk zijn. Een goede applicatieontwikkelomgeving biedt high level primitieven als RPC of logisch gemeenschappelijk geheugen aan, in plaats van low level primitieven als point-to-point message passing. Bij veel RPC-omgevingen wordt de ingewikkelde low level communicatie met behulp van een stubgenerator voor de applicatieprogrammeur transparant gemaakt. De stubs worden op basis van een high level specificatie gegenereerd. Deze specificatie wordt 'interfacedefinitie' genoemd [Birrell].

fouten

Een *volledig transparante* applicatieontwikkelomgeving is op het gebied van beschikbaarheid van systemen en betrouwbaarheid van gegevens voor de meeste toepassingen om economische redenen niet efficiënt [Bal]. Omgevingen die de programmeur fouttolerante primitieven bieden zijn minder transparant, maar hebben een breder toepassingsgebied. Met primitieven voor atomaire transacties kan de programmeur zelf de voor een applicatie geschikte mate van fouttolerantie implementeren.

Fout-transparantie heeft bij RPC-systemen te maken met de *faal-semantiek* van een RPC. Zero-or-one semantiek garandeert dat de side-effects van een serverprocedure òf in hun geheel optreden, òf (bij uitval van een component) geen enkel effect hebben op de toestand van het systeem. Bij zero-or-one semantiek wordt voor de uitvoering van een RPC de systeemtoestand vastgelegd op secundair geheugen. Voor procedures zonder side-effects—idempotente procedures—is het vastleggen van de toestand niet nodig: bij at-least-once semantiek vindt na een time-out bij de client hertransmissie plaats (zie pagina 62).

Een gedistribueerde applicatie is opgebouwd uit processen met verschillende adresruimtes. Veel concepten uit sequentiële programmeertalen gaan uit van één aaneengesloten adresruimte. De semantiek van globale variabelen, pointervariabelen en parameteroverdracht *by reference* is niet efficiënt in een gescheiden-geheugen omgeving te implementeren (zie pagina 46). De programmeur moet zich realiseren dat een RPC zonder (logisch) gemeenschappelijk geheugen niet meer dan een surrogaat procedureaanroep kan zijn. Een PRP RPC lijkt op een gewone aanroep, maar is het niet. Het doel van Birrell & Nelson van pagina 8 is dat men geen expert hoeft te zijn om gedistribueerde applicaties te schrijven. Zo men dan geen expert meer hoeft te zijn, moet men zich wel degelijk bewust zijn van de gevolgen die het ontbreken van een gemeenschappelijke adresruimte tussen processen heeft.

Omdat een gedistribueerd systeem gecompliceerder is dan een gecentraliseerd systeem is het opsporen van fouten in een gedistribueerde applicatie moeilijker dan in een vergelijkbare sequentiële applicatie. Om de testfase efficiënt te laten verlopen is het van belang dat compiler en run-time systeem van een taal alle afwijkingen van de regels van de taal—verkeerde array-indices, ongeïnitialiseerde pointers—signaleren [Hoare]. Omdat het uitvoeren van run-time checks ten koste gaat van de efficiëntie van een programma wordt dit vaak achterwege gelaten.

Sun

Uit de literatuur zijn criteria afgeleid die bruikbaar zijn voor de beoordeling van applicatieontwikkelomgevingen voor gedistribueerde systemen. Naast een lijst met algemene criteria voor gedistribueerde omgevingen is voor de beoordeling van de Sun omgeving een lijst met specifieke RPC-criteria beschreven—deel een van de probleemstelling. De uitkomsten van de beoordeling van de Sun omgeving—deel twee van de probleemstelling—komen overeen met de beschrijvingen van [Chin, Coulouris].

De Sun omgeving is een loosely coupled omgeving volgens het werkstation/server model. De applicatieontwikkelomgeving is ontworpen rond het proces als expliciete eenheid van parallellisme. De Sun systemen zijn voor de ontwikkeling van client/server-applicaties ontworpen (§ 3.4.4, [Sun 1]). Deze categorie applicaties kent een structuur die goed in modules met een specifieke functionaliteit te verdelen is—reden 3 uit § 2.1.4. Hiervoor is een omgeving waarin men expliciet parallelle modules kan specificeren geschikt. De Sun omgeving heeft geen voorzieningen voor fouttolerantie. Voor de implementatie van applicaties die een hoge mate van bedrijfszekerheid vereisen zal extra programmatuur moeten worden gekocht of geschreven.

De Sun ontwikkelomgeving volgt de structuur van § 3.2.1, waaraan in navolging van [Birrell] een codegenerator (RPCGEN) is toegevoegd. De syntax van de programmamodules is standaard C; ze worden direct door de C compiler verwerkt. Het gebruik van de RPCGEN-stubprocedures is niet transparant voor de programmeur. Parameters moeten op een gekunstelde en foutgevoelige manier met structures en pointers worden overgezonden, en men moet extra initialisatie- en foutafhandelingscode schrijven.

PRP

PRP is ontworpen om drie van de acht zwakke punten van de Sun omgeving te verbeteren—deel drie van de probleemstelling, zie pagina 93. PRP is een preprocessor die een extra stub-laag om de RPCGEN-stubs heen produceert. Het gebruik van de RPCGEN-stubs wordt zoveel mogelijk transparant gemaakt. De programmeur behoeft geen interfacedefinitie meer te schrijven, het specificeren van de servernaam is vereenvoudigd, de extra foutcontroles zijn weggevallen, en parameters kunnen op de in C gebruikelijke wijze worden gespecificeerd. De syntax en semantiek van de parameteroverdracht van PRP RPC's benadert die van lokale C procedures. Ter illustratie van de transparantieverhoging is in appendix B en C een voorbeeldapplicatie zonder en met PRP opgenomen.

Het PRP-model is volgens § 3.2.2 opgezet. De programmamodules worden door PRP, een preprocessor, bewerkt voor ze door de C compiler worden gecompileerd. PRP genereert code voor de interfacedefinitie voor RPCGEN, de aanroepen van system-calls en de aanroepen van RPCGEN-stubs. De syntax van programmamodules voor PRP is niet standaard C, maar 'extended C'. PRP programmamodules bevatten met het oog op transparantie zo min mogelijk vreemde *taalconstructies* zoals RPCGEN. Wel zijn er drie nieuwe *syntaxelementen* geïntroduceerd (§ 6.1). De taal die PRP accepteert is niet strikt ANSI C. In § A.4 is beschreven op welke punten problemen optreden bij de integratie van C in een gedistribueerde omgeving.

verder onderzoek

De RPC-specifieke beoordelingscriteria zijn niet bruikbaar voor omgevingen die op een ander interproces-communicatieparadigma—zoals logisch gemeenschappelijke variabelen—gebaseerd zijn. Voor een gedetailleerde beoordeling van dit soort omgevingen moeten andere criteria opgesteld worden. De lijst algemene criteria kan hiertoe een aanzet vormen.

De introductie van extra stubs vergroot de overhead van een RPC. Om parameters niet onnodig over te zenden kan de programmeur bij elke parameter de overdracht*richting* (input, output of beide) opgeven. Omdat de extra overhead in verhouding tot de netwerkoverhead waarschijnlijk relatief klein zal zijn bestaat het vermoeden dat dit bij de meeste parametertypen geen grote vertraging zal veroorzaken. In hoeverre de transparantieverhoging van PRP uiteindelijk gepaard gaat met afname van de efficiëntie kan alleen door snelheidstests worden uitgewezen.

Het PRP-model is tot en met fase 4 van pagina 122 geïmplementeerd. Voor de ondersteuning van alle mogelijkheden van de taal C moet dit ook voor de overige fasen gebeuren. Wanneer dit is gebeurd moeten de snelheids- en correctheidstesten plaats vinden. Voor het ontwerpen van een testprogramma kan gebruik gemaakt worden van de ontwerpschema's voor de codegeneratie. Deze geven een uitputtende beschrijving van de door PRP te produceren code.

PRP biedt een oplossing voor drie zwakke punten van de Sun omgeving. Voor de andere punten, op het gebied van fouttolerantie (zero-or-one semantiek), interproces globale variabelen, en parallellisme van de server zijn verdere verbeteringen van de Sun omgeving mogelijk (zie pagina 93). De beschrijvingen van de criteria waarop deze punten beoordeeld zijn kunnen als concreet uitgangspunt dienen voor verder onderzoek (zie hoofdstuk 3).

Op het gebied van fouttolerantie zijn al veel onderzoeksresultaten geboekt. Naar parallelle serverstructuren, en ook naar impliciet parallellisme (parallelliserende compilers) wordt nog veel onderzoek verricht [Bal]. Deze problemen vereisen een ander soort oplossing dan een preprocessor voor RPCGEN kan bieden.

C is een procedurele taal met een laag abstractieniveau [Kernighan 2, p. 1]. Voor applicaties die impliciet parallellisme of atomaire transacties vereisen is de standaard Sun applicatieontwikkelomgeving niet de meest geschikte keuze. Een andere taal dan C, of een ander communicatieparadigma dan RPC, kan voor bepaalde toepassingen beter geschikt zijn—bijvoordeeld een fouttolerante omgeving als Argus of Camelot, een functionele taal als ParAlfl, of een gemeenschappelijke variabele-taal als Linda of Orca.

Daarnaast is een conceptueel aaneengesloten adresruimte voor de taal C wel haast fundamenteel. Aangezien een gedistribueerde omgeving gescheiden geheugen kent, is het de vraag of C wel zo geschikt is voor het programmeren van gedistribueerde applicaties (zie § A.4).

Hoofdstuk 8

Samenvatting

gedistribueerdheid

In deze scriptie wordt onderzocht hoe het schrijven van toepassingsprogramma's voor de gedistribueerde omgeving van Sun zo goed mogelijk ondersteund kan worden. Om dit te bereiken wordt geprobeerd de transparantie van de Sun remote procedure call te vergroten. Hiervoor is een literatuurstudie verricht naar de eisen die men aan een applicatieontwikkelomgeving voor gedistribueerde programma's kan stellen. Op basis van deze eisen is een programma geschreven met als doel de programmatuurontwikkeling te vereenvoudigen.

Het onderwerp van deze scriptie is hoe men de ontwikkeling van applicaties voor gedistribueerde computersystemen in het algemeen en de Sun omgeving in het bijzonder adequaat kan ondersteunen.

Een gedistribueerd systeem bestaat uit computers die in een netwerk met elkaar zijn verbonden. De processoren hebben geen gemeenschappelijk geheugen, ze communiceren met elkaar door middel van boodschappen via het netwerk. Het nadeel van communicatie via een netwerk is, dat het trager is dan via gemeenschappelijk geheugen. Het voordeel is dat computers geografisch verspreid kunnen worden opgesteld, en zo potentieel een grotere betrouwbaarheid hebben dan gecentraliseerde systemen. Een voorbeeld van een gedistribueerd model is het werkstation/server-model.

Er zijn vier redenen om applicaties op een gedistribueerd computersysteem te implementeren:

1. Een kortere doorlooptijd voor een enkele berekening.
2. Toegenomen betrouwbaarheid en beschikbaarheid.
3. De mogelijkheid bepaalde delen van het systeem te gebruiken om specifieke functionaliteit efficiënt aan te kunnen bieden.
4. De mogelijkheid inherente gedistribueerdheid van een applicatie te benutten.

Applicaties die om reden 3 of 4 op een gedistribueerd systeem worden geïmplementeerd worden vaak volgens het client/server-model opgezet. Een server is een computer die een bepaalde dienst aan de clients aanbiedt.

Een gedistribueerd systeem heeft drie fundamentele eigenschappen: *fouttolerantie*, *parallellisme*, en *transparantie*.

Een gedistribueerd systeem is opgebouwd uit meerdere computers. Bij de uitval van een computer kan het systeem de taken verdelen over de overgebleven computers en zo een applicatie blijven uitvoeren—fouttolerantie. Door een probleem in meerdere delen te splitsen kunnen verschillende computers er gelijktijdig aan rekenen—parallellisme. Het verschil tussen een *netwerk* operating system en een *gedistribueerd* operating system is transparantie. In een netwerk operating system is de gebruiker zich bewust van de verschillende componenten, en moet de namen ervan kennen om bijvoorbeeld een programma elders op een computer op te starten. In een gedistribueerd systeem is het verschil tussen de computers onzichtbaar voor de gebruiker—het is een *virtuele uniprocessor*.

Vanwege de grote complexiteit van een gedistribueerd systeem is transparantie moeilijk te verwezenlijken. In veel gevallen gaat transparantie ten koste van de efficiëntie van een systeem. Een gedistribueerd systeem bevat veel compromissen. Deze keuzes zijn van invloed op de categorie toepassingen waar het systeem geschikt voor is.

De Sun omgeving kent geen parallelliserende compilers. De programmeur moet zelf een applicatie in parallelle modules verdelen. Verder kent de Sun omgeving geen ondersteuning voor fouttolerantie. Door deze twee eigenschappen is de Sun omgeving met name geschikt voor applicaties die om reden 3 en 4 op een gedistribueerd systeem worden geïmplementeerd, en niet om reden 1 en 2. Dit zijn niet-fouttolerante client/server applicaties.

probleemstelling

De probleemstelling luidt als volgt:

- Aan welke criteria moet een goede ontwikkelomgeving voor gedistribueerde applicaties voldoen?
- In hoeverre voldoet de Sun ontwikkelomgeving aan deze criteria?
- Op welke wijze kan het programmeren voor de Sun omgeving vereenvoudigd worden?

De Sun omgeving is gebaseerd op de remote procedure call. Een remote procedure call (RPC) is een veelgebruikte vorm van interproces-communicatie. In tegenstelling tot een gewone *intra*proces-procedureaanroep is een RPC een *inter*proces-procedureaanroep.

RPC-primitieven worden in de Sun omgeving in de vorm van system-calls door het operating system aangeboden. Deze system-calls hebben een zwakke semantiek. Het schrijven van een applicatieprogramma met deze system-calls is erg ingewikkeld. Naar analogie van het systeem van Birrell & Nelson kent de Sun omgeving daarom een stubgenerator. De naam van deze codegenerator is RPCGEN. Op basis van een file met de declaraties van de remote procedures—de interfacedefinitie—genereert RPCGEN een aantal

C modules die de stubs bevatten.[1] Een stub is een lokale procedure die de details van het gebruik van RPC-system-calls afschermt van de applicatie-programmeur.

Deze stubs zijn ingewikkeld te gebruiken. Een RPCGEN RPC kan maximaal één input- en één outputparameter hebben. Om meerdere parameters over te zenden moet de programmeur ze in *structures* groeperen. Daarnaast moet de programmeur veel foutcontroles schrijven. Er zijn veel mogelijkheden om bij het gebruik van RPCGEN-stubs fouten te maken.

Om de applicatieontwikkelomgeving van Sun te beoordelen wordt op basis van de literatuur in hoofdstuk 3 een aantal criteria geformuleerd—deel een van de probleemstelling. In de beoordeling wordt een aantal sterke en zwakke punten van de Sun omgeving geïdentificeerd—deel twee van de probleemstelling.

PRP

Voor een aantal van de zwakke punten uit de beoordeling wordt een verbetering voorgesteld: het PRP-model—deel drie van de probleemstelling. PRP staat voor Pre RPCGEN Processor. Deze zwakke punten hebben betrekking op het lage transparantieniveau dat de RPCGEN-stubs bieden—het ingewikkelde gebruik ervan.

Het doel van PRP is om op basis van de standaard programmamodules van een applicatie de input van een RPCGEN-programma te genereren: een interfacedefinitie, één input- en één outputparameter per RPC, en intialisatiecode en foutcontroles. Er is een programma geschreven om te onderzoeken of het PRP-model werkt. De complexe RPCGEN-stubs worden van de programmeur afgeschermd door extra PRP-stubs. De aanroepwijze van PRP-stubs komt meer met die van lokale procedures overeen dan de aanroepwijze van RPCGEN-stubs.

Het bleek niet mogelijk om met een eenvoudige preprocessor de gewenste output uitsluitend op basis van ANSI C te genereren. Er zijn drie fundamentele elementen aan de inputsyntax toegevoegd: een aanduiding welke procedures met een RPC aangeroepen worden, een plaatsaanduiding (netwerkadres) voor deze *remote* procedures, en een richtingsaanduiding om onduidelijkheden bij de parameteroverdracht weg te nemen en de efficiëntie ervan te verhogen. In § 5.3.2 en § 6.1 wordt dit beschreven. In § A.4 is beschreven op welke verdere punten er problemen optreden bij de integratie van C in een gedistribueerde omgeving.

Uit de beoordeling van het PRP-model op basis van de criteria van hoofdstuk 3 is naar voren gekomen dat het mogelijk is de transparantie van een RPC te verhogen. De programmeur kan bij een PRP-stub meerdere parameters gebruiken, men hoeft geen extra initialisatie- of foutcontrolecode te schrijven, en bij eenvoudige applicaties is het mogelijk de plaatsaanduiding van een RPC weg te laten. De aanroepsyntax van een lokale procedureaanroep wordt dicht benaderd.

[1] De Sun applicatieontwikkelomgeving is gebaseerd op UNIX en de taal C.

Het PRP-model gaat net als RPCGEN uit van processen met gescheiden geheugen. Geheugenreferenties hebben in een ander proces geen betekenis. Call by reference wordt met kopieëren van parameters gesimuleerd. In bepaalde situaties verschilt de semantiek van een RPC en een lokale procedureaanroep hierdoor (zie pagina 61). Ook de faal-semantiek van procedures met side-effects is verschillend (zie pagina 62).

De transparantie is met eenvoudige middelen—een preprocessor—zoveel mogelijk verhoogd. Een aantal meer fundamentele problemen vormt een beletsel voor volledige transparantie. De applicatieprogrammeur moet zich steeds realiseren dat een RPC slechts een surrogaat procedureaanroep is, en de processen geen gemeenschappelijke adresruimte hebben.

Een conceptueel aaneengesloten adresruimte is voor de taal C wel haast fundamenteel. Aangezien een gedistribueerde omgeving gescheiden geheugen kent, is het de vraag of C wel zo geschikt is voor het programmeren van gedistribueerde applicaties.

aanbevelingen

De PRP-stubs introduceren enige extra overhead bij de overdracht van parameters. Met snelheidstests zal moeten worden vastgesteld in hoeverre het vermoeden juist is dat deze overhead acceptabel is.

Om het PRP-model op uitvoerbaarheid te onderzoeken zijn een aantal fasen uit het ontwerp geïmplementeerd. Wanneer alle fasen geïmplementeerd zijn, en de PRP-implementatie alle mogelijkheden van gedistribueerde C applicaties ondersteunt, moeten correctsheids- en snelheidstests worden uitgevoerd.

Het PRP-model probeert drie zwakke punten van de Sun omgeving te verbeteren. Op het gebied van fouttolerantie (zero-or-one semantiek), interproces globale variabelen, en serverparallellisme zijn verdere verbeteringen van de Sun omgeving mogelijk. Deze problemen zijn te omvangrijk om binnen het kader van een eenvoudige preprocessor voor RPCGEN op te kunnen lossen.

De beschrijvingen van de criteria in hoofdstuk 3 kunnen als concreet uitgangspunt dienen voor verder onderzoek naar de overige vijf punten van pagina 93.

Appendix A

Handleiding prototype

In deze appendix worden punten besproken die voor het gebruik van het PRP-prototype van belang zijn. Het PRP-*model* verschilt op een aantal punten van het PRP-*prototype* zoals het hier besproken wordt, omdat dit prototype slechts een deel van het model implementeert. (In § 6.3 is behandeld welk deel van het model in het prototype geïmplementeerd wordt.)

In deze appendix wordt met 'PRP' het programma-dat-het-prototype-implementeert bedoeld.

A.1 Een voorbeeld

A.1.1 De compileergang

Het maken van een gedistribueerde applicatie met PRP bestaat uit de volgende stappen:

1. Bepaal uit welke modules de applicatie moet bestaan en schrijf deze. Dit zijn de `.c`-files (met de bijbehorende `.h`-files).

2. Genereer de PRP-stubfiles, de XDR-file (`.x`-file) en de `makefile` met `prp`—het PRP-*programma*.

3. Laat RPCGEN op basis van de `.x`-file de XDR-conversieroutines en de RPCGEN-stubs genereren.

4. Maak met de C compiler de verschillende executable images—de modules uit de eerste stap, die de processen van de applicatie gaan vormen.

5. Start de serverprocessen op de juiste machines op, en laat de gebruiker weten dat de clientprocessen klaar voor gebruik zijn.

Stap 3 en 4 bestaan uit vrij veel files (zie figuur C.1) die wellicht een wat onoverzichtelijke indruk geven. Door met `make` de `makefile` uit stap 2 te gebruiken, worden in stap 3 en 4 automatisch de juiste files verwerkt.

A.1.2 De applicatie (1)

In appendix C is de sourcecode van een voorbeeldapplicatie opgenomen. In deze paragraaf wordt aan de hand van deze applicatie besproken hoe een PRP-applicatie gecompileerd en uitgevoerd kan worden.

De applicatie bestaat uit drie modules, die drie processen vormen. Er is één client-module en er zijn twee server-modules. De ene server rekent de produktvector van twee input-vectoren uit, en de andere drukt een vector op het scherm van de server af. Het client-proces leest waarden voor twee vectoren uit een file, en roept vervolgens de rekenserver en de afdrukserver door middel van twee RPC's aan. Bij de eerste RPC—naar de rekenserver—wordt het serveradres niet gespecificeerd (broadcast). Bij de tweede RPC—naar de afdrukserver—wordt de server die de gebruiker heeft opgegeven geselecteerd. Wanneer de gebruiker de client opstart, moet deze twee argumenten meegeven: de naam van de afdrukserver en de naam van de file met de waarden van de twee vectoren. De client zou bijvoorbeeld als `$ calc sus.eur.nl vectoren.dat` kunnen worden opgestart.

Voor een succesvolle uitvoering van de applicatie moeten de serverprocessen in het systeem actief zijn. De programmeur moet de reken- en afdrukservers respectievelijk op een station opgestart hebben dat in de `/etc/hosts`-file van de client voorkomt,[1] en op alle stations die een gebruiker eventueel voor de afdrukserver zou mogen specificeren.

De applicatie bestaat uit drie modules; de source-code bevindt zich in drie `.c`-files: `calc.c` bevat de client, `calc_ber.c` de rekenserver en `calc_pr.c` de printserver. Er is een headerfile (`calc.h`) met globale constanten en de remote-prototypes van de twee serverprocedures.

A.1.3 PRP (2)

Door nu het programma `prp` met de filenamen van de applicatie als argumenten uit te voeren—`prp calc.c calc_pr.c calc_ber.c`, of zelfs `prp *.c`—worden de volgende files gegenereerd:

- Een XDR-file, `PRP_intf.x`, met de declaraties van de parameters van de serverprocedures. RPCGEN genereert aan de hand van deze file de conversieroutines voor de RPC-parameters.

  Een tweede functie van de XDR-file—de interfacedefinitie—is het declareren van serverprocedures en programma- en versienummers. Voor de programma- en versienummers kent het PRP-model een aparte file, `PRP_x.h`. In deze file kan de programmeur twee waarden opgeven voor de identificatie van de applicatie binnen de Sun-omgeving. Wanneer de file niet bestaat, wordt deze, voorzien van standaardwaarden, gecreëerd. Het programmanummer moet bij Sun Microsystems worden aangevraagd [Sun 1]; het versienummer wordt bij elke wijziging van de applicatie door PRP verhoogd. In § C.2 is een voorbeeld van `PRP_x.h` te vinden.

[1] Bij een broadcast-call worden alle stations uit de `/etc/hosts`-file benaderd.

- Twee files, `PRP_c_stub.c` en `PRP_s_stub.c`, die respectievelijk de PRP-client- en de PRP-serverstubs van de serverprocedures bevatten.

  De werking van de PRP-stubs wordt in § A.3 beschreven. De naam van de clientstubs bestaat uit de naam van de serverprocedure voorafgegaan door `_PRP_STUB_`. De PRP-clientstub bevat de aanroep van de RPCGEN-clientstub. Het RPC-subsysteem aan de serverkant roept vervolgens de RPCGEN-serverstub in het serverproces aan, die de PRP-serverstub aanroept. De PRP-serverstub roept vervolgens de originele serverfunctie aan die de programmeur geschreven heeft. De naam van deze laatste is voorzien van `_PRP_PROC_`, ter voorkoming van naamsconflicten met de RPCGEN-stubs.

- De oorspronkelijke applicatiemodules uit stap 1 worden enigszins gewijzigd. Om te beginnen moeten de functienamen in aanroep en definitie van de serverprocedures van `_PRP_STUB_` en `_PRP_PROC_` voorzien worden. Daarnaast worden er ANSI-prototypes van de serverprocedures gegenereerd teneinde de consistentiecontrole op parametertypen van ANSI C te waarborgen. Door deze prototypes signaleert de C compiler inconsistenties tussen het remote-prototype en aanroep en definitie in de applicatiecode zelf.

  Applicatiemodules die een serverprocedure aanroepen krijgen `_cl` achter hun naam, en modules waarin een een serverprocedure wordt gedefinieerd `_sv`.

- Om te voorkomen dat al deze files met de hand moeten worden gecompileerd wordt een `makefile` gegenereerd, waarmee stap 3 en 4 met een commando kunnen worden uitgevoerd.

  De `makefile` die dit prototype genereert is relatief eenvoudig, en niet geschikt voor complexe client/server configuraties als geneste RPC's en meerdere verschillende clients per server. Voor dit soort applicaties moet men zelf een geavanceerdere `makefile` schrijven.

Voor de voorbeeldapplicatie worden door PRP de volgende files gegenereerd:

- PRP_intf.x: de interfacedefinitie voor RPCGEN(eventueel ook PRP_x.h);
- PRP_c_stub.c: de file met de PRP-clientstubs voor de serverprocedures;
- PRP_s_stub.c: de file met de PRP-serverstubs;
- calc_cl.c: de bijgewerkte applicatiemodule (client);
- calc_ber_sv.c: de bijgewerkte applicatiemodule (rekenserver);
- calc_pr_sv.c: de bijgewerkte applicatiemodule (afdrukserver);
- makefile.prp: de `makefile` voor stap 3 en 4.

A.1.4 RPCGEN en de C compiler (3 & 4)

De interfacedefinitie wordt door RPCGEN verwerkt. RPCGEN genereert vier files op basis van `PRP_intf.x`: een file met de XDR-conversieroutines, twee files met de RPCGEN-stubs, en een headerfile met parameterdeclaraties.

- PRP_intf_xdr.c bevat de code voor het (de)serializen van de RPC-parameters. Deze routines worden door beide RPCGEN-stubs aangeroepen voor het marshallen. De XDR-routines moeten met beide stubfiles worden meegelinkt.
- PRP_intf_clnt.c bevat de RPCGEN-clientstubs van de remote-procedures.
- PRP_intf_svc.c bevat de RPCGEN-serverstubs.
- PRP_intf.h bevat C declaraties van de in de XDR-file gedeclareerde RPC-parameters, alsmede prototypes van de RPCGEN-stubs. Deze file wordt ge-`#include` in de PRP-stubfiles.

Nadat de RPCGEN-stubs geproduceerd zijn, kan de C compiler de applicatiemodules, de verschillende stubfiles en de XDR-routines tot executable images compileren.

Alle PRP-serverstubs die door dit prototype worden gegenereerd zitten in één file—`PRP_s_stub.c`. De voorbeeldapplicatie kent twee serverfiles—`calc_ber.c` en `calc_pr.c`. De serverstubfile moet met beide server-images meegelinkt worden opdat de serverstubs de serverprocedures kunnen aanroepen. Om te voorkomen dat de stubs aan procedures refereren die tijdens het linken niet aanwezig zijn, moeten ook beide applicatie-serverfiles met de ene stubfile worden meegelinkt. Op deze wijze ontstaat één executable serverimage dat zowel als rekenserver als als afdrukserver optreedt.

Bij het compileren van de voorbeeldapplicatie moeten zodoende twee executable images geproduceerd worden: een client-image, en een (gecombineerd) server-image. De client wordt gevormd door de volgende files uit stap 2:

- calc_cl.c: de client-applicatiefile
- PRP_c_stub.c: de PRP-clientstubs
- PRP_intf_clnt.c: de RPCGEN-clientstubs
- PRP_intf_xdr.c: de XDR-conversieroutines

De server wordt gevormd door deze files:

- calc_ber_sv.c: de rekenserver-applicatiefile, èn
- calc_pr_sv.c: de afdruk-applicatiefile
- PRP_s_stub.c: de PRP-serverstubs
- PRP_intf_svc.c: de RPCGEN-serverstubs
- PRP_intf_xdr.c: de XDR-conversieroutines

A.1.5 Opstarten van de modules (5)

De laatste stap is het opstarten van de applicatie. Hiertoe moeten de servers actief zijn op de stations waar de client een RPC naar zou kunnen (mogen) gaan doen.

Wanneer de stations hetzelfde filesysteem delen kunnen de servers eenvoudig vanuit de (remote) shell's worden opgestart als achtergrond-processen. Is dit niet het geval, dan moeten de executable images naar het andere filesysteem worden gekopiëerd, en vandaar opgestart worden.

Er is een veelheid aan manieren om dit te bereiken, afhankelijk van de inrichting van het systeem waarop men zich bevindt. De volgende twee voorbeelden zullen in het algemeen wel werken.

Stel dat `slc09`, `slc10` en `ipc04` dezelfde fileserver delen, en de server op `slc10` en `ipc04` moet draaien, en de client vanuit `slc09`.

```
slc09 % ls
calc    calc_sv    vectoren.dat
slc09 % rsh ipc04                  maak een remote shell op ipc04
ipc04 % calc_sv &                  start de server in de achtergrond op
ipc04 % rsh slc10
slc10 % calc_sv &                  start de server ook hier op
slc10 % rsh slc09
slc09 % calc slc10 vectoren.dat    de client wordt opgestart
```

Een andere mogelijkheid is om `ftp` te gebruiken. Stel dat niet `slc10`, maar `sus.eur.nl` de output moet afdrukken, en dat deze laatste een eigen filesysteem heeft, en dat de remote shell en `rlogin` niet werken.

```
slc09 % ls
calc     calc_sv     vectoren.dat
slc09 % rsh ipc04
ipc04 % calc_sv &                  start de server in de achtergrond op
ipc04 % ftp sus.eur.nl
na inloggen op sus.eur.nl:
ftp> binary                        executables zijn geen ASCII-files
ftp> put calc_sv                   kopieer de server
ftp> bye
ipc04 % telnet sus.eur.nl          maak contact met servermachine
na inloggen op sus.eur.nl:
eursus % calc_sv &                      start server op
eursus % telnet slc09.cs.few.eur.nl     ga terug voor de client
na inloggen op slc09:
slc09 % calc sus.eur.nl vectoren.dat    de client wordt opgestart
```

In [Sun 1] is meer informatie over het opstarten van de verschillende modules te vinden.

A.2 C extensies

A.2.1 Verschil PRP-C en ANSI C

PRP-C wijkt op een aantal punten af van ANSI C. Hieronder worden deze punten opgesomd. Allereerst worden de punten genoemd die voor *elke* PRP-applicatie afwijken. Daarna worden punten beschreven die de programmeur volgens eigen inzicht kan toepassen.

- *Remote* Voor elke C-functie die door een functie uit een ander proces aangeroepen moet kunnen worden moet men een ANSI prototype[2] voorzien van het keyword `remote` in de source code opnemen. De plaats in de applicatiecode van deze remote-prototypes is niet van belang omdat PRP de applicatiecode in zijn geheel analyseert, voordat de functienamen in de gebruikersfiles aangepast worden (zie figuur 5.4). De in C gebruikelijke manier is om prototypes voorin de `.c`-file of in een headerfile op te nemen.

- *Geen globale variabelen* Voor het delen van gegevens tussen twee functies uit een verschillend proces kan men niet gebruik maken van globale variabelen. Pogingen hiertoe worden helaas niet gesignaleerd (zie ook § A.4).

  De semantiek van *intra*proces globale variabelen is in PRP-C gelijk aan ANSI C.

- *Pointers* Interproces-geheugenreferenties zijn in verband met de efficiëntie niet geïmplementeerd in de Sun-omgeving. Het gebruik van pointers voor parameters in een RPC wordt door PRP alleen ondersteund wanneer dit in C gebruikelijk is. Dit is het geval bij referenceparameters en bij strings en arrays. (Dynamisch gealloceerde datastructuren en recursieve datastructuren zijn in dit prototype niet geïmplementeerd.)

- *Variabel aantal parameters* Zoals op pagina 105 is beschreven, moeten PRP-serverprocedures een vast aantal parameters hebben. Iets als `printf(fmt, ...);` is niet mogelijk.

- *Datatypen* Dit prototype ondersteund als parameter van een PRP RPC de volgende datatypen:

  - de gebruikelijke combinaties van `int`, `signed`, `unsigned`, `long` en `short`. Bijvoorbeeld: `int`, `unsigned`, `unsigned long`, `long`, `signed short int`;
  - de gebruikelijke combinaties van `char`, `unsigned` en `signed`. Bijvoorbeeld: `char`, `unsigned char`;
  - `float`, `double` en `long double`;

[2] Vergelijkbaar met de `forward`-declaratie in Pascal.

  - statisch gedeclareerde arrays van simple (bovenstaande) types zoals: `int a[10]`, `long a[600]` en `char[6]`;
  - 'strings': in dit PRP-prototype wordt de RPC-parameter-declaratie `char *s` in het XDR-type `string` vertaald. Dit houdt in dat alle elementen worden overgezonden totdat een element `NULL` is. Een pointer naar een enkelvoudige referenceparameter van het type `char` moet in dit prototype als `int *c` worden gedeclareerd. Dit kan in een heterogene omgeving tot conversie problemen leiden (zie ook § A.4.1).

  Zoals in § 6.3 is beschreven, worden structures, unions en dynamische geheugenallocatie door dit prototype niet ondersteund voor RPC-parameters. Daarbij komt dat meervoudige indirectie (`int **n`) niet zinvol is in een gedistribueerde omgeving, en door dit prototype niet wordt toegestaan. Enkelvoudige indirectie komt voor bij reference parameters (`out` en `inout`). Indirectie bij een valueparameter (`in`: `in int *n`) wordt niet ondersteund.

Voor veel toepassingen zal men van de volgende extensies gebruik moeten maken:

- *@-notatie* Wanneer men een specifieke server wil selecteren, kan men het netwerkadres van de server tussen twee @-symbolen specificeren. De identifier wordt bij de eerste RPC door de PRP-stubs geëvalueerd, waarna alle volgende RPC's van deze procedure naar deze server zullen gaan. (In dit prototype is niet in de mogelijkheid van server te wisselen voorzien. Dit zou men kunnen oplossen door de RPC aan de client-kant in een `case`-statement op te nemen.)

  Laat men de servernaam weg, dan wordt een broadcast naar alle stations uit `/etc/hosts`—alle 'lokale' stations—gedaan, en wordt de eerst reagerende server voor alle volgende RPC's naar deze procedure gebruikt.

- *'Richting'* Ter voorkoming van onnodig kopieëren van parameters, of om PRP's default vertaling van C's parametermechanisme naar het copy mechanism van RPCGEN (zie punt c, pagina 61) te voorkomen, kan de programmeur `in`, `out` of `inout` voor de eigenlijke parameterdeclaratie zetten.

A.2.2 Parametertypen

Hieronder wordt de parameterdeclaratiesyntax en de interpretatie daarvan van ANSI C en PRP-C behandeld.

De interpretatie van de drie groepen parametertypen—simple, string, array—is in C als in onderstaande tabel. PRP moet zich hieraan houden. Dit is een belangrijke eis voor transparantie.

C value/reference

type	*overdracht*
simple	value
string	reference
array	reference

C's call by value overdracht is in een RPC omgeving call by copy. C's call by reference (eigenlijk call by pointer*value*) is in RPC context call by copy/restore. Zie ook pagina 61.

Input parameters worden in RPCGEN's in-structure overgezonden (call by copy). Output parameters in RPCGEN's out-structure. Input/output parameters worden in beide overgezonden (call by copy/restore).

schema's PRP input

De volgende twee tabellen beschrijven de input die PRP moet interpreteren. Een remote procedure prototype zou er bijvoorbeeld zo uit kunnen zien:

```
remote int bereken(in  float a[10], in  int na,
                   in  float b[10], in  int nb,
                   out float c[10], out int *nc);
```

Het woord `remote` geeft aan dat het om een RPC declaratie gaat. De woorden `in` en `out` geven de richting aan van de parameters. Deze procedure zou volgens PRP's default regels ook werken, zonder de `in`s en `out`s. Als voorbeeld is 10 voor de arraydimensie gekozen.

In § A.3.1 worden de PRP-stubs beschreven die op basis van dit prototype gegenereerd zouden moeten worden. Op pagina 82 is de bijbehorende `.x`-file afgedrukt.

De tabel hieronder geeft de interpretatie die het PRP-prototype geeft aan de standaard C declaratie syntax.

PRP default

type	*value (geen ster)*		*reference (wel ster)*	
simple	int n	**in**	int *n	**in/out**
	char c	**in**	char *c	–
string	char *s	**in/out**	char *s	**in/out**
array	int a[10]	**in/out**	int *a[10]	**in/out**
	char s[10]	**in/out**	char *s[10]	**in/out**

Het uitgangspunt is dat een sterretje slechts by reference (in/out) kan betekenen. `char` is een uitzondering, daar betekent een sterretje `NULL`-terminated string, ook by reference. Rechte haken duiden een array aan, eveneens altijd by reference. Ook bij `char` betekenen rechte haken array, een `char c[]` hoeft niet `NULL`-terminated te zijn.

Ondanks het feit dat strings en arrays wanneer ze zonder ster gedeclareerd zijn in C by reference worden overgedragen zijn ze in de value kolom opgenomen. In deze tabel ligt de nadruk op de syntactische aspecten van een declaratie, op wel of geen ster. Strings en arrays kunnen volgens PRP's defaultregels niet by value worden overgedragen. In de volgende tabel is beschreven hoe dat wel mogelijk is.

De volgende tabel geeft vanuit het gezichtspunt van de applicatieprogrammeur aan wat men moet declareren om de gewenste overzendrichting (call by copy, restore of copy/restore) te bereiken.

PRP declaraties

type	*copy* *in* *value*	*restore* *out*	*copy/restore* *in/out* *reference*	
simple	int n		int ∗n	*standaard C*
	in int n	**out** int ∗n	**inout** int ∗n	*PRP extensie*
	in int ∗n			*ontraden*
	char c			*standaard C*
	in char c			*PRP extensie*
string			char ∗s	*standaard C*
			char ∗∗s	*ontraden*
	in char ∗s	**out** char ∗s	**inout** char ∗s	*PRP extensie*
			inout char ∗∗s	*ontraden*
array			int a[10]	*standaard C*
			int ∗a[10]	*ontraden*
	in int a[10]	**out** int a[10]	**inout** int a[10]	*PRP extensie*
	in int ∗a[10]	**out** int ∗a[10]	**inout** int ∗a[10]	*ontraden*

A.3 Werking stubs

Het doel van de PRP-stubs is om de RPC en de lokale procedureaanroep van C qua syntax en semantiek gelijk te laten zijn. Zoals het voorbeeldprogramma in appendix C aangeeft lijkt dit op het eerste gezicht aardig te lukken. De PRP-stub-aanroep lijkt qua uiterlijk meer op een lokale C functieaanroep dan een RPCGEN-stub-aanroep. Intern maken PRP-stubs gebruik van RPCGEN-stubs, en ze hebben te maken met dezelfde problemen.

Deze problemen worden in § A.4 beschreven; hier wordt ingegaan op de werking van de PRP-stubs zoals ze door het PRP-prototype worden gegenereerd.

De PRP-clientstub bevat de aanroep van de RPCGEN-clientstub. Deze zorgt er voor dat het RPC-subsysteem aan de server-kant de RPCGEN-serverstub in het serverproces aanroept, die op zijn beurt de PRP-serverstub aanroept. De PRP-serverstub roept vervolgens de originele serverfunctie aan die de programmeur geschreven heeft.

De code van de PRP-clientstub ziet er globaal als volgt uit:

- initialiseer het RPC systeem: bepaal waar de server procedure zich bevindt en voer de `clnt_create()` aanroep uit. Deze initialisatie vindt voor elke serverprocedure eenmalig plaats. Het is met de stubs die dit prototype genereert niet mogelijk van server te wisselen tussen twee RPC's naar dezelfde procedure.
- vul de RPCGEN inputparameter structure met de eigenlijke parameters.
- roep de RPCGEN-clientstub aan. Dit is de 'eigenlijke' RPC. Het serializen van parameters vindt in de RPCGEN-stubs plaats.
- ga na of het resultaat de `NULL`-pointer is. Zo ja, geef een foutmelding. (Het PRP-*prototype* stopt het proces bij een RPC-systeemfout; volgens het PRP-*model* zou de foutcode in een globale variabele gesignaleerd moeten worden.)
- pak de RPCGEN-outputparameter-structure uit en vul de eigenlijke uitvoer parameters alsmede de returnwaarde van de procedure.

De PRP-serverstub is eenvoudiger. Deze kent de volgende functies:

- pak de RPCGEN-inputparameter-structure uit en vul de inputparameter-variabelen van de serverprocedure die de programmeur geschreven heeft.
- roep de gebruikers serverprocedure aan volgens de standaard C syntax.
- vul de RPCGEN outputparameter-structure met de output parameters en de returnwaarde van de gebruikersprocedure. De controle gaat nu terug naar de RPCGEN-stub, die de parameters gaat serializen.

A.3.1 Schema's PRP-stubs

In de nu volgende tabellen wordt besproken welke output PRP voor de stubs moet genereren. De datatypes die worden onderscheiden zijn: simple, string en statisch array.

In § 5.2.2 zijn de achtergronden van de te genereren code behandeld.

De PRP-client-stub ziet er bijvoorbeeld globaal als volgt uit:

```
/*
** declaratie
*/

int PRP_STUB_bereken(float a[10], int na,
                     float b[10], int nb,
                     float c[10], int *nc)
```

```
{
   /*
   ** declaratie van de RPCGEN parameters
   */

   bereken_IN_t  bereken_IN;
   bereken_UIT_t *bereken_UIT;

   /*
   ** initialisatie
   */
   if (geen @servernaam@)
      servernaam = clnt_broadcast(PROG, VERS, procedurenummer);
   adres = clnt_create(servernaam, PROG, VERS, "tcp");
   if (adres not ok)
      error("RPC: Initialisatie error!!!");

   /*
   ** vul de RPCGEN input structure
   */
   bereken_IN.a.a_val = a;
   bereken_IN.a.a_len = 10;
   bereken_IN.na = na;
   bereken_IN.b.b_val = b;
   bereken_IN.b.b_len = 10;
   bereken_IN.nb = nb;

   /*
   ** roep de RPCGEN stub aan - dit is de eigenlijke RPC!
   */
   bereken_UIT = bereken_13(&bereken_IN, adres);

   /*
   ** foutafhandeling van RPC errors
   */
   if (bereken_UIT == NULL)
      error("RPC: systeemfout!!!");

   /*
   ** pak de RPCGEN output structure uit
   */
   *nc = bereken_UIT -> nc;
   memcpy(c, bereken_UIT -> c.c_val, 10 * sizeof(float));

   return (bereken_UIT -> resultaat);
}
```

In dit voorbeeld ligt de nadruk op de veranderlijke code. Dat is de code die parameters bewerkt. De vaste code, de initialisatie en de foutafhandeling zijn verkort weergegeven om redenen van overzichtelijkheid.

De tabel hieronder beschrijft de declaratie die PRP moet genereren voor de PRP-clientstub.

PRP-clientstub declaratie

type	*in*	*out*	*in/out*
simple	int n	int *n	int *n
string	char *s	char *s	char *s
array	int a[10]	int a[10]	int a[10]

De volgende tabel beschrijft de code die PRP moet genereren voor de PRP-clientstub zodat de RPCGEN input structure gevuld wordt. Bij arrays wordt als voorbeeld voor de cardinaliteit 10 gekozen.

PRP-clientstub in-structure vullen

type	*in*	*out*	*in/out*
simple	IN.n = n	–	IN.n = *n
string	IN.s = s	–	IN.s = s
array	IN.a.a_val = a	–	IN.a.a_val = a
	IN.a.a_len = 10	–	IN.a.a_len = 10

De tabel hierna beschrijft de code die PRP moet genereren voor de PRP-clientstub zodat de RPCGEN output structure uitgepakt wordt en de actuele gebruiker parameters gevuld worden.

PRP-clientstub uit-structure uitpakken

type	*in*	*out*	*in/out*
simple	–	*n = UIT→n	*n = UIT→n
string	–	s = UIT→s	s = UIT→s
array	–	memcpy(a, UIT→a.a_val, 10 * sizeof(int));	memcpy(a, UIT→a.a_val, 10 * sizeof(int));

Arrays worden hier met `memcpy()` gekopieerd. De pointer naar de eerste waarde van een array is namelijk een `const` pointer. Deze beginwaarde van het array mag niet gewijzigd worden [Kernighan 2]. Een assignment van de variabele zelf (de pointer naar de datastructuur) als bij strings is hierdoor niet mogelijk. Er zit niets anders op dan de datastructuur in zijn geheel te kopiëren.

PRP-server-stub

De volgende tabellen gaan over de PRP-serverstub. Hieronder ziet u net als bij de client-stub een voorbeeld hoe de te genereren code er globaal uit zou moeten zien. De declaratie moet aansluiten bij de aanroep van deze functie in de PRP-client-stub.

```
/*
** declaratie server stub
*/

bereken_UIT_t *bereken_13(bereken_IN_t * bereken_IN)
{
   static bereken_UIT_t bereken_UIT;

   /*
   ** declaratie van de lokale variabelen
   */

   float *a;
   int na;
   float *b;
   int nb;
   float *c;
   int nc;

   /*
   ** vul de parameters uit de RPCGEN input structure
   */
   a = bereken_IN -> a.a_val;
   na = bereken_IN -> na;
   b = bereken_IN -> b.b_val;
   nb = bereken_IN -> nb;

   /*
   ** roep de gebruikers server functie aan
   */
   bereken_UIT.resultaat = PRP_PROC_bereken(a, na,
                                            b, nb,
                                            c, &nc);

   /*
   ** vul de RPCGEN output structure
   */
   bereken_UIT.c.c_val = c;
   bereken_UIT.c.c_len = 10;
   bereken_UIT.nc = nc;
```

```
   return (&bereken_UIT);
}
```

Onderstaande tabel beschrijft de code die PRP moet genereren voor de PRP-serverstub om de lokale variabelen te declareren die als parameters aan de gebruikers server functie zullen worden meegegeven.

PRP-serverstub variabelen-declaratie

type	*in*	*out*	*in/out*
simple	int n	int n	int n
string	char *s	char *s	char *s
array	int *a	int *a	int *a

De volgende tabel beschrijft de code die PRP moet genereren voor de PRP-server-stub zodat de RPCGEN input structure uitgepakt wordt en de actuele gebruiker parameters gevuld worden.

PRP-serverstub in-structure uitpakken

type	*in*	*out*	*in/out*
simple	n = IN→n	–	n = IN→n
string	s = IN→s	–	s = IN→s
array	a = IN→a.a_val	–	a = IN→a.a_val

Hieronder ziet u de code die PRP moet genereren voor de PRP-server-stub om de gebruikers-serverfunctie op de juiste wijze aan te roepen.

PRP-serverstub aanroep

type	*in*	*out*	*in/out*
simple	n	&n	&n
string	s	s	s
array	a	a	a
structure	r	&r	&r

De laatste tabel beschrijft de code die PRP moet genereren voor de PRP-server-stub zodat de RPCGEN-outputstructure wordt gevuld met de gebruikerparameters die teruggezonden moeten worden.

PRP-serverstub uit-structure inpakken

type	*in*	*out*	*in/out*
simple	–	UIT.n = n	UIT.n = n
string	–	UIT.s = s	UIT.s = s
array	–	UIT.a.a_val = a	UIT.a.a_val = a
	–	UIT.a.a_len = 10	UIT.a.a_len = 10

In C kan de semantiek van een outputparameter via het `return`-statement en het returntype van de functie bereikt worden. PRP ondersteunt dit door een returnwaarde als een outputparameter te behandelen.

A.4 Problemen

Hier worden mogelijke problemen die zich kunnen voordoen bij het schrijven van een PRP-applicatie besproken. Deze problemen zijn terug te voeren op de slechte integreerbaarheid van de sequentiële concepten uit de taal C met de gedistribueerde Sun omgeving (zie ook § 3.2).

A.4.1 C in een gedistribueerde omgeving

adresruimte

In Sun's gescheiden-geheugen omgeving zijn geheugenreferenties tussen verschillende processoren vanwege inefficiëntie niet geïmplementeerd. De volgende concepten van C worden hierdoor beïnvloed:

- *Globale variabelen* In een PRP-applicatie kunnen verschillende processen geen gemeenschappelijke globale variabelen bezitten. De reden hiervoor is bij punt a in § 3.4.2 besproken. De mogelijkheid van `remote`-functies naast `extern`-functies doet wellicht het bestaan van `remote`-variabelen naast `extern`-variabelen vermoeden. Deze variabelen zijn echter niet mogelijk bij PRP.

  De programmeur moet er zelf opletten dat globale variabelen, net als lokale variabelen een beperkte *scope* hebben. De scope van een lokale variabele is tot de procedure beperkt, en de scope van een 'globale' variabele tot het proces.

- *Copy mechanism versus definitional mechanism* Een PRP RPC past het copy mechanism voor parameteroverdracht toe. De reden is efficiëntie (zie hiervoor punt c in § 3.4.2). De semantiek van call by copy/restore verschilt op een paar subtiele punten van de semantiek van call by reference. Een voorbeeld van de onverwachte problemen die dit kan veroorzaken is op pagina 61 te vinden. Dit soort fouten zullen waarschijnlijk niet frequent voorkomen. Àls ze voorkomen zullen ze, omdat men ze niet verwacht, moeilijk te traceren zijn. Een goed begrip van het verschil tussen de twee parameteroverdracht-mechanismen is hier onontbeerlijk.

De parameteroverdracht is ook bij PRP-applicaties niet volledig transparant.

- *Pointers en RPC-parameters* Geheugenreferenties tussen verschillende processoren zijn om redenen van efficiëntie niet geïmplementeerd. Bij parameteroverdracht volgens het copy mechanism—bij RPC's—worden de (onderliggende) waarden van variabelen overgebracht, gegevens over de lokatie ervan zijn alleen zinvol om de onderliggende waarde te bereiken. In principe kan men als RPC-parameter een pointervariabele specificeren—bijvoorbeeld `int ***n` in plaats van `int n`. Door de pointer af te lopen kan de clientstub de onderliggende waarde bereiken, en de stub aan de serverkant kan de reeks verwijzingen weer opbouwen. (Op deze wijze vindt het serializen en deserializen van lijsten en bomen plaats.)

  Echter, het specificeren van een pointervariabele als parameter in een RPC waar ook de waarde zelf gespecificeerd zou kunnen worden—zoals bij een enkele integer—is overbodig. Bij enkelvoudige datatypen vindt de overdracht minder efficiënt plaats, en het kan verwarrend werken of duiden op een denkfout bij de programmeur.

typing

De taal C is *weakly typed*. Dit houdt in dat het type van een variabele tijdens het verloop van een programma kan veranderen. Het is in principe mogelijk dat het actuele type van een variabele op het moment van aanroep van een remote-procedure afwijkt van het gedeclareerde type, bijvoorbeeld door een *typecast*. Zeker bij pointervariabelen komt dit vrij vaak voor. Ook komt het nogal eens voor dat een variabele die als pointer-naar-`char` gedeclareerd wordt, als pointer-naar-`double` (of nog iets anders) gebruikt wordt.

Deze handelwijzen lopen spaak bij parameters van PRP RPC's. PRP genereert XDR-conversiecode op basis van de declaratie die wordt afgeleid uit het `remote`-prototype.[3] Voor de conversiecode van parameters wordt alleen naar dit `remote`-prototype gekeken, at compile time. De programmeur moet er voor zorgen dat het actuele type bij de RPC overeen komt met het gedeclareerde type uit het `remote`-prototype.

Het feit dat het gedeclareerde type niet altijd uitsluitsel geeft over het type at run-time levert meer problemen voor het PRP-model op. In § 6.3.2 is aan de orde geweest dat het actuele datatype van een C-union niet uit de declaratie is af te leiden, en dat aan de declaratie van een pointer niet te zien is of de pointer naar een enkelvoudige datastructuur wijst, of naar de beginwaarde van een array. Het is niet mogelijk om vast te stellen hoeveel

[3] Als de parametertypen in de serverfunctie-definitie afwijken van die van het prototype, worden deze door PRP genegeerd—de serverfunctie-definitie wordt niet door PRP geanalyseerd. Deze inconsistentie zal door de meeste compilers op zijn minst gesignaleerd worden met een waarschuwing.

gegevenselementen de stubs moeten overzenden. C moet hiertoe met nog meer syntaxelementen worden uitgebreid.

Ook bij strings en recursieve datastructuren[4] als bomen, gelinkte lijsten en grafen schiet de taaldefinitie van C tekort om foutvrije RPC-parameteroverdracht te garanderen. Uit de declaratie van deze datatypen is de omvang niet at compile-time af te leiden. Het ontbreekt ook aan een betrouwbare methode om at run-time de omvang (cardinaliteit) vast te stellen. Wel is het zo dat de conventie bestaat dat het laatste element de waarde `NULL` bevat. De XDR-routines die RPCGEN genereert gaan er vanuit dat de programmeur zich aan deze conventie houdt.

relatieve traagheid

Niet elke gebruikerstoepassing is geschikt om op een parallel systeem geïmplementeerd te worden; sommige algoritmes zijn inherent sequentieel. Van de klasse toepassingen waarvoor parallelle implementatie wel zinvol is, kent een gedeelte fine grain parallellisme, en een gedeelte large grain. Alleen van toepassingen met large grain parallellisme is het zinvol implementatie op gedistribueerde hardware te overwegen (zie § 2.1.4).

De reden hiervoor is dat de communicatie in een gescheiden-geheugen omgeving relatief traag is. Een RPC is door de communicatieoverhead aanmerkelijk trager dan een intraproces-procedureaanroep. Naarmate er minder frequent wordt gecommuniceerd—'grote' procedures—zal deze overhead een kleiner probleem worden.

Dit impliceert dat bij de partitionering in modules tijdens de ontwerpfase rekening moet worden gehouden met de traagheid van een RPC. Zoals in § 3.4.5 en in punt f op pagina 66 is opgemerkt brengt de transparantie die bij RPC-implementatie vaak wordt nagestreefd ook problemen met zich mee. Het zou kunnen zijn dat de illusie ontstaat dat een RPC net zo eenvoudig en net zo vaak als een lokale procedureaanroep kan worden gedaan. *Men moet een RPC met mate—vanwege de traagheid—en zorgvuldig—vanwege de subtiele semantische verschillen—gebruiken.*

A.4.2 Tekortkomingen van het PRP-prototype

Bij het gebruik van het PRP-prototype moet de programmeur rekening houden met de volgende tekortkomingen van de huidige versie:

- *Netwerkadres* Wanneer voor een bepaalde remote-procedure eenmaal een server is vastgesteld, is het niet mogelijk deze bij een volgende RPC naar dezelfde remote-procedure te verwisselen voor een andere. De netwerknaam die de eerste keer is opgegeven blijft geldig gedurende het gehele programma.

[4]Recursieve datastructuren zijn niet in het PRP-prptotype geïmplementeerd. Omdat de problematiek dezelfde achtergrond heeft—C is weakly typed—worden ze hier toch genoemd.

Dit zou men kunnen oplossen door voor de verschillende servers evenzoveel verschillende RPC's te maken, en de aanroep ervan in een `case`-statement te plaatsen.

Achter de netwerknaam moet een tweede @-symbool geplaats worden—`proc@node@`—in tegenstelling tot de adressering die onder andere bij e-mail gebruikelijk is—`user@node`.

- *Diepe indirectie* Wanneer men onnodig[5] veel pointerverwijzingen voor een RPC-parameter specificeert—`int **n` in plaats van `int n`—negeert PRP de overbodige indirectieniveaus, en brengt ze terug tot het minimum: geen pointers bij call by value, en één pointer bij call by reference. PRP genereert hierover een mededeling.

  Hoewel deze handelwijze in overeenstemming is met § A.4.1: *Pointers en RPC-parameters*, kan de rigide afwijzing van mogelijk correcte programmatuur tot veel onnodig werk bij de applicatieprogrammeur leiden. Een vriendelijker werkwijze is wellicht het genereren van een waarschuwing, en toch de gespecificeerde pointerverwijzingen door de PRP-stubs te laten aflopen en nabouwen.

- *RPC systeemfouten* Bij het optreden van een RPC systeemfout stopt de PRP-stub de verwerking. Het is wenselijk om deze reactie wat te kunnen nuanceren. Een alternatieve oplossing is om door middel van globale variabelen het gedrag van de PRP-stub te beïnvloeden. De PRP-stub kan op zijn beurt ook door middel van een globale variabele een foutcode aan de rest van het programma doorgeven.

- *String* PRP kent geen expliciet string datatype. Een parameter die als `char *s` gedeclareerd wordt, wordt als string behandeld. Het is met dit prototype niet mogelijk een enkelvoudig karakter als referenceparameter te specificeren.

  Een oplossing voor dit probleem is de invoering—net als bij RPCGEN—van het expliciete datatype '`string`'. Een `char *s`-parameter zou in dit geval als een enkelvoudig karakter als referenceparameter worden opgevat.

- *Datatypen* Zoals in de lijst op pagina 122 is beschreven ondersteunt dit prototype nog geen structures, meerdimensionale arrays, en dynamisch gealloceerde arrays als parametertype.

A.4.3 Afsluiting

Een verschil tussen het programmeren van een RPCGEN-applicatie en een PRP-applicatie is dat men bij RPCGEN meer code moet schrijven. Het schrijven van een PRP-applicatie kost wel minder werk, maar niet minder kennis van C, RPCGEN en gedistribueerd programmeren.

[5]Dat wil zeggen: onnodig om het verschil tussen value- en reference parameter aan te geven.

De door velen geroemde vrijheid die de taal C haar gebruikers laat betekent ook dat de verantwoordelijkheid voor de correctheid van het programma meer bij de programmeur ligt: deze moet zich goed realiseren wat de consequenties van bepaalde 'details' zijn. Waar men door de regels van sommige andere talen voor het maken van fouten wordt behoed—zie 'typesecurity' op pagina 53—moet men in C in een gedistribueerde omgeving veel kennis over een complexe verzameling feiten en eigenaardigheden hebben—zoals het copy/restore mechanisme, het declaratie-type van een pointer, de consequenties van een ongeïnitialiseerde pointer, en de scope van globale variabelen.

In de taal C worden pointervariabelen veelvuldig gebruikt omdat het (a) de enige mogelijkheid is om referenceparameters en recursieve datastructuren te gebruiken en het (b) zeer efficiënte uniproces programma's mogelijk maakt [Kernighan 2]. Pointervariabelen bevatten geheugenadressen; C programmeurs moeten veelvuldig met geheugenadressen manipuleren. De essentie van gedistribueerde systemen is juist *gescheiden* geheugen (§ 2.3.1), waarin geheugenadressen in verschillende delen van een programma naar verschillende waarden wijzen. Het programmeren van gedistribueerde applicaties in C vereist—mede omdat C niet typesecure is en fouten niet door de taal gesignaleerd worden—de nodige voorzichtigheid en het nodige inzicht. Voor het programmeren van hoog niveau gedistribueerde *toepassingsprogrammatuur* (in tegenstelling tot systeemprogrammatuur) is C minder geschikt.

Appendix B

Voorbeeldprogramma

RPCGEN

Deze voorbeeldapplicatie bestaat uit een client- en twee servermodules. De client module leest twee arrays uit een file in. De ene server berekent de produktvector van deze twee arrays. De andere server drukt de resultaatvector af op de stdoutput van die server. Deze versie is geschikt voor verwerking door RPCGEN.

De files zijn achtereenvolgens:

- calc.h
- interface.x
- calc_ber.c
- calc_pr.c
- calc.c

headerfile

Onderstaande headerfile bevat de globale definities.

```
/*
** calc.h
*/

#define OK 0
#define ERR -1

#define OMVANG 10

#define REKENSERVER "slc09.cs.few.eur.nl"
```

---

interfacedefinitie

De interfacedefinitie die door RPCGEN gecompileerd wordt:

```
/*
** interface.x
** interfacedefinitie RPCGEN illustratie
*/

/*
** Function 'druk_af' -- input parameters
*/
struct IN_druk_af_t {
   string antws<>;
   int n;
   float vector<>;
};


/*
** Function 'druk_af' -- output parameters
*/
struct UIT_druk_af_t {
   string RES_resultaat<>;
};


/*
** Function 'bereken' -- input parameters
*/
struct IN_bereken_t {
   float c<>;
   int nb;
   float b<>;
   int na;
   float a<>;
};


/*
** Function 'bereken' -- output parameters
*/
struct UIT_bereken_t {
   int RES_resultaat;
   int nc;
   float c<>;
};


/*
** Versionnumbers for program and remote routines
*/

program PROG {
   version VERS {
      UIT_druk_af_t druk_af(IN_druk_af_t) = 2;
      UIT_bereken_t bereken(IN_bereken_t) = 1;
   } = 13;                     /* dit is versie dertien */
 } = 0x2000076;
```

rekenserver

```
/*
** calc_ber.c
** RPCGEN versie van rekenserver
*/

#include "calc.h"

#include <rpc/rpc.h>     /* verplicht voor RPCGEN programma's */
#include <stdlib.h>
#include "interface.h"   /* door RPCGEN gegenereerd */


/*
** Function 'bereken' -- prp server stub
*/
UIT_bereken_t *bereken_13 (IN_bereken_t * IN_bereken)
{
   static UIT_bereken_t UIT_bereken;

   int nc;
   float * c;
   int nb;
   float * b;
   int na;
   float * a;
   int i;

   c  = IN_bereken -> c.c_val;  [1]
   nb = IN_bereken -> nb;
   b  = IN_bereken -> b.b_val;
   na = IN_bereken -> na;
   a  = IN_bereken -> a.a_val;

   xdr_free(xdr_UIT_bereken_t, &UIT_bereken);

   nc = (na > nb) ? na : nb;

   for (i = 0; i < nc; i++) {
      c[i] = ((i < na && i < nb) ? a[i] * b[i] : (float)0.0);
   }

   UIT_bereken.RES_resultaat = OK;

   UIT_bereken.nc = nc;
   UIT_bereken.c.c_val = c;
   UIT_bereken.c.c_len = OMVANG;
```

[1] Om te voorkomen dat ingewikkelde structure-expressies het algoritme ondoorzichtig maken worden de RPC-parameters eerst naar lokale variabelen gekopieerd—het in- en uitpakken van de parameters.

```
   return (&UIT_bereken);
}
```

afdrukserver

```
/*
** calc_pr.c
** RPCGEN versie van printserver
*/


#include <rpc/rpc.h>      /* verplicht voor RPCGEN programma's */
#include <stdlib.h>
#include "interface.h"    /* door RPCGEN gegenereerd */

/*
** Function 'druk_af' -- prp server stub
*/
UIT_druk_af_t *druk_af_13 (IN_druk_af_t * IN_druk_af)
{
   static UIT_druk_af_t UIT_druk_af;

   char * antws;
   int n,
       i,
       resultaat;
   float * vector;

   antws  = IN_druk_af -> antws;
   n      = IN_druk_af -> n;
   vector = IN_druk_af -> vector.vector_val;

  /* Geef geheugen vrij dat door de RPCGEN-stubs voor het */
  /* serializen gealloceerd is. (Zie Sun manual) */
   xdr_free(xdr_UIT_druk_af_t, &UIT_druk_af);

   printf("Het resultaat van de berekening (%s) is:\n", antws);

   for (i = 0;
        i < n && (resultaat = printf("%f\n", vector[i])) > 0;
        i++)
      ;

   if (i != n || resultaat < 0) {
      UIT_druk_af.RES_resultaat = "Wij gingen fout";
   } else {
      UIT_druk_af.RES_resultaat = "Wij zijn OK!";
   }

   return (&UIT_druk_af);
}
```

client

```
/*
** Calc.c
** I/O module van een prg dat het volgende doet:
** 1. lees twee arrays in
** 2. roep een (remote) routine aan die het product berekent en
**    als een vector terugstuurt.
**
**    RPCGEN versie van de client
**
*/

/*
** werkwijze van main:
** roep aan met parameter file naam.
** 1. bepaal dimensies
** 2. alloceer geheugen voor vectoren
** 3. vul vectoren 1 en 2
** 4. bereken; resultaat in vector 3 (c)
** 5. druk af op stdout
*/

#include "calc.h"

#include <stdio.h>
#include <rpc/rpc.h>      /* verplicht voor RPCGEN programma's */
#include "interface.h"    /* door RPCGEN gegenereerd */

/* lokale prototypes */
static int lees_omvang (int *na,
                        int *nb,
                        FILE *datafp);

static int lees_vector (float vector[],
                        int n,
                        FILE *datafp);


int main(int argc, char *argv[])
{
   float a[OMVANG], b[OMVANG], c[OMVANG];
   int   na, nb, nc;
   FILE *datafp;
   int   antw;
   char *str;

   /* RPC structures */
```

```
static CLIENT * CL;
IN_bereken_t    IN_bereken;
UIT_bereken_t * UIT_bereken;
IN_druk_af_t    IN_druk_af;
UIT_druk_af_t * UIT_druk_af;


if (argc != 3) {
   fprintf(stderr, "Usage: %s servername datafile\n", argv[0]);
   exit(1);
}

if ((datafp = fopen(argv[2], "r")) == 0) {
   fprintf(stderr, "%s: cannot open datafile %s\n", argv[0], argv[2]);
   exit(7);
}

if (lees_omvang(&na, &nb, datafp) != 0
    || na > OMVANG || nb > OMVANG ) {
   fprintf(stderr, "%s: corrupt dimension in datafile %s\n", argv[0], argv[2]);
   exit(2);
}

if (lees_vector(a, na, datafp) != 0) {
   fprintf(stderr, "%s: corrupt vector 1 data in datafile %s\n", argv[0], argv[2]);
   exit(3);
}

if (lees_vector(b, nb, datafp) != 0) {
   fprintf(stderr, "%s: corrupt vector 2 data in datafile %s\n", argv[0], argv[2]);
   exit(4);
}

fclose(datafp);

/*
** Initialisation of RPC system by servername
*/
if ((CL = clnt_create(REKENSERVER, PROG, VERS, "tcp")) == NULL) { [2]
   fprintf(stderr, "ERROR 1, (%s:%d): Cannot find server.
    Contact your system administrator.\n", __FILE__, __LINE__);
   clnt_pcreateerror(REKENSERVER);    /* RPCGEN fout-routine */
   exit(99);   /* foutcode ter onderscheiding van RPC fouten */
}

/* vul parameter structure */
IN_bereken.c.c_val = c;
IN_bereken.c.c_len = OMVANG;
IN_bereken.nb = nb;
IN_bereken.b.b_val = b;
```

[2] Hier is voor TCP gekozen in verband met maximum pakketomvang bij UDP, ook al is de serverprocedure idempotent.

```
IN_bereken.b.b_len = OMVANG;
IN_bereken.na = na;
IN_bereken.a.a_val = a;
IN_bereken.a.a_len = OMVANG;

/* aanroep van de RPCGEN-stub; de RPC */
UIT_bereken = bereken_13 (&IN_bereken, CL);

/*
** RPC default error mechanism
*/
if (UIT_bereken == NULL) {
   fprintf(stderr, "ERROR 2, (%s:%d): RPC System Error.
    Contact your system administrator.\n", __FILE__, __LINE__);
   clnt_perror(CL, REKENSERVER);
   exit(98);
}

/* uitpakken van de output parameters */
nc = UIT_bereken -> nc;
memcpy(c, UIT_bereken->c.c_val, OMVANG * sizeof(float));
antw = UIT_bereken -> RES_resultaat;

if (antw != 0) {
   fprintf(stderr, "%s: calculation failed\n", argv[0]);
   exit(5);
}


/* RPC naar print server */
/*
** Initialisation of RPC system by servername
*/
if ((CL = clnt_create(argv[1], PROG, VERS, "tcp")) == NULL) {
   fprintf(stderr, "ERROR 1, (%s:%d): Cannot find server.
    Contact your system administrator.\n", __FILE__, __LINE__);
   clnt_pcreateerror(argv[1]);
   exit(99);
}

/* inpakken vam parameters */
IN_druk_af.antws = "calc versie 3";
IN_druk_af.n = OMVANG;
IN_druk_af.vector.vector_val = c;
IN_druk_af.vector.vector_len = OMVANG;

              /* de tweede RPC */
UIT_druk_af = druk_af_13 (&IN_druk_af, CL);

/*
** RPC default error mechanism
*/
if (UIT_druk_af == NULL) {
```

```
      fprintf(stderr, "ERROR 2, (%s:%d): RPC System Error.
       Contact your system administrator.\n", __FILE__, __LINE__);
      clnt_perror(CL, argv[1]);
      exit(98);
   }

   str = UIT_druk_af -> RES_resultaat;

   if (str == NULL) {
      fprintf(stderr, "%s: cannot print results\n", argv[0]);
      exit(6);
   }

   printf("druk_af: %s.\n", str);

   return (0);
}


/*
** lees de twee array grenzen in
** geef fout terug bij i/o error
*/

static int lees_omvang (int *na,
                        int *nb,
                        FILE *datafp)
{
   if (fscanf(datafp, "omvang: %d %d", na, nb) <= 0) {
      return (ERR);
   } else {
      return (OK);
   }
}


/*
** lees waarden uit file datafp tot: n waarden gelezen zijn
**                                of: EOF bereikt wordt
** retourneer errorwaarde bij i/o error of early EOF
*/

static int lees_vector (float vector[],
                        int n,
                        FILE *datafp)
{
   int i,
       resultaat;

   for (i = 0;
        i < n && (resultaat =
              fscanf(datafp, " waarde: %f", &(vector[i]) )) > 0;
        i++)
```

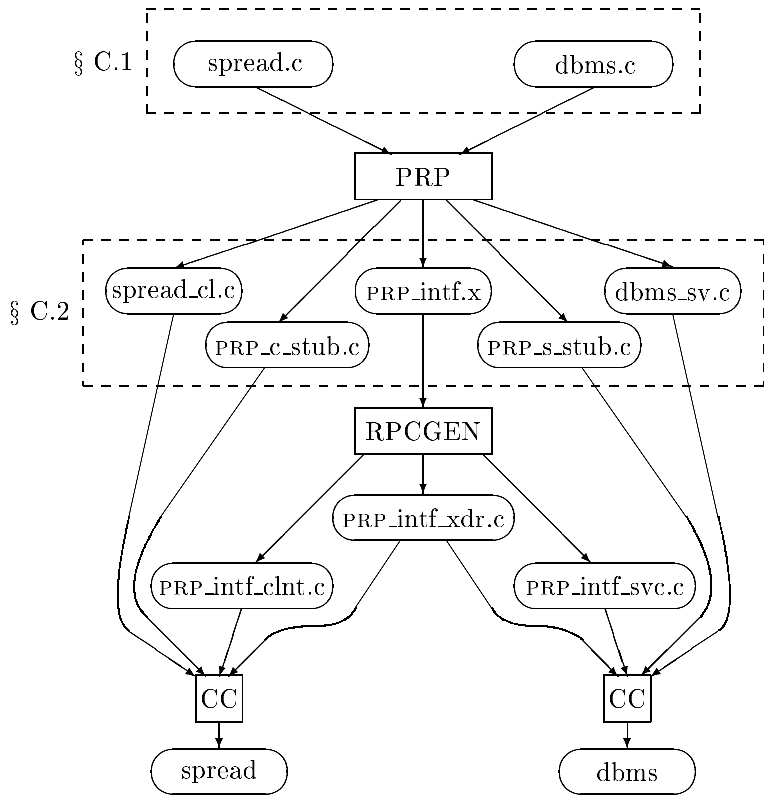
```
      ;

   if (i != n || resultaat < 0) {
      return (ERR);
   } else {
      return (OK);
   }
}
```

Appendix C

Voorbeeldprogramma PRP

Deze voorbeeldapplicatie heeft dezelfde functionaliteit als het vorige programma. Deze versie is voor PRP geschikt. In § C.1 worden de sourcefiles van de applicatieprogrammeur gegeven. Nadat deze door PRP verwerkt zijn, resulteert de output van § C.2. In figuur C.1 wordt dit proces nog eens globaal weergegeven—maar dan voor een applicatie met één server.

Figuur C.1: PRP files



C.1 PRP-applicatiecode

Het commando `prp calc.c calc_ber.c calc_pr.c` start de verwerking van de applicatiefiles, waarbij ook een makefile voor de verdere compilatiegang gegenereerd wordt.

De files zijn achtereenvolgens:

- calc.h
- calc_ber.c
- calc_pr.c
- calc.c

headerfile

Onderstaande headerfile bevat de prototypes van de applicatie. Zowel client als servers includen deze file.

```
/*
** calc.h
** Deze file bevat de globale prototypes
**              van de "remote" functies
*/

#define OK 0
#define ERR -1

#define OMVANG 10

                    /* het zijn STATISCHE arrays */
remote int bereken  (in float a[OMVANG],     /* in */
                     in int na,              /* in */
                     in float b[OMVANG],     /* in */
                     in unsigned short nb,   /* in */
                     out float c[OMVANG],    /* uit */
                     out int *nc);           /* uit */

remote char * druk_af  (in float vector[OMVANG],  /* in */
                        int n,
                        in char *antws);          /* in */
```

---

rekenserver

```
/*
** Calc_ber.c
** reken module van de PRP illustratie.
** Deze module draait op de server.
*/

#include "calc.h"
```

```
/*
** bereken de resultaat vector.
** c[i] = a[i] * b[i]
*/

int bereken  (float a[OMVANG],         /* in */
              int na,                  /* in */
              float b[OMVANG],         /* in */
              int nb,                  /* in */
              float c[OMVANG],         /* uit, dus ref(array) */
              int *nc)                 /* uit */
{
   int i;

   *nc = (na > nb) ? na : nb;       /* nc is max(na, nb) */

   for (i = 0; i < *nc; i++) {
      c[i] = ((i < na && i < nb) ? a[i] * b[i] : (float)0.0);
   }

   return (OK);
}
```

---

afdrukserver

```
/*
** Calc_pr.c
** afdrukmodule van de PRP illustratie.
** Deze module draait op de tweede server.
*/

#include <stdio.h>
#include "calc.h"


/*
** druk een vector af op stdout
** gebruik een library die printf fouten laat rapporteren...
*/


char * druk_af  (float vector[OMVANG], /* in */
                 int n,
                 char *antws)          /* in */
{
   int i,
       resultaat;

   printf("Het resultaat van de berekening (%s) is:\n", antws);

   for (i = 0;
```

```
      i < n && (resultaat = printf("%f\n", vector[i])) > 0;
      i++)
      ;

   if (i != n || resultaat < 0) {
      /* i/o error; te weinig waarden afgedrukt */
      return ("Wij gingen fout");
   } else {
      return ("Wij zijn OK!");
   }
}
```

client

```
/*
** Calc.c
** I/O module van een prg dat het volgende doet:
** 1. lees twee arrays in
** 2. roep een (remote) routine aan die het product berekent
**    en als een vector terugstuurt.
**
** Dit is een voorbeeld voor de Pre-RPCGEN-Processor.
*/


#include <stdio.h>            /* voor printf() en scanf() */
#include <stdlib.h>
#include "calc.h"


/* lokale prototypes */

static int lees_omvang (int *na,         /* in */
                        int *nb,         /* in */
                        FILE *datafp);   /* in */

static int lees_vector (float vector[],  /* uit */
                        int n,           /* in */
                        FILE *datafp);   /* in */

/*
** werkwijze van main:
** roep aan met parameter file naam.
** 1. vul vectoren 1 en 2
** 2. bereken; resultaat in vector 3 (c)
** 3. druk af op stdout
*/
```

```
int main(int argc, char *argv[])
{
   float a[OMVANG], b[OMVANG], c[OMVANG];
   int   na, nb, nc;
   FILE *datafp;
   int   antw;
   char *str;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s servername datafile\n", argv[0]);
      exit(1);
   }

   if ((datafp = fopen(argv[2], "r")) == NULL) {
      fprintf(stderr, "%s: cannot open datafile %s\n", argv[0], argv[2]);
      exit(7);
   }

   if (lees_omvang(&na, &nb, datafp) != OK
       || na > OMVANG || nb > OMVANG) {
      fprintf(stderr, "%s: corrupt dimension in datafile %s\n", argv[0], argv[2]);
      exit(2);
   }

   if (lees_vector(a, na, datafp) != OK) {
      fprintf(stderr, "%s: corrupt vector 1 data in datafile %s\n", argv[0], argv[2]);
      exit(3);
   }

   if (lees_vector(b, nb, datafp) != OK) {
      fprintf(stderr, "%s: corrupt vector 2 data in datafile %s\n", argv[0], argv[2]);
      exit(4);
   }

   fclose(datafp);

               /* RPC; broadcast */
   antw = bereken(a, na, b, nb, c, &nc);

   if (antw != OK) {
      fprintf(stderr, "%s: calculation failed\n", argv[0]);
      exit(5);
   }

               /* geadresseerde RPC */
   if ((str = druk_af(c, nc, "calc versie 3")@argv[1]@) == NULL) {
      fprintf(stderr, "%s: cannot print results\n", argv[0]);
      exit(6);   /* functie druk_af geeft nooit NULL terug */
   }             /* test is bij deze vorm van druk_af niet nodig */

   printf("druk_af: %s.\n", str);

   return (0);
```

```
}


/*
** lees de twee array grenzen in
** geef fout terug bij i/o error
*/
static int lees_omvang (int *na,         /* in */
                        int *nb,         /* in */
                        FILE *datafp)    /* in */
{
   if (fscanf(datafp, "omvang: %d %d", na, nb) <= 0) {
      return (ERR);
   } else {
      return (OK);
   }
}

/*
** lees waarden uit file datafp tot: n waarden gelezen zijn
**                                of: EOF bereikt wordt
** retourneer errorwaarde bij i/o error of early EOF
*/
static int lees_vector (float vector[],  /* uit */
                        int n,           /* in */
                        FILE *datafp)    /* in */
{
   int i,
       resultaat;

   for (i = 0;
        i < n
        && (resultaat = fscanf(datafp, " waarde: %f",
                                       &(vector[i]) )) > 0;
        i++)
      ;

   if (i != n || resultaat < 0) {
      /* i/o error; te weinig waarden */
      return (ERR);
   } else {
      return (OK);
   }
}
```

C.2 PRP-output

In dit deel zijn de files die PRP produceert weergegeven. De interfacedefinitie wordt verder door RPCGEN verwerkt, de overige files door de C compiler.

De files zijn achtereenvolgens:

- PRP_intf.x

- PRP_s_stub.c
- calc_ber_sv.c
- calc_pr_sv.c
- PRP_c_stub.c
- calc_cl.c

interfacedefinitie

```
/*
** PRP_intf.x
** Do not modify this file
** It was generated by PRP, Pre RPCGEN Preprocessor
*/

#include "PRP_x.h"

/*
** Function 'druk_af' -- input parameters
*/
struct _PRP_IN_druk_af_t {
   string antws<>;
   int n;
   float vector<>;
};

/*
** Function 'druk_af' -- output parameters
*/
struct _PRP_UIT_druk_af_t {
   string _PRP_RES_resultaat<>;
};

/*
** Function 'bereken' -- input parameters
*/
struct _PRP_IN_bereken_t {
   float c<>;
   unsigned short nb;
   float b<>;
   int na;
   float a<>;
};

/*
** Function 'bereken' -- output parameters
*/
struct _PRP_UIT_bereken_t {
   int _PRP_RES_resultaat;
   int nc;
   float c<>;
```

```
};


/*
** Versionnumbers for program and remote routines
*/

program _PRP_PROG {
   version _PRP_VERS {
      _PRP_UIT_druk_af_t druk_af(_PRP_IN_druk_af_t) = 3;
      void _PRP_UDP_druk_af(void) = 4;
      _PRP_UIT_bereken_t bereken(_PRP_IN_bereken_t) = 1;
      void _PRP_UDP_bereken(void) = 2;
   } = _PRP_VORIG_VERS;
} = _PRP_VORIG_PROG;
```

Onderstaande file definieert de versienummers. De applicatieprogrammeur geeft hier eenmalig waarden in op. Het versienummer wordt door PRP telkens verhoogd.

```
/*
** PRP_x.h
** PRP/RPCGEN versionnumbers; edit freely.
** This file will be overwritten at each run of PRP!
*/

#define _PRP_VORIG_PROG 0x2000076
#define _PRP_VORIG_VERS 16
```

PRP-server-stubs

Deze file bevat de PRP-stub van elke remote serverprocedure. Elke stub heeft een hulp-stub die eventuele broadcast-calls beantwoordt.

```
/*
** PRP_s_stub.c
** Do not modify this file
** It was generated by PRP, Pre RPCGEN Preprocessor
*/

#include <rpc/rpc.h>
#include <stdlib.h>
#include "PRP_intf.h"

extern char *_PRP_PROC_druk_af (
       float vector[10],       /* input */
       int n,                  /* input */
       char *antws );          /* input */

/*
** Function 'druk_af' -- prp udp broadcast companion
```

```
*/
void * _prp_udp_druk_af_16 (void)
{
   /* just answer a non-null value */
   char notnull;
   return ((void *) &notnull);
}

/*
** Function 'druk_af' -- prp server stub
*/
_PRP_UIT_druk_af_t *druk_af_16 (_PRP_IN_druk_af_t * _PRP_IN_druk_af)
{
   static _PRP_UIT_druk_af_t _PRP_UIT_druk_af;

   char * antws;
   int n;
   float * vector;

   antws = _PRP_IN_druk_af -> antws;
   n = _PRP_IN_druk_af -> n;
   vector = _PRP_IN_druk_af -> vector.vector_val;

   xdr_free(xdr__PRP_UIT_druk_af_t, &_PRP_UIT_druk_af);

   _PRP_UIT_druk_af._PRP_RES_resultaat = _PRP_PROC_druk_af (vector, n, antws);

   return (&_PRP_UIT_druk_af);
}

extern int _PRP_PROC_bereken (
       float a[10],             /* input */
       int na,                  /* input */
       float b[10],             /* input */
       unsigned short nb,       /* input */
       float c[10],             /* output */
       int *nc );               /* output */

/*
** Function 'bereken' -- prp udp broadcast companion
*/
void * _prp_udp_bereken_16 (void)
{
   /* just answer a non-null value */
   char notnull;
   return ((void *) &notnull);
}

/*
** Function 'bereken' -- prp server stub
*/
_PRP_UIT_bereken_t *bereken_16 (_PRP_IN_bereken_t * _PRP_IN_bereken)
{
   static _PRP_UIT_bereken_t _PRP_UIT_bereken;

   int nc;
   float * c;
```

```
   unsigned short nb;
   float * b;
   int na;
   float * a;

   c = _PRP_IN_bereken -> c.c_val;
   nb = _PRP_IN_bereken -> nb;
   b = _PRP_IN_bereken -> b.b_val;
   na = _PRP_IN_bereken -> na;
   a = _PRP_IN_bereken -> a.a_val;

   xdr_free(xdr__PRP_UIT_bereken_t, &_PRP_UIT_bereken);

   _PRP_UIT_bereken._PRP_RES_resultaat = _PRP_PROC_bereken (a, na, b, nb, c, &nc);

   _PRP_UIT_bereken.nc = nc;
   _PRP_UIT_bereken.c.c_val = c;
   _PRP_UIT_bereken.c.c_len = 10;

   return (&_PRP_UIT_bereken);
}
```

rekenserver

Dit is de rekenserver, `calc_ber_sv.c`. Deze file is door `cpp` berwerkt waardoor er geen commentaar in staat. De wezenlijke verandering is de naam van de procedure. Daar is nu `_PRP_PROC_` voorgezet.

```
int _PRP_PROC_bereken  (float a[10],
                        int na,
                        float b[10],
                        int nb,
                        float c[10],
                        int *nc)
{
   int i;

   *nc = (na > nb) ? na : nb;

   for (i = 0; i < *nc; i++) {
      c[i] = ((i < na && i < nb) ? a[i] * b[i] : (float)0.0);
   }

   return (0);
}
```

afdrukserver

Dit is de afdrukserver, `calc_pr_sv.c`. Deze file is ook door `cpp` berwerkt. De wezenlijke verandering is de naam van de procedure. Daar is `_PRP_PROC_` voorgezet. De door de `cpp` toegevoegde file `stdio.h` bevat geen informatie die voor het PRP-model relevant is. Vanwege de omvang zijn de desbetreffende regels niet in de onderstaande file opgenomen.

```
char * _PRP_PROC_druk_af  (float vector[10],
                           int n,
                           char *antws)
{
   int i,
       resultaat;

   printf("Het resultaat van de berekening (%s) is:\n", antws);

   for (i = 0; i < n && (resultaat = printf("%f\n", vector[i])) > 0; i++)
      ;

   if (i != n || resultaat < 0) {
      return ("Wij gingen fout");
   } else {
      return ("Wij zijn OK!");
   }
}
```

PRP-client-stubs

Deze file bevat de PRP-stubs voor de aanroepen van de remote procedures door de client.

```
/*
** PRP_c_stub.c
** Do not modify this file
** It was generated by PRP, Pre RPCGEN Preprocessor
*/

#include <rpc/rpc.h>
#include <stdlib.h>
#include <stdio.h>
#include <netdb.h>
#include "PRP_intf.h"

 /* code voor broadcast call */
static struct sockaddr_in broad_socket;

bool_t okfunc(caddr_t sp, struct sockaddr_in *sock)
{
   broad_socket = *sock;
   return (TRUE);
}
```

```
/*
** Function 'druk_af' -- prp client stub
*/
char *_PRP_STUB_druk_af (
       float vector[10],            /* input */
       int n,                       /* input */
       char *antws,                 /* input */
       char * _PRP_server_naam)
{
   static CLIENT * _PRP_CL;
   _PRP_IN_druk_af_t    _PRP_IN_druk_af;
   _PRP_UIT_druk_af_t * _PRP_UIT_druk_af;
   static char in_naam [512];
   struct hostent * host_entry, hent;
   enum clnt_stat broadresult;

   static _PRP_bestaat_al = 0;

   /*
   ** Broadcast determination of servername
   */
   if (!_PRP_bestaat_al && _PRP_server_naam == NULL) {
      /*
      ** broadcast --- try to reach the udp companion
      */
      broadresult = clnt_broadcast(_PRP_PROG, _PRP_VERS,
            4, /* procedure number */  xdr_void, NULL, xdr_void, NULL,
            (bool_t (*)(caddr_t sp, struct sockaddr_in *sock))okfunc);

      if (broadresult == RPC_SUCCESS) {
         /*
         ** convert socket address to servername
         */
         host_entry = gethostbyaddr(&(broad_socket.sin_addr.S_un.S_un_b),
                          4 * sizeof(char), broad_socket.sin_family);
         if (host_entry == NULL) {
            fprintf(stderr, "PRPERROR 4, (%s:%d): Cannot convert servername.
             Contact your system administrator.\n", __FILE__, __LINE__);
            exit(96);
         }
         hent = *host_entry;
         strncpy(in_naam, hent.h_name, 512);
         _PRP_server_naam = in_naam;
      } else {
         fprintf(stderr, "PRPERROR 3, (%s:%d): Cannot find a server using broadcast.
          Contact your system administrator.\n", __FILE__, __LINE__);
         exit(97);
      }
   }

   /*
   ** Initialisation of RPC system by servername
   */
   if (!_PRP_bestaat_al && (_PRP_CL =
       clnt_create(_PRP_server_naam, _PRP_PROG, _PRP_VERS, "tcp")) == NULL) {
      fprintf(stderr, "PRPERROR 1, (%s:%d): Cannot find server.
       Contact your system administrator.\n", __FILE__, __LINE__);
```

```
      clnt_pcreateerror(_PRP_server_naam);
      exit(99);
   }
   _PRP_bestaat_al = 1;
   _PRP_IN_druk_af.antws = antws;
   _PRP_IN_druk_af.n = n;
   _PRP_IN_druk_af.vector.vector_val = vector;
   _PRP_IN_druk_af.vector.vector_len = 10;

                    /* de eigenlijke RPC */
   _PRP_UIT_druk_af = druk_af_16 (&_PRP_IN_druk_af, _PRP_CL);

   /*
   ** RPC default error mechanism
   */
   if (_PRP_UIT_druk_af == NULL) {
      fprintf(stderr, "PRPERROR 2, (%s:%d): RPC System Error.
       Contact your system administrator.\n", __FILE__, __LINE__);
      clnt_perror(_PRP_CL, _PRP_server_naam);
      exit(98);
   }

   return (_PRP_UIT_druk_af -> _PRP_RES_resultaat);
}

/*
** Function 'bereken' -- prp client stub
*/
int _PRP_STUB_bereken (
       float a[10],             /* input */
       int na,                  /* input */
       float b[10],             /* input */
       unsigned short nb,       /* input */
       float c[10],             /* output */
       int *nc,                 /* output */
       char * _PRP_server_naam)
{
   static CLIENT * _PRP_CL;
   _PRP_IN_bereken_t    _PRP_IN_bereken;
   _PRP_UIT_bereken_t * _PRP_UIT_bereken;
   static char in_naam [512];
   struct hostent * host_entry, hent;
   enum clnt_stat broadresult;

   static _PRP_bestaat_al = 0;

   /*
   ** Broadcast determination of servername
   */
   if (!_PRP_bestaat_al && _PRP_server_naam == NULL) {
      /*
      ** broadcast --- try to reach the udp companion
      */
      broadresult = clnt_broadcast(_PRP_PROG, _PRP_VERS,
          2, /* procedure number */ xdr_void, NULL, xdr_void, NULL,
          (bool_t (*)(caddr_t sp, struct sockaddr_in *sock))okfunc);
```

```
   if (broadresult == RPC_SUCCESS) {
      /*
      ** convert socket address to servername
      */
      host_entry = gethostbyaddr(&(broad_socket.sin_addr.S_un.S_un_b),
                   4 * sizeof(char), broad_socket.sin_family);
      if (host_entry == NULL) {
         fprintf(stderr, "PRPERROR 4, (%s:%d): Cannot convert servername.
          Contact your system administrator.\n", __FILE__, __LINE__);
         exit(96);
      }
      hent = *host_entry;
      strncpy(in_naam, hent.h_name, 512);
      _PRP_server_naam = in_naam;
   } else {
      fprintf(stderr, "PRPERROR 3, (%s:%d): Cannot find a server using broadcast.
       Contact your system administrator.\n", __FILE__, __LINE__);
      exit(97);
   }
}

/*
** Initialisation of RPC system by servername
*/
if (!_PRP_bestaat_al && (_PRP_CL =
    clnt_create(_PRP_server_naam, _PRP_PROG, _PRP_VERS, "tcp")) == NULL) {
   fprintf(stderr, "PRPERROR 1, (%s:%d): Cannot find server.
    Contact your system administrator.\n", __FILE__, __LINE__);
   clnt_pcreateerror(_PRP_server_naam);
   exit(99);
}
_PRP_bestaat_al = 1;
_PRP_IN_bereken.c.c_val = c;
_PRP_IN_bereken.c.c_len = 10;
_PRP_IN_bereken.nb = nb;
_PRP_IN_bereken.b.b_val = b;
_PRP_IN_bereken.b.b_len = 10;
_PRP_IN_bereken.na = na;
_PRP_IN_bereken.a.a_val = a;
_PRP_IN_bereken.a.a_len = 10;


_PRP_UIT_bereken = bereken_16 (&_PRP_IN_bereken, _PRP_CL);

/*
** RPC default error mechanism
*/
if (_PRP_UIT_bereken == NULL) {
   fprintf(stderr, "PRPERROR 2, (%s:%d): RPC System Error.
    Contact your system administrator.\n", __FILE__, __LINE__);
   clnt_perror(_PRP_CL, _PRP_server_naam);
   exit(98);
}

*nc = _PRP_UIT_bereken -> nc;
memcpy(c, _PRP_UIT_bereken->c.c_val, 10 * sizeof(float));
```

```
   return (_PRP_UIT_bereken -> _PRP_RES_resultaat);
}
```

client

Deze file bevat de client. Na preprocessing door `cpp` zijn hier prototypes voor de serverstub-procedures toegevoegd, zodat parametertype-checking plaats vindt. De door de `cpp` toegevoegde files `stdio.h` en `stdlib.h` bevatten geen informatie die voor het PRP-model relevant is. Vanwege de omvang zijn de desbetreffende regels niet in de onderstaande file opgenomen.

```
/*
** Function 'druk_af' -- prp client stub prototype
*/
extern char *_PRP_STUB_druk_af (
       float vector[10],             /* input */
       int n,                        /* input */
       char *antws,                  /* input */
       char * _PRP_server_naam);

/*
** Function 'bereken' -- prp client stub prototype
*/
extern int _PRP_STUB_bereken (
       float a[10],              /* input */
       int na,                   /* input */
       float b[10],              /* input */
       unsigned short nb,        /* input */
       float c[10],              /* output */
       int *nc,                  /* output */
       char * _PRP_server_naam);


static int lees_omvang (int *na,
                        int *nb,
                        struct _iobuf *datafp);

static int lees_vector (float vector[],
                        int n,
                        struct _iobuf *datafp);

int main(int argc, char *argv[])
{
   float a[10], b[10], c[10];
   int   na, nb, nc;
   struct _iobuf *datafp;
   int antw;
   char *str;
```

```
   if (argc != 3) {
      fprintf((&_iob[2]), "Usage: %s servername datafile\n", argv[0]);
      exit(1);
   }

   if ((datafp = fopen(argv[2], "r")) == 0) {
      fprintf((&_iob[2]), "%s: cannot open datafile %s\n", argv[0], argv[2]);
      exit(7);
   }

   if (lees_omvang(&na, &nb, datafp) != 0) {
      fprintf((&_iob[2]), "%s: corrupt dimension in datafile %s\n",
                          argv[0], argv[2]);
      exit(2);
   }

   if (lees_vector(a, na, datafp) != 0) {
      fprintf((&_iob[2]), "%s: corrupt vector 1 data in datafile %s\n",
                          argv[0], argv[2]);
      exit(3);
   }

   if (lees_vector(b, nb, datafp) != 0) {
      fprintf((&_iob[2]), "%s: corrupt vector 2 data in datafile %s\n",
                          argv[0], argv[2]);
      exit(4);
   }

   fclose(datafp);

             /* De RPC: aanroep van de PRP-stub */
   antw = _PRP_STUB_bereken(a, na, b, nb, c, &nc, 0 /* broadcast */);

   if (antw != 0) {
      fprintf((&_iob[2]), "%s: calculation failed\n", argv[0]);
      exit(5);
   }

   if ((str = _PRP_STUB_druk_af(c, nc, "calc versie 3", argv[1])) == 0) {
      fprintf((&_iob[2]), "%s: cannot print results\n", argv[0]);
      exit(6);
   }

   printf("druk_af: %s.\n", str);

   return (0);
}

static int lees_omvang (int *na,
                        int *nb,
                        struct _iobuf *datafp)
{
   if (fscanf(datafp, "omvang: %d %d", na, nb) <= 0) {
```

```
      return (-1);
   } else {
      return (0);
   }
}


static int lees_vector (float vector[],
                        int n,
                        struct _iobuf *datafp)
{
   int i,
       resultaat;

   for (i = 0;
        i < n
         && (resultaat = fscanf(datafp, " waarde: %f", &(vector[i]) )) > 0;
        i++)
      ;

   if (i != n || resultaat < 0) {
      return (-1);
   } else {
      return (0);
   }
}
```

Literatuur

[America] America, P.H.M.: *Issues in the design of a parallel object-oriented language*, ESPRIT project 415 document 452, Philips Research Laboratories, Eindhoven, maart 1989.

[Bal] Bal, H.E.: *The shared data-object model as a paradigm for programming distributed systems*, Centrale Huisdrukkerij Vrije Universiteit Amsterdam (1989). (proefschrift)

[Birman] Birman, K.P., T.A. Joseph: *Exploiting replication in distributed systems*, In: S.J. Mullender (Ed.): *Distributed systems*, p. 319–368, New York, New York: ACM Press/Addison-Wesley (1989).

[Birrell] Birrell, A.D., B.J. Nelson: *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, volume 2, p. 39–59, februari 1984.

[De Bruin] Bruin, A. de: *Inleiding operating systems*, Aantekeningen bij het college Telematica, eerste semester 1987/88, Erasmus Universiteit Rotterdam. (syllabus)

[Chin] Chin-A-Lien, K., L. Hoekstra, J. Laanstra, A. Nales, A. Spruit: *Bankapplicatie: credit en debet van/met RPC*, Werkcollege Netwerken (1990/1991) Erasmus Universiteit Rotterdam. (werkcollege rapport)

[Coulouris] Coulouris, G.F., J. Dollimore: *Distributed systems: Concepts and design*, Wokingham, England: Addison-Wesley (1988).

[Date] Date, C.J.: *Database systems volume I*, fourth edition, Reading, Massachusetts: Addison-Wesley (1986).

[Davidson] Davidson, S.B.: *Replicated data and partition failures*, In: S.J. Mullender (Ed.): *Distributed systems*, p. 265–292, New York, New York: ACM Press/Addison-Wesley (1989).

[DeMarco] DeMarco, T.: *Structured analysis and system specification*, Englewood Cliffs, New Jersey: Yourdon Press/Prentice-Hall (1979).

[Fortier] Fortier, P.J.: *Design of distributed operating systems*, New York, New York: McGraw-Hill (1988).

[Hoare] Hoare, C.A.R.: *The Emperor's old clothes*, Communications of the ACM, volume 24, no. 2, p. 75–83, februari 1981.

[Johnsonbaugh] Johnsonbaugh, R.: *Discrete mathematics*, New York, New York: Macmillan (1984).

[Van Katwijk] Katwijk, J. van: *Inleiding software engineering*, a243A cursusjaar 1991–1992, Faculteit Technische Wiskunde en Informatica, TU Delft (juni 1991). (syllabus)

[Kernighan 1] Kernighan, B.W., R. Pike: *The UNIX programming environment*, Englewood Cliffs, New Jersey: Prentice-Hall (1984/1985).

[Kernighan 2] Kernighan, B.W., D.M. Ritchie: *The C programming language*, second edition, Englewood Cliffs, New Jersey: Prentice-Hall (1988).

[Laffra] Laffra, J.C.: *Procol—A concurrent object language with protocols, delegation, persistence, and constraints*, Erasmus Universiteit Rotterdam (1992). (proefschrift)

[Leipoldt] Leipoldt, M.F.J.: *Het gebruik van metaforen in het informatica-onderwijs*, NIOC, Maastricht: NGI 1990.

[Liskov] Liskov, B., R. Scheifler: *Guardians and actions: Linguistic support for robust, distributed programs*, ACM Transactions on Programming languages and systems, volume 5, no. 3, p. 381–404, juli 1983.

[Matthijsen] Matthijssen, R.L., J.H.J.M. Truijens: *Computers, datacommunicatie en netwerken*, Schoonhoven: Academic Service (1987).

[Mullender 1] Mullender, S.J.: *Introduction*, In: S.J. Mullender (Ed.): *Distributed systems*, p. 3–18, New York, New York: ACM Press/Addison-Wesley (1989).

[Mullender 2] Mullender, S.J.: *Protection*, In: S.J. Mullender (Ed.): *Distributed systems*, p. 117–132, New York, New York: ACM Press/Addison-Wesley (1989).

[Page-Jones] Page-Jones, M.: *The practical guide to structured systems design*, Englewood Cliffs, New Jersey: Prentice-Hall (1980).

[Van Renesse] Renesse, R. van: *The functional processing model*, Centrale Huisdrukkerij Vrije Universiteit Amsterdam (1989). (proefschrift)

[Rochkind] Rochkind, M.J.: *Advanced UNIX programming*, Englewood Cliffs, New Jersey: Prentice-Hall (1985).

[Van der Sluis] Sluis, A. van der, C.A.C. Görts: *Cursus Pascal*, zesde druk, Schoonhoven: Academic Service (1986).

[Spector] Spector, A.Z.: *Distributed transaction processing facilities*, In: S.J. Mullender (Ed.): *Distributed systems*, p. 191–215, New York, New York: ACM Press/Addison-Wesley (1989).

[Sun 1] Sun Microsystems: *Network programming manual*, revision A, Sun Microsystems (1990). (manual)

[Sun 2] SunOS/UNIX manual pages: *rpc_clnt_calls, yacc*, Sun Microsystems (1990). (manual)

[Tanenbaum 1] Tanenbaum, A.S.: *Structured computer organization*, second edition, Englewood Cliffs, New Jersey: Prentice-Hall (1984).

[Tanenbaum 2] Tanenbaum, A.S.: *Operating systems: Design and implementation*, Englewood Cliffs, New Jersey: Prentice-Hall (1987).

[Tanenbaum 3] Tanenbaum, A.S.: *Computer networks*, second edition, Englewood Cliffs, New Jersey: Prentice-Hall (1988).

[Tanenbaum 4] Tanenbaum, A.S., R. van Renesse: *A critique of the remote procedure call paradigm*, In: R. Speth (Ed.): EUTECO '88 *Research into networks and distributed applications*, p. 775–783, Amsterdam: North-Holland/Elsevier science publ. (1988).

[Tanenbaum 5] Tanenbaum, A.S.: *Modern operating systems*, Englewood Cliffs, New Jersey: Prentice-Hall (1992).

[Van 't Veld] Veld, S.F.N. van 't: *16 Methoden voor systeemontwikkeling*, Amsterdam: Tutein Nolthenius (1990).

[Watt 1] Watt, D.A.: *Programming language concepts and paradigms*, Hemel Hempstead: Prentice-Hall (1990).

[Watt 2] Watt, D.A.: *Programming language syntax and semantics*, Hemel Hempstead: Prentice-Hall (1991).

[Weihl 1] Weihl, W.E.: *Remote Procedure Call*, In: S.J. Mullender (Ed.): *Distributed systems*, p. 65–86 New York, New York: ACM Press/Addison-Wesley (1989).

[Weihl 2] Weihl, W.E.: *Theory of nested transactions*, In: S.J. Mullender (Ed.): *Distributed systems*, p. 237–262 New York, New York: ACM Press/Addison-Wesley (1989).