CALLEBAUT Gilles

GEDISTRIBUEERDE SYSTEMEN

2015 - 2016

Inhoudsopgave

Lijst van figuren

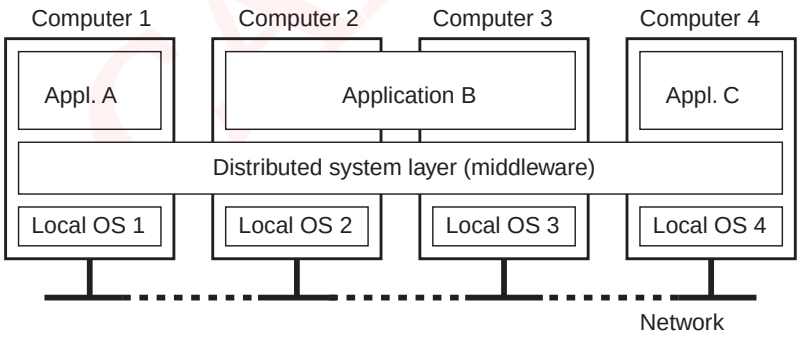
Hoofdstuk 1

Introductie

Inhoudsopgave

1.1 Definition of distributed system

Gedistribueerde systemen zijn verzamelingen van onafhankelijke computers waarbij het voor de gebruikers lijkt dat het gaat over één coherent systeem.



1.2 Goals

In deze sectie worden vier belangrijke doelen besproken. Deze doelen moeten gehaald worden om het bouwen van een gedistribueerd systeem de moeite waard te maken.

- Resources moeten gemakkelijk toegankelijk gemaakt worden
- Distributie van resources moet transparant gebeuren
- Het moet open zijn
- Het moet schaalbaar zijn

1.2.1 Making Resources Accessible

Resources zoals printers, computers, data, bestanden moeten worden gedeeld. Dit kan zijn om *economische* en *collaboratie* redenen. Kostelijke apparaten, e.g. supercomputers, worden beter gedeeld om de kost te drukken. Met dank aan het internet is het mogelijk om met een groep te communiceren die wijd verspreid is, denk maar aan Skype.

Er worden wel nieuwe problemen waargenomen die gepaard gaan met de verhoogde connectiviteit, e.g. veiligheid, privacy, ongewenste communicatie en performantie-kosten. Dit laatste slaat dan specifiek op gedistribueerde systemen.

1.2.2 Distribution Transparancy

Een belangrijk doel van gedistribueerde systemen is het verbergen dat processen en resources verspreid zijn over meerdere fysische computers. M.a.w. een gedistribueerd systeem moet transparant zijn. Hieronder zijn enkele aspecten opgesomd waaraan een transparant gedistribueerd systeem moet voldoen.

Access
: Verberg verschillen in data representatie en hoe deze toegankelijk is. Een voorbeeld is een printer, de gebruiker hoeft niet te weten of het gaat over een netwerk-printer of een printer die geconnecteerd is via usb.

Location
: Verberg waar de resource is opgeslagen. Wanneer een gebruiker een URL intypt hoeft deze niet te weten op welke fysische machine deze resource zich bevindt.

Migration
: Verberg dat de resource zich eventueel kan verplaatsen.

Relocation Verberg dat de resource wordt verplaatst wanneer hij in gebruik is.

Replication
: Verberg dat een resource is gedupliceerd.

Concurrency
: Verberg dat een resource kan worden gedeeld met andere competitieve gebruikers.

Failure

Verberg het falen en herstellen van een resource. Als er een server uitvalt van `www.google.be`, moet deze toegankelijk blijven.

Degree of Transparency

Alhoewel volledige transparantie preferabel is voor elk gedistribueerd systeem is dit niet mogelijk of niet wenselijk. Soms is het namelijk belangrijk om de locatie van resources wel toegankelijk te maken; de locatie van een printer is zo'n voorbeeld.

Er is ook een *trade-off* tussen de *transparantie-graad* en de *performantie* van een systeem. Bijvoorbeeld, we willen replica's op verschillende continenten consistent houden. Updates kunnen hierdoor niet transparant blijven voor de gebruikers. Er moet ook nog een trade-off gemaakt worden met de *comprehensibility* (verstaanbaarheid) zodat gebruikers en applicatie ontwikkelaar nooit worden misleid door te geloven dat er transparantie is. De gebruikers zullen hierdoor het gedrag van een gedistribueerd systeem beter begrijpen en er dus beter mee kunnen omgaan.

1.2.3 Openness

Een open gedistribueerd systeem biedt diensten aan volgens standaard regels die de *syntax* (structuur) en *semantiek* (betekenis) van de diensten beschrijven.

De syntax van diensten wordt beschreven in een Interface Definition Language (IDL)[1]. Dit zijn de namen van de functies, paramtertypes, return waarden, ... Wat deze diensten doen, i.e. de semantiek, worden beschreven door de natuurlijke taal.

Specificaties moeten compleet en neutraal zijn. Met *compleet* wordt bedoeld dat men een implementatie kan maken met de specificatie. Interfaces zijn vaak niet compleet, de ontwikkelaar moet dan implementatie-specifieke details toevoegen. De specificaties mogen niet voorschrijven hoe een implementatie er moet uit zien: het moet *neutraal* zijn.

Volledigheid en neutraliteit zijn belangrijk voor interoperability en portability.

Interoperability

Interoperability karakteriseert de mate waarin twee implementaties van verschillende makers naast elkaar kunnen bestaan en samenwerken waarbij ze enkel steunen op elkaars diensten die gespecificeerd staan door een gemeenschappelijke standaard.

Portability

Portability karakteriseert de mate waarin een applicatie ontwikkeld voor een systeem A kan worden uitgevoerd, zonder modificatie, door een ander gedistribueerd systeem B met dezelfde interface (IDL) als A.

[1] Een IDL beschrijft **meestal** enkel de syntax van een dienst.

Extensibility

Het systeem moet ook gemakkelijk worden geconfigureerd uit verschillende componenten. Deze componenten moeten ook eenvoudig kunnen worden vervangen, of nieuwe componenten moeten kunnen toegevoegd worden.

Separting Policy from Mechanism

Het komt de flexibiliteit van een gedistribueerd systeem ten goede wanneer dit is opgebouwd uit kleine en gemakkelijk vervangbare componenten. Er zijn dus ook interfaces nodig voor het interne gedeelte.

In het geval van webcaching moet een browser documenten kunnen opslaan en de gebruikers de mogelijkheid geven aan de gebruikers om te beslissen welke document en hoelang deze moeten worden opgeslagen. Er kan een cache strategie worden ontworpen in de vorm van een component, deze kan dan worden ge-plugged in de browser.

1.2.4 Scalability

De schaalbaarheid van een systeem kan worden opgemeten door drie verschillende dimensies.

1. **Size**
   Meer gebruikers en resources moeten eenvoudig kunnen toegevoegd worden.
2. **Location**
   Een geografisch schaalbaar systeem bevat gebruikers en resources die ver van elkaar liggen.
3. **Administration**
   Het systeem moet gemakkelijk beheerbaar zijn ook als het verschillende onafhankelijke administratieve organisaties omvat.

Een systeem dat voldoet aan één van de schaalbaarheid eisen ondervindt veelal performantie verlies.

Scalability Problems

Size Scalability Als er meer gebruikers en resources ondersteunt moeten worden, botsen we vaak met de limitaties van gecentraliseerde diensten, data en algoritmen.

Enkel *gedecentraliseerde algoritmen* zouden moeten worden gebruikt. Deze hebben de volgende eigenschappen:

- Geen één machine heeft complete informatie over de staat van het systeem
- Machines maken enkel keuzes op basis van lokale informatie
- Het falen van één machine brengt het algoritme niet in het gedrang
- Er is geen impliciete assumptie dat een globale klok bestaat.

Geographical Scalability In een wide-area systeem zal de IPC (InterProcess Communication) trager gaan dan in een LAN. Er ontstaan *communicatie vertragingen*, die kunnen worden opgevangen door van synchrone naar asynchrone communicatie over te schakelen.

Een ander probleem dat geografische schaalbaarheid hindert, is dat communicatie in WANs onbetrouwbaar en 'altijd' point-to-point zijn. In LANs is het mogelijk om een broadcast te sturen om te vragen of er een machine een dienst kan verlenen. Hierdoor moeten er speciale locatie services ontworpen worden die schaalbaar zijn, zie hoofdstuk 5.

Amdinistrative Scalability Een groot probleem zijn de conflicterende policies gerelateerd aan betalingen, veiligheid en beheer. Nieuwe domeinen zouden bijvoorbeeld kunnen worden afgeschermd van een ander domein. Het nieuwe domein moet zichzelf ookk beschermen tegen aanvallen van het gedistribueerde systeem –het zelfde geldt ook voor het omgekeerde geval.

Scaling Techniques

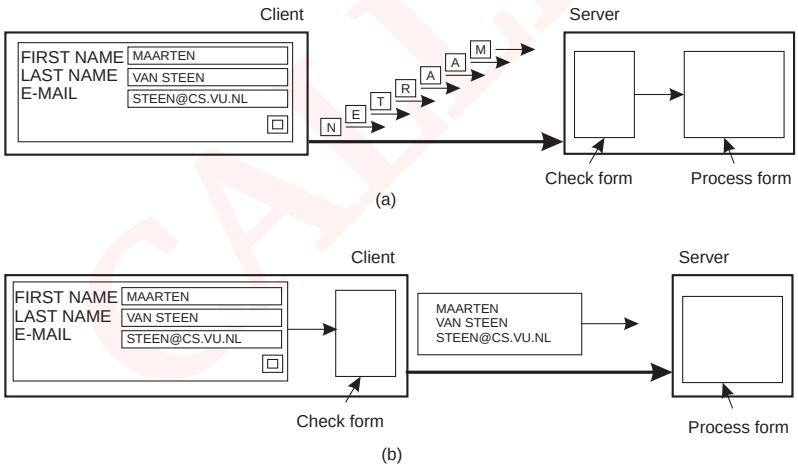
Schaalbaarheid problemen worden veroorzaakt door gelimiteerde capaciteit van servers en netwerken, en resulteren in performantie problemen.

Er bestaan drie technieken om schaalbaarheidsproblemen aan te pakken, i.e. verbergen van de communicatie-traagheid, distributie en replicatie.

Hiding communication latencies Het verbergen van de communicatie-traagheid is belangrijk voor de geografische schaalbaarheid. Het basis idee is als volgt, wacht zo min mogelijk op antwoorden van afgelegen diensten. Dit kan worden verkregen via twee technieken:

1. Van synchrone naar asynchrone communicatie
   Start een interrupt handler om de vorige request af te handelen.

2. Verplaats de berekeningen naar de gebruiker
   Dit kan worden verwezenlijkt door JavaScript en Java Applets.



Distribution Distributie bestaat uit het verdelen van een component in kleinere delen, die dan worden verdeeld over het systeem, e.g. DNS.

Replication Vermits schaalbaarheid vaak leidt tot performantie degradatie zouden componenten moeten worden gerepliceerd. Dit leidt tot

- Een verhoogde vorm van *availibility*
- Verhoogde performantie door de verspreiding van de belasting

Caching is een speciale vorm van replicatie. Bij caching zal de keuze tot replicatie liggen bij de gebruiker van de resource, in tegenstelling tot replicatie waarbij de keuze ligt bij de eigenaar van de resource. Caching zal overigens gebeuren *on demand* waarbij replicatie op voorhand gepland is.

Vermits we nu zitten met meerdere kopieën worden we geconfronteerd met consistentie problemen. Modificaties moeten worden gepropageerd naar alle andere kopieën. Er is nood aan een *globale synchronisatie mechanisme*, wat extreem moeilijk tot onmogelijk is om te implementeren.

1.2.5 Pitfalls

- The network is reliable.
- The network is secure.
- The network is homogeneous.
- The topology does not change.
- Latency is zero.
- Bandwidth is infinite.
- Transport cost is zero.
- There is one administrator.

1.3 Types of distributed systems

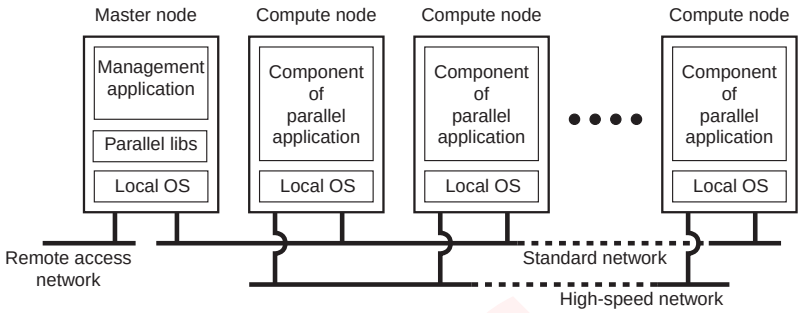
Er kan een onderscheid worden gemaakt tussen distributed *computing* systems, distributed *information* systems en distributed *embedded* systems.

1.3.1 Distributed Computing Systems

Gedistribueerde computing systemen worden gebruikt om hoge-performantie berekeningen te kunnen uitvoeren. gedistribueerde computing systemen kunnen worden opgedeeld in twee subgroepen. De eerste opdeling is deze waarbij er gebruik wordt gemaakt van een cluster van gelijkaardige computers, die verbonden zijn met een snelle LAN. Dit zijn de cluster computing systemen. In het grid systeem, de tweede opdeling, worden PC's opgebouwd als een federatie van computer systemen.

Cluster Computing Systems

Cluster computing systemen worden gebruikt om een parallel geprogrammeerd reken-intensief programma in parallel uit te voeren op verschillende machines. Deze machines hebben dezelfde karakteristieken, i.e. draaien hetzelfde OS en voeren eenzelfde taak uit binnen een netwerk.

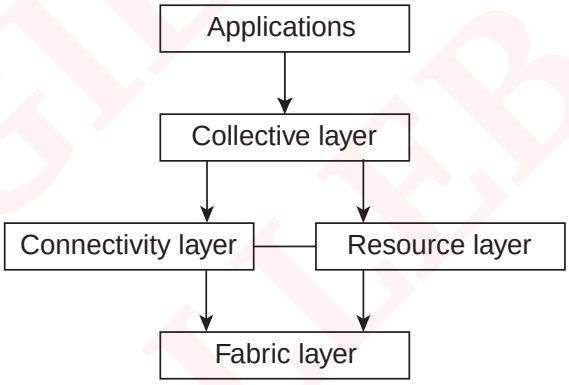


Figuur 1.1: Cluster Computing System - Homogeen

Grid Computing System

Grid computing systemen hebben een hoge graad van heterogeniteit, geen assumpties over hardware, besturingssystemen, netwerken, administratieve domeinen, beveiligings-policies, etc.

Er wordt een virtuele organisatie gecreëerd om alle taken te volbrengen, wat leidt tot een service georiënteerde architectuur waarbij elke machine zijn eigen taak heeft.



Figuur 1.2: Grid Computing System - Heterogeen

De architectuur, weergegeven in figuur 1.2, bestaat uit vier lagen. De middleware is de combinatie van de collective, connectivity en resource lagen.

1.3.2 Distributed Information Systems

De distributed information systems werden gebruikt bij organisaties die geconfronteerd waren met een overvloed aan netwerk-applicaties waarbij interoperability een pijnlijke gebeurtenis was.

Bijvoorbeeld, een gebruiker wil een reis boeken (vliegtuig, hotel en huurwagen). Dit is een *composit request* waarbij verschillende diensten moeten worden geleverd, met het idee '*we boeken alles of niets*'.

Transaction Processing Systems

RPCs (*Remote Procedure calls*) worden vaak ge-encapsuleerd in een transactie. `BEGIN_TRANSACTION` en `END_TRANSACTION` worden gebruikt om de scope van de transactie af te bakenen. Waarbij in de body van een transactie alle operaties worden uitgevoerd of geen.

Transacties worden gekarakteriseerd door volgende eigenschappen (ACID):

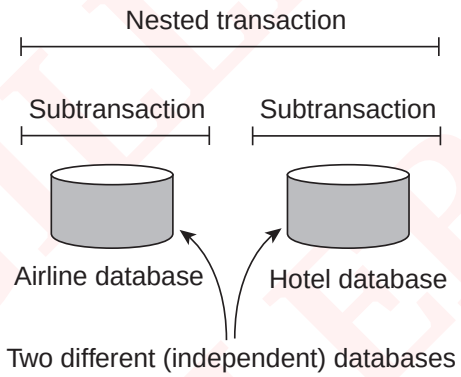
Atomic
De transactie gebeurt ondeelbaar en in een onmiddellijke actie.

Consistent
De transacties bevat systeem invarianten, i.e. eigenschappen die onveranderlijk blijven nadat er een transformatie is doorgevoerd.

Isolated
Parallele transacties interfereren niet met elkaar.

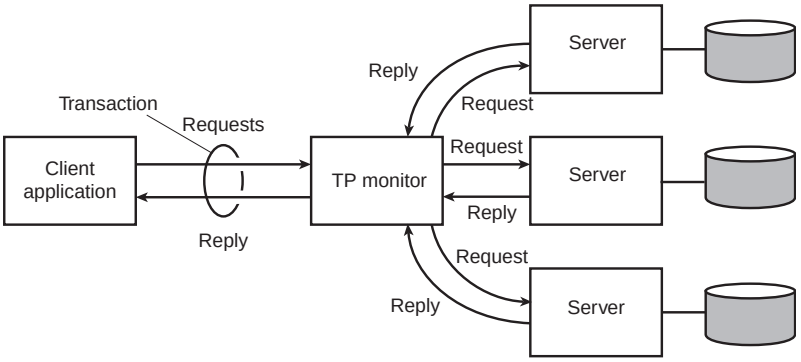
Durable
Eens een transactie is doorgevoerd, blijven de veranderingen permanent.

Een *nested transaction* bestaat uit een aantal subtransacties, zoals weergegeven op figuur 1.3.



Figuur 1.3: Een geneste transactie

Transaction Processing Monitor De transaction processing monitor (TP monitor) behandelt de gedistribueerde (of geneste) transacties om applicaties op server of databank niveau te integreren. Dit is weergegeven in figuur 1.4 op de volgende pagina. De hoofdtaak was om applicaties toegang te bieden voor meerdere servers/databanken –door een transactional programming model aan te bieden.

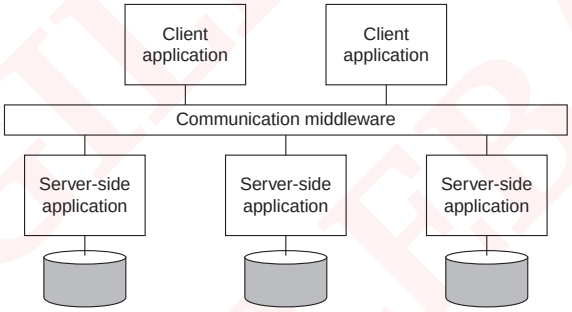


Figuur 1.4: De rol van een TP monitor in gedistribueerde systemen

Enterprise Application Integration

Vermits meer applicaties zich begonnen te ontkoppelen van de databanken waarop ze werden gebouwd, was het evident om diensten te bieden die applicaties integreerden los van hun databanken, i.e. EAI (*Enterprise Application Integration*). Het hoofdidee bestond erin om bestaande applicaties de mogelijkheid te bieden om direct informatie met elkaar te delen, dit staat weergegeven op figuur 1.5.

Hierbij kan één applicatie zich voordoen als de TP monitor. Een voorwaarde is dat de applicaties hun interface moeten blootstellen.



Figuur 1.5: Middleware als communicatie middel in EAI

Verschillende types van commmunicatie middleware bestaan:

Remote Procedure Call (RPC)
: Een applicatie component kan een request verzenden naar een remote applicatie –door een lokale *procedure oproep*. Het resultaat wordt dan terug bezorgd als het resultaat van de procedure call.

RMI
: RMI is een speciaal geval van RPC, het functioneert op objecten in plaats van applicaties.

Message-Oriented Middleware (MOM)
: MOM lost het probleem op van RMI en RPC, i.e. waarbij beide systemen *up-and-running* moeten zijn en waarbij er moet geweten zijn hoe ze naar elkaar moeten refereren. Om de *strakke koppeling* te vermijden wordt er gebruik gemaakt van het communiceren via berichten. De systemen worden opgebouwd rond het *publish/subscribe principe*.

1.3.3 Distributed Pervasive Systems

Gedistribueerde pervasive systemen of ook wel embedded systemen genoemd, zijn vaak klein, aangedreven door batterijen, mobiel en beschikken enkel over draadloze connecties. Deze systemen zijn onderdeel van de omgeving.

Pervasive applicaties moeten voldoen aan deze drie voorwaarden:

1. Omarm contextuele veranderingen
   Het apparaat moet op de hoogte zijn van zijn veranderende omgeving.

2. Aanmoedigen van ad-hoc compositie
   De aanmoediging van ad-hoc compositie refereert naar het feit dat veel apparaten in pervasive systemen op verschillende manieren zullen worden gebruikt door gebruikers. Apparaten moeten dan ook gemakkelijke en applicatie-afhankelijke adaptatie ondersteunen. Bijvoorbeeld, er moeten eenvoudig nieuwe sensoren toegevoegd of verwijderd worden aan een systeem.

3. Herken delen als default
   Distributie transparantie is niet aangewezen. De distributie van data, processen en controle is verbonden aan deze systemen, daarom is het beter het openbaar te maken dan te verbergen.

Een voorbeeld van een pervasive systeem zijn home systems, systemen gebouwd rond home networks zoals computers, TVs, smartphones, ... Zulke systemen moeten volledig zelf-configurerend en zelf-beherend zijn. Sharing restricties moeten ook aanwezig zijn om de “personal space” te respecteren.

1.4 Summary

Distributed systems consist of autonomous computers that work together to give the appearance of a single coherent system. One important advantage is that they make it easier to integrate different applications running on different computers into a single system. Another advantage is that when properly designed, distributed systems scale well with respect to the size of the underlying network. These advantages often come at the cost of more complex software, degradation of performance, and also often weaker security. Nevertheless, there is considerable interest worldwide in building and installing distributed systems.

Distributed systems often aim at hiding many of the intricacies related to the distribution of processes, data, and control. However, this distribution transparency not only comes at a performance price, but in practical situations it can never be fully achieved. The fact that trade-offs need to be made between achieving various forms of distribution transparency is inherent to the design of distributed systems, and can easily complicate their understanding. -

Matters are further complicated by the fact that many developers initially make assumptions about the underlying network that are fundamentally wrong. Later, when assumptions are dropped, it may turn out to be difficult to mask unwanted behavior. A typical example is assuming that network latency is not significant. Later, when porting an existing system to a wide-area network, hiding latencies may deeply affect the system’s original design. Other pitfalls include assuming that the network is reliable, static, secure, and homogeneous.

Different types of distributed systems exist which can be classified as being oriented toward supporting computations, information processing, and pervasiveness. Distributed computing systems are typically deployed for high-performance applications often originating from the field of parallel computing. A huge class of distributed can be found in traditional office environments where we see databases playing an important role. Typically, transaction processing systems are deployed in these environments. Finally, an emerging class of distributed systems is where components are small and the system is composed in an ad hoc fashion, but most of all is no longer managed through a system administrator. This last class is typically represented by ubiquitous computing environments.

Hoofdstuk 2

Architecturen

De software architecturen vertellen ons hoe software componenten georganiseerd zijn en hoe ze moeten omgaan met elkaar.

In dit hoofdstuk bekijken we de traditionele gecentraliseerde architecturen waarbij één server de meeste software componenten implementeert, waarbij remote clients de server kunnen contacteren via simpele communicatie. Dit in tegenstelling tot gedecentraliseerde architecturen en hybride organisaties waarbij machines een gelijkaardige rol vervullen.

Inhoudsopgave

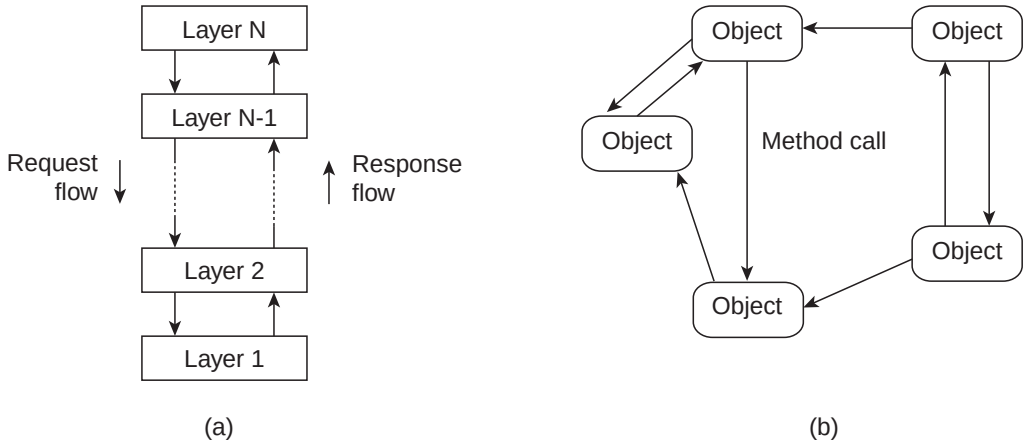
2.1 Architectural Styles

Via componenten en connectoren kunnen er verschillende configuraties worden gerealiseerd. De belangrijkste stijlen zijn:

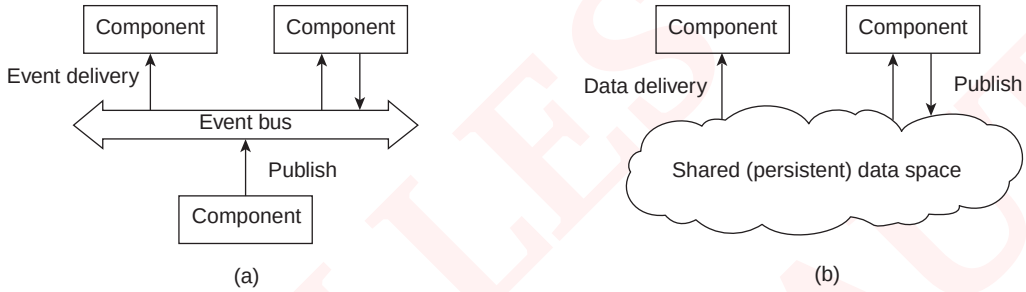
- Layered architectures
- Object-based architectures
- Data-centered architectures
- Event-based architectures

Layered architectures

Componenten worden georganiseerd op basis van lagen, waarbij bovenliggende lagen de (vlak) onderliggende laag kunnen oproepen –maar niet omgekeerd. Beperkingen hierbij zijn dat enkel



Figuur 2.1: De (a) laag en (b) object-gebaseerde architectuur stijl



Figuur 2.2: De (a) event-gebaseerde en (b) gedeelde data-ruimte architectuur stijl

aangrenzende lagen met elkaar kunnen communiceren. Deze architectuur stijl is weergegeven op figuur (a) 2.1.

object-based architectures

Een lossere organisatie, die is geïllustreerd op figuur (b) 2.1, is die van een object-gebaseerde architectuur stijl. Objecten (componenten) communiceren via elkaar via (R)PC. Dit mechanisme werkt op basis van een *client/server model*.

Data-centered architectures

Processen publiceren events waarna de middleware garandeert dat enkel de processen die geabonneerd zijn op die gebeurtenissen deze zullen ontvangen, dit is het *publish/subscribe systeem*. Processen zijn hierdoor los gekoppeld, i.e. referential decoupling (decoupled in space), de referenties moeten niet gekend zijn van de componenten die de taak uitvoert. Een service kan door verschillende componenten worden afgehandeld. Zo'n stijl is weergegeven op (a) 2.2.

Event-based architectures

Wanneer event-based en data-centered architecturen gecombineerd worden krijgen we *shared data spaces*, wat weergegeven is op (b) 2.2. Processen zijn nu ook ontkoppeld in de tijd (*decoupled*

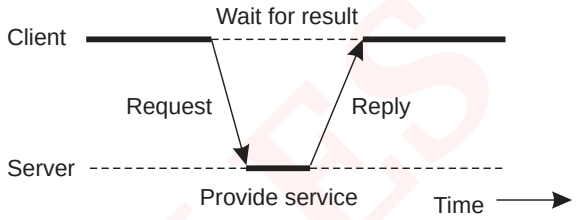
in time), i.e. beiden moeten niet actief zijn op het moment van de communicatie. Taken worden gedropt in een *shared repository* waarbij componenten deze taken uitvoeren wanneer zij actief worden en de mogelijkheid hebben om deze uit te voeren.

2.2 System Architectures

2.2.1 Centralized Archtectures

In een LAN kan er gebruikt worden van connectionless protocollen, als het netwerk betrouwbaar is en als de requests idempotent[1] zijn.

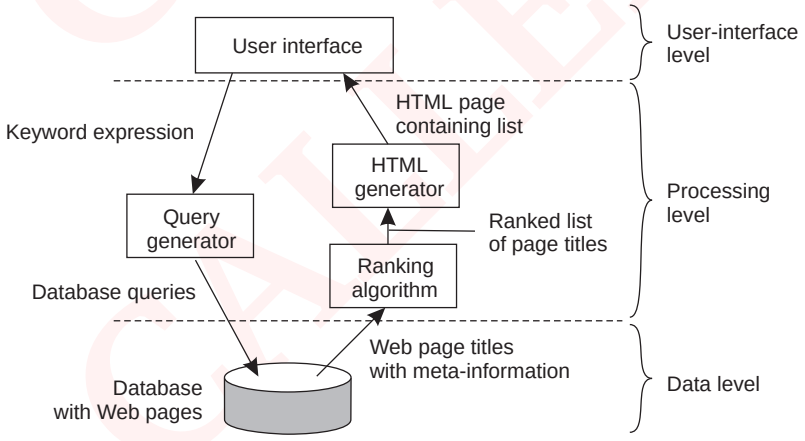
Als men echter overgaat naar WAN volstaat de connectionless protocollen niet meer en moet men overgaan naar connection-oriented protocollen, e.g. TCP/IP.



Figuur 2.3: Algemene interactie tussen client en server

Application Layering

Veel client-server applicaties kunnen worden opgedeeld in drie verschillende onderdelen: een gedeelte die de gebruikersinteractie afhandelt, een deel die de databank of bestandssysteem bedienen en een middel gedeelte die de core functionaliteit van een applicatie bevat.



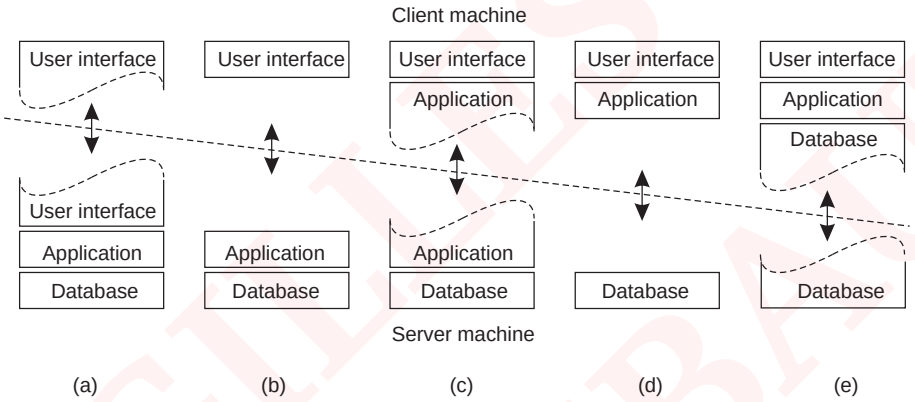
Figuur 2.4: Een vereenvoudigde organisatie van een internet zoekmachine in drie verschillende lagen.

[1] Wanneer het antwoord verloren raakt dan kan het herverzenden van de request ervoor zorgen dat de operatie tweemaal wordt uitgevoerd. Als dit niet resulteert in een verkeerde situatie dan is de operatie idempotent.

Multitiered Architectures Er bestaan verschillende mogelijkheden om deze lagen te verdelen over enkele machines. De eenvoudigste organisatie is degene waarbij de client en server worden opgedeeld op twee verschillende machines. De client bevat dan enkel de user-interface level. Een server machine zal het processing en data gedeelte bevatten. Dit noemt men een *two tier* architectuur.

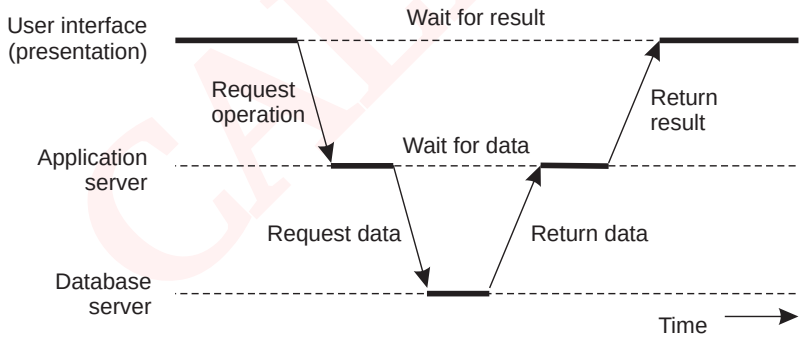
Als men het aantal machines verhoogt –waarop de applicatie is verdeeld– zal de processing power toenemen en zal er een lossere koppeling mogelijk zijn, i.e. componenten kunnen apart aangesproken worden. Een nadeel hierbij is wel dat de communicatie delay ook verhoogt.

Two tier Architecture In een two tier architectuur verdelen we de lagen over een client en server machine. Figuur 2.5 geeft de mogelijke opdelingen weer. (a) Waarbij de clients worden aanzien als *thin clients* of terminals die enkel de data representeren. (b) is een voorbeeld van een website, bij (c) heeft de website mogelijkheid om client-code uit te voeren. In (d) en (e) spreekt men van *fat clients*, e.g. banking applicatie (d) en local caching (e).



Figuur 2.5: Alternatiesve client-server organisaties

Three tier Architecture Een server moet zich soms ook gedragen als client, zoals weergegeven in figuur 2.6. In deze architectuur kunnen programma's die een gedeelte vormen van de processing level op een andere machine zitten.



Figuur 2.6: Een voorbeeld waarbij een server zich gedraagt als client.

2.2.2 Decentralized Archtectures

Gecentreerde architecturen gaan logische componenten van elkaar scheiden en plaatsen op andere machines. Zo kan men bijvoorbeeld een server per laag toewijzen. Dit noemt met *verticale distributie*.

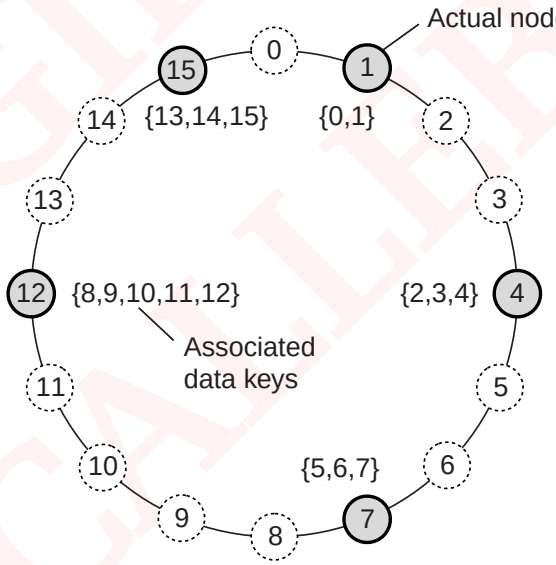
In de gecentraliseerde distributies gaat men de lagen horizontaal gaan verdelen. Peer-to-peer systemen is een voorbeeld van een horizontale distributie architectuur. Elk proces zal zich gedragen als een client en een server op het zelfde moment. Processen moeten communiceren via overlay networks, i.e. netwerken waar noden gevormd worden door processen en de linken de mogelijke communicatie wegen voorstelt. Deze kunnen op twee manieren gevormd worden: gestructureerde en ongestructureerde netwerken.

Structured peer-to-peer systems

In gestructureerde peer-to-peer systemen is het overlay netwerk via een *deterministische procedure* opgebouwd. De meets gebruikt procedure is om processen te organiseren op basis van een *gedistribueerde hash table* (DHT).

Data en nodes krijgen een identifier vanuit dezelfde identifier space. Wanneer het data item wordt gezocht moet men het netwerkadres terugkrijgen van de node verantwoordelijk voor die data. Voorbeelden hiervoor zijn het Chord systeem en het Content Addressable Network.

Chord System In het Chord systeem worden de nodes in een kring georganiseerd. Een data item met key k wordt dan ge-mapped op de actieve node met de kleinste identifier $id \sim k$.



Figuur 2.7: De mapping van data items naar nodes in het Chord systeem

Nodes kunnen zichzelf gemakkelijk organiseren, de naastliggende buren hoeven enkel gekend zijn.

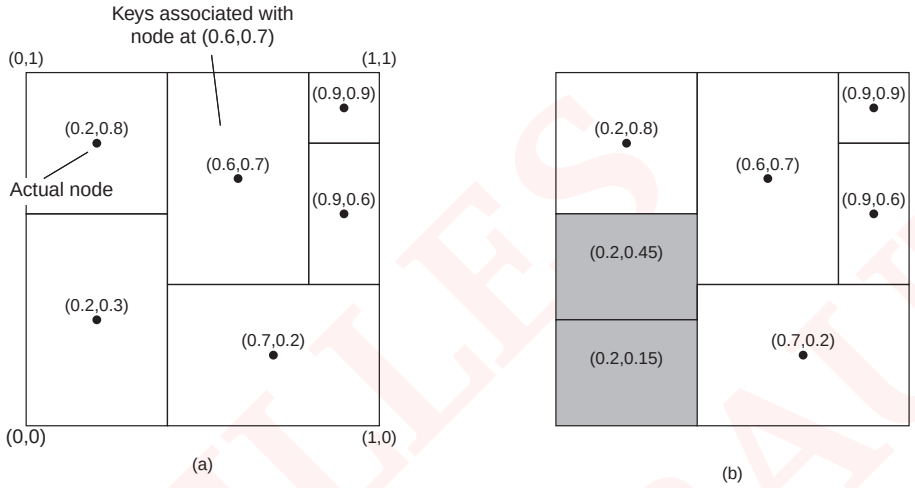
Node sluit zich aan Er wordt een random identifier id gegenereerd. De node zoekt het netwerkadres op van de data met key id. De nieuwe node contacteert de `succ(id)` en zijn voorganger en

voegt zichzelf in de kring.

Node verlaat het systeem Een node id kondigt zijn vertrek aan bij zijn voorganger en opvolger en geeft zijn data door aan `succ(id)`.

Een node hoeft dus enkel zijn twee buren te kennen. Dit zorgt voor een gemakkelijke *membership management*.

Content Addressable Network (CAN) CAN gebruikt een d-dimensionaal cartesiaans coördinaten ruimte. Deze wordt verdeeld onder de actieve nodes in het systeem.



Figuur 2.8: (a) De mapping van data items naar nodes in het CAN systeem. (b) Splitsen van gebieden wanneer een node zich aansluit.

Als voorbeeld nemen we een 2-dimensionale ruimte, zoals weergegeven in figuur (a) 2.8. De ruimte is verdeelt over zes noden, waarbij elk punt voor een id staat van een data item.

Node sluit zich aan Als een node P het CAN systeem wil betreden dan kiest hij een arbitrair punt en zoekt de node Q op die verantwoordelijk is voor die regio. De node Q splitst dan zijn gebied op in twee gebieden, zoals weergegeven in figuur (a) 2.8.

Node verlaat het systeem Het verlaten van een CAN systeem is problematischer vermits het gebied niet altijd helemaal kan worden overgenomen door een buur. Als node (0.6,0.7) het systeem wilt verlaten en het gebied bijvoorbeeld wordt overgedragen aan (0.9,0.9) dan kan het gebied niet worden samengevoegd tot een rechthoek.

Routing Wanneer we de node willen te weten komen van een data item gaat men als volgt te werk. Men wil –door naar een buur te gaan– dichter bij de coördinaten van het data item komen. Wanneer men vanuit (0.2,0.3) de node wilt kennen van het item (0.9,0.8) dan zal hij zijn bovenbuur contacteren (0.2,0.8), die dan weer op zijn buur rechts van hem gaat contacteren (0.9,0.7) om als laatste terecht te komen bij node (0.9,0.9).

Het nadeel van het CAN systeem –t.o.v. het Chord systeem– is de problematiek die gepaard gaat bij het verlaten van een node. Een voordeel echter is dat elke node meer rechtstreekse buren ter zijne beschikking heeft.

Unstructured peer-to-peer systems

Het idee bij ongestructureerde peer-to-peer netwerken is dat elke node een lijst van (gedeeltelijk random) buren bijhoudt. Deze lijst van buren wordt de *partial view* genoemd. Als men een data item nodig heeft moet men via flooding de locatie te weten zien te komen.

In het framework van Jelasty et al. (2004, 2005a) wordt er verondersteld dat noden regelmatig de entries van hun partial view uitwisselen. Elke entrie bestaat ui de node id en een age die weergeeft hoe oud de referentie naar die node is. Elke node zal dan zijn partial view updaten d.m.v. twee threads: een actie en passieve thread. De actieve node start de communicatie met een andere node.

De actieve thread selecteert een node van zijn partial view. Als de entries moeten worden *gepushed* naar de geselecteerde peer, dan wordt er een buffer gemaakt die $c/2+1$ entries bevatten. De buffer bevat $c/2$ entries uit de partial view, plus een entrie die zichzelf bevat –met age 0. De $c/2$ worden gekozen a.d.h.v. de leeftijd en een random factor, zoals te zien op listing 2.1 en 2.2.

Listing 2.1: Acties voor aan actieve thread

```
select a peer P from the current partial view;
if PUSH_MODE {
        mybuffer = [(MyAddress, 0)]; // eigen doorsturen met age 0
        permute partial view;
        move H oldest entries to the end;
        append first c/2 entries to mybuffer;
        send mybuffer to P;
} else {
        send trigger to P;
}
if PULL_MODE {
        receive P's buffer;
}
construct a new partial view from the current one and P's buffer;
increment the age of every entry in the new partial view;
```

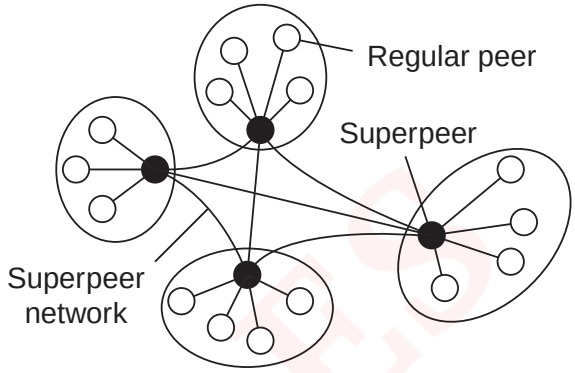
Listing 2.2: Acties voor aan passieve thread

```
receive buffer from any process Q;
if PULL_MODE {
        mybuffer = [(MyAddress, 0)];
        permute partial view;
        move H oldest entries to the end;
        append first c/2 entries to mybuffer;
        send mybuffer to P;
}
construct a new partial view from the current one and P's buffer;
```

```
increment the age of every entry in the new partial view;
```

Superpeers Het nadeel van ongestructureerde peer-to-peer systemen is dat er een broadcast vereist is om data locaties te weten te komen, wat resulteert in flooding.

Er kan gebruik worden gemaakt van superpeers, die altijd online zijn, een vaste locatie hebben, hoge data snelheid en veel rekenkracht hebben. Normale peers maken dan verbinding met best passende superpeers om een superpeer netwerk te vormen, dit is weergegeven op figuur 2.9.



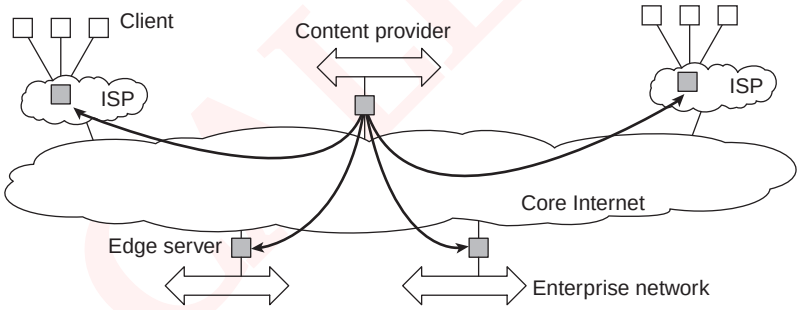
Figuur 2.9: Een hiërarchische organisatie van nodes in een superpeer netwerk.

2.2.3 Hybrid Archtectures

Hybride architecturen zijn een combinatie tussen client-server systemen met gedecentraliseerde architecturen.

Edge-Server Systems

Clients connecteren met het internet d.m.v. een edge server. Een edge server gedraagt zich als een origin server vanwaar alle content afkomstig is.

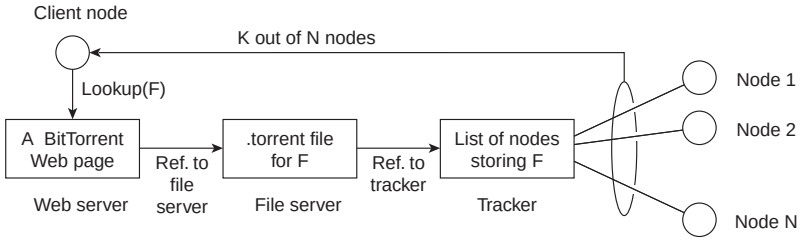


Figuur 2.10: Het internet gezien als een collectie van edge servers.

Collaborative Distributed Systems

BitTorrent is een peer-to-peer bestandsdownload systeem. Het eerste gedeelte gedraagt zich als een client-server systeem. Wanneer de nodes zijn geïdentificeerd van waar de chunks moeten

gedownload, komt men terecht in een P2P systeem.



Figuur 2.11: De principiële werking van BitTorrent.

2.3 Summary

Distributed systems can be organized in many different ways. We can make a distinction between software architecture and system architecture. The latter considers where the components that constitute a distributed system are placed across the various machines. The former is more concerned about the logical organization of the software: how do components interact, it what ways can they be structured, how can they be made independent, and so on.

A key idea when talking about architectures is architectural style. A style reflects the basic principle that is followed in organizing the interaction between the software components comprising a distributed system. Important styles include layering, object orientation, event orientation, and data-space orientation.

There are many different organizations of distributed systems. An important class is where machines are divided into clients and servers. A client sends a request to a server, who will then produce a result that is returned to the client. The client-server architecture reflects the traditional way of modularizing software in which a module calls the functions available in another module. By placing different components on different machines, we obtain a natural physical distribution of functions across a collection of machines.

Client-server architectures are often highly centralized. In decentralized architectures we often see an equal role played by the processes that constitute a distributed system, also known as peer-to-peer systems. In peer-to-peer systems, the processes are organized into an overlay network, which is a logical network in which every process has a local list of other peers that it can communicate with. The overlay network can be structured, in which case deterministic schemes can be deployed for routing messages between processes. In unstructured networks, the list of peers is more or less random, implying that search algorithms need to be deployed for locating data or other processes.

Hoofdstuk 3

Processen

Inhoudsopgave

3.1 Threads

3.1.1 Introduction to threads

In tegenstelling tot processen, zal er bij threads geen inspanningen gedaan worden om een hoge graad *concurrrency transparancy*[1] te verwezelijken. Een thread context zal echter vaak bestaan uit enkel de CPU context en informatie voor thread management.

Tread Usage in Nondistributed Systems

Voordelen van threads binnen non-gedistribueerde systemen:

[1] Elke keer een proces wordt opgestart zal het besturingssysteem een volledig onafhankele adresruimte creëren, het besturingssysteem zal ook de registers van de memory management unit (MMU) moeten aanpassen en address translation caches zoals bij de TLB moeten invalideren. Processen kunnen ook worden uit-geswapped worden. De concurrency transparancy komt dus met een relatief hoge prijs.

- Wanneer een single-threaded proces een blocked system call uitvoert, zal heel het proces worden geblocked. Dit in tegenstelling tot wanneer er gebruik gemaakt wordt van verschillende threads.
- Via threads is het mogelijk om gebruik te maken van parallellisme op een multiprocessor systeem.
- Er is minder process communication overhead door gebruik van threads. Gedeeld geheugen vraagt minder overhead dan IPC[2].
- Soms is het gemakkelijker om een programma te zien als een collectie van samenwerkende threads.

Thread Implementation

Threads kunnen zichtbaar gemaakt worden voor de kernel, of kunnen enkel in de user space onderhouden worden.

User Level Threads Hier is het gemakkelijk om threads te creëren en te vernietigen. Er is weinig overhead gepaard bij het aanmaken van threads.

Een tweede voordeel is dat thread switching goedkoop is. Enkel de waarden van de CPU registers moeten worden opgeslagen of worden geladen.

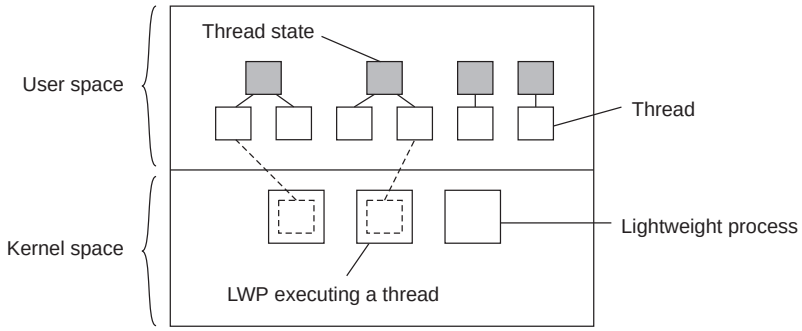
Een groot nadeel echter is als een user-level thread wordt geblocked, dan zal het gehele proces worden geblocked – bij blocking system calls.

Kernel Level Threads In het geval van kernel-level threads zullen de operaties (creëren, verwijderen, synchronisatie,...) moeten worden uitgevoerd door de kernel. Deze vergt een system call die gepaard gaat met meer context switching. Thread switching wordt zo even duur als process switching.

Een voordeel is wel dat niet heel het proces wordt geblocked, i.e. non-blocking system call.

Lightweight processes (LWP) De oplossing bevindt zich bij een combinatie van user en level threads, een hybride vorm. Hierbij worden alle operaties in user space uitgevoerd. De LWPs worden uitgevoerd in de context van een proces. De LWPs worden uitgevoerd in hun eigen user-level threads, waarbij een thread kan worden toegewezen aan een LWP.

[2] IPC vraagt tussenkomst van de kernel. Eerst moet er een switch gemaakt worden van user naar kernel space (binnen S1), dan moet er een context switch gemaakt worden (tussen S1 en S2), om dan te eindigen met een kernel naar user space switch (binnen S2).



Figuur 3.1: Combinatie van kernel-level lightweight processes en user-leve threads

Een LWP wordt aangemaakt via een system call, die dan de scheduling routine moet uitvoeren op zoek naar een een thread om uit te voeren. Deze LWPs maken en beheren dan de user-level threads.

3.1.2 Threads in Distributed Systems

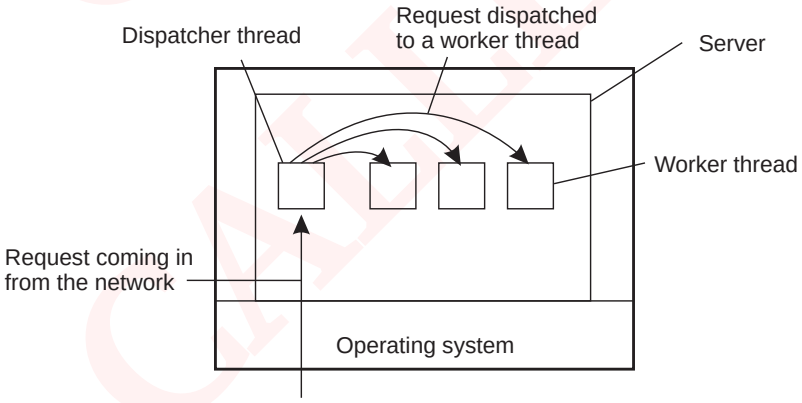
Multithreaded Clients

Bij clients kunnen communication latencies worden vebrorgen door te werken met threads. Als we het voorbeeld nemen van een webbrowser, kan een webbrowser enerzijds verschillende parallele connecties aangaan voor het ophalen van data. Anderzijds kunnen deze connecties ook verspreidt geraken over meerdere replica's, waardoor de data nog sneller kan worden opgehaald.

Men kan ook een aparte thread aanmaken voor het weergeven van inkomende data.

Multithreaded Servers

Een voorbeeld van een model is het dispatcher/worker model, dit is weergegeven in figuur 3.2.



Figuur 3.2: Een multithreaded server georganiseerd in en dispatcher/worker model

Hier gaat de dispatcher inkomende requests ontvangen en een idle worker thread kiezen om de request af te handelen.

Vermits men per sessie een thread zal aanmake, wordt er geen informatie gelekt naar andere clients/sessies, wat de veiligheid ten goede komt.

Alternatieven Bovenop multithreading kunnen er alternatieven worden gekozen. De server kan een signle-threaded proces zijn, waardoor er geen parallellisme mogelijk is door de blocking system calls. Wat leidt tot een slechte performantie.

Een andere mogelijkheid is een finite-state machine (ook een single-threaded proces) die meerdere request kan afhandelen, maar die een grotere complexieit meebrengt voor de programmeurs. De staten van alle requests moeten worden bijgehouden. Hierdoor is er we parallellisme mogelijk door non-blocking system calls.

3.2 Virtualization

Via *multi-threading* wordt er gedaan alsof we beschikken over meerdere CPUs, i.e. virtuele CPUs. Hetzelfde kan worden doorgetrokken naar het voordoen van meerdere resources, wat *resource virtualization* wordt genoemd.

3.2.1 The Role of Virtualization in Distributed Systems

Virtualisatie kan helpen om legacy software[3] te draaien om nieuwe platformen.

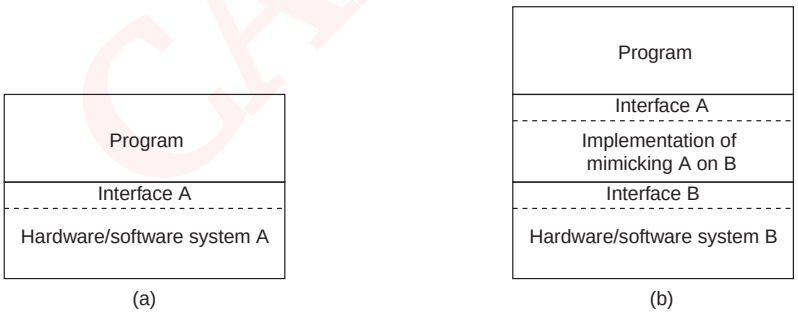
Als applicaties beschikken over hun eigen virtual machine moeten er minder platformen en machines beschikbaar zijn. Deze virtual machine zal ook de flexibiliteit en draagbaarheid van applicaties ten goede komen.

3.2.2 Architectures of Virtual Machines

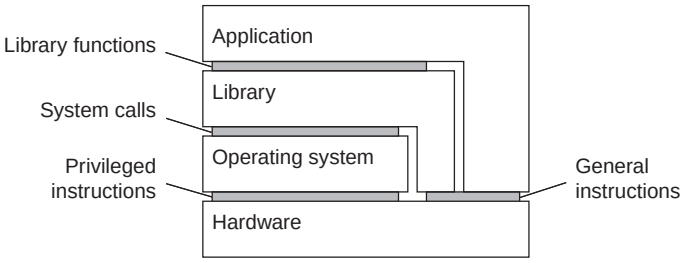
De essentie van virtualisatie is het gedrag nabootsen van de interfaces tussen de systeem lagen. De computerinterfaces van een computer systeem worden weergegeven op figuur 3.4 op de volgende pagina.

Virtualisatie kan twee verschillende vormen aannemen. De eerste maakt gebruik van een runtime systeem die een instructie set bevat waarvan uitvoerende applicaties gebruik kunnen maken. Zo kunnen instructies geïnterpreteerd worden (Java VM) of ze kunnen worden geëmuleerd. Dit type van virtualisatie wordt een *process virtual machine* genoemd, wat wijst op de virtualisatie voor een proces.

[3] A legacy system is an old method, technology, computer system, or application program, öf, relating to, or being a previous or outdated computer system."



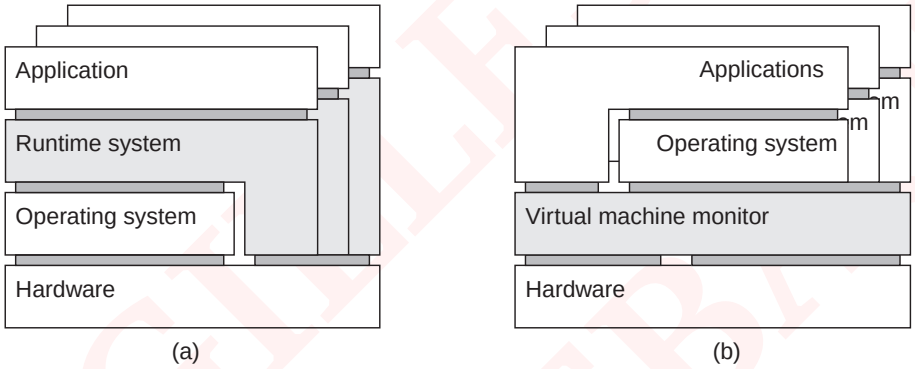
Figuur 3.3: (a) Algemene organisatie tussen een programma, interface en het systeem. (b) Algemene organisatie van ee virtualisatie systeem A boven een systeem B.



Figuur 3.4: Verschillende interfaces aangeboden door computer systemen

Een tweede mogelijkheid is het aanbieden van een extra laag die een volledige instructie set bevat, die de hardware volledig verbergt. Deze interface wordt gelijktijdig aangeboden aan verschillende programma's waardoor er nu verschillende besturingssystemen onafhankelijk en parallel op hetzelfde platform kunnen draaien. Virtual machine monitors laten ons toe om SW naar andere HW te porten. Programma's worden volledig worden geïsoleerd.

Deze twee types worden weergegeven in figuur 3.5.



Figuur 3.5: (a) een process virtual machine, met meerdere instanties van (applicatie, runtime) combinaties. (b) Een virtual machine monitor, met meerdere instanties van (applicaties, besturingssysteem) combinaties.

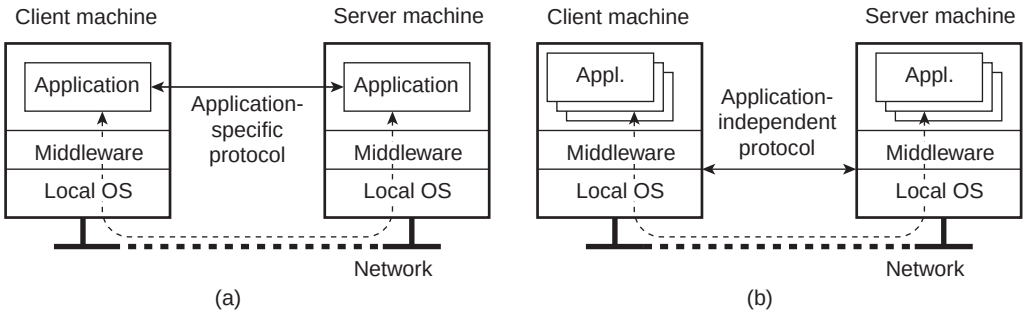
3.3 Clients

Clients kunnen op twee manieren interageren met een server. Een client kan via een application-level protocol synchronisatie toelaten, e.g. agenda synchronisatie met een smartphone. Hierbij spreekt men van fat-clients.

Een tweede oplossing is dat de client een directe verbinding heeft met de remote services door een user interface. Hierbij zullen deze thin-clients zich gedragen als terminals waarbij alles wordt verwerkt en opgeslagen op de server.

3.3.1 Networked User Interfaces

In het geval van de thin-client aanpak spreekt men van networked user interfaces. Dit is een oplossing voor remote info retrieval. Een voorbeeld hierva is het X Window systeem binnen UNIX.



Figuur 3.6: (a) Een networked applicatie met zijn eigen protocol. (b) een Algemene oplossing die toegang naar remote applicaties toestaat.

3.3.2 Client-side Software for Distribution Transparency

Access transparancy

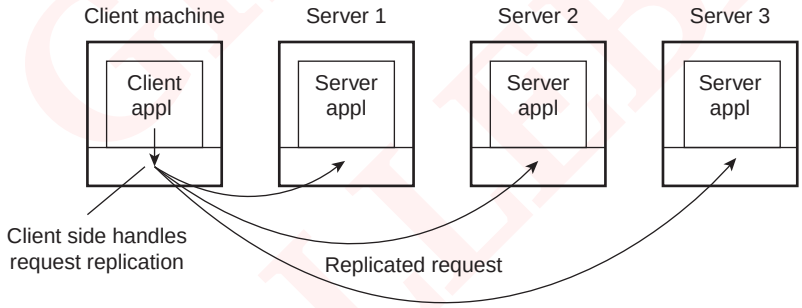
Er kan een client-stub worden gegenereerd om de interface van de server te kunnen aanspreken.

Location, migration en relocation transparancy

De clients-middleware kan de geografische locatie van een gebruiker verbergen en veranderingen doorvoeren naar de server.

Replication transparancy

Client-side software van transparant replica's aanspreken en de responses gebundeld doorgeven aan de client applicatie.



Figuur 3.7: Transparante replicatie van servers door een client-side oplossing.

Failure transparancy

De client kan de server meerdere malen aanspraken tijdens het afwezig blijven van een antwoord van de server.

Concurrrency en persistence transparancy

Worden beiden door servers geleverd. In het eerste geval door tussenliggende servers.

3.4 Servers

Een server is een implementatie van een specifieke dienst die geleverd wordt voor clients. De organisatie van een server is telkens dezelfde: wacht op een inkomende request - handle deze af - wacht op de volgende request - ....

3.4.1 General Design Issues

Server Organisations

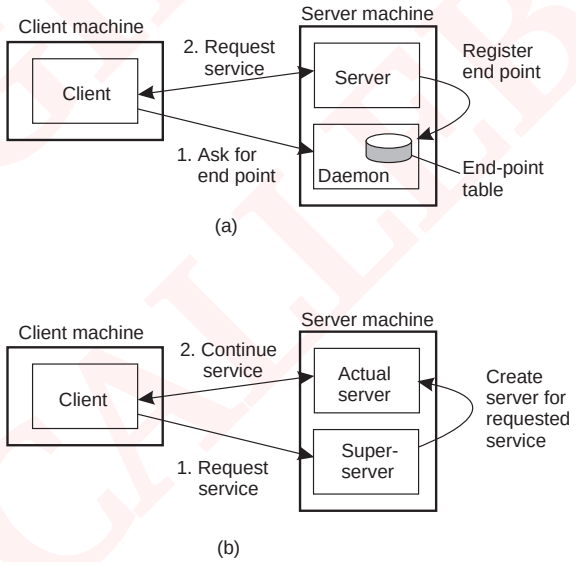
In het geval van een *iteratieve server* zal de server de request zelf afhandelen en als nodig een response sturen naar de client.

Een *concurrent server* zal de request niet zelf afhandelen, maar deze doorgeven aan een aparte thread of proces, waarna het direct wacht op de volgende inkomende request.

Server Contact Point

Een andere probleem die clients ondervinden is, op welke end-point (`[ip address, port number]`) luisteren welke servers. Belangrijke diensten hebben hun eigen poortnummer toegewezen gekregen. In het volgende hoofdstuk wordt er ingegaan op het verkrijgen van de ip adressen van de machines waar de dienst draait.

Server Creation



Figuur 3.8: (a) Client-to-server binding d.m.v. een daemon. (b) Client-to-server binding d.m.v. een super-server.

Beforehand server creation (deamon) Een machine heeft een deamon die luistert op een gekende poort, deze deamon beheert de servers. De client contacteert de deamon, waarna hij de

specifieke server kan contacteren. Dit heeft als voordeel dat er flexibeler kan worden omgesprongen met poorten.

On-demand server creation (superserver) In plaats al die passieve processen te moeten bijhouden maakt men gebruik van een superserver die luistert naar elke end point van de services. Hierdoor zullen er geen passieve servers zijn, er kunnen meer servers afgesloten worden na voltooiing van een request. In UNIX zal de *inetd* deamon luisteren naar internet service poorten en een proces *forken* die de request kan afhandelen. Het nadeel is de performantie overhead vermits er nog een server moet worden opgestart op het moment van de aanvraag.

Stateful versus stateless servers

Een stateless server houdt geen informatie bij over clients. een Webserver bijvoorbeeld zal enkel antwoorden op afzonderlijke HTTP requests.

Een statefull server zal echter persistent informatie behouden van de clients, e.g. een FTP-server. Waar er een onderscheidt kan gemaakt worden tussen *session state* en *permanent state*. In het laatste geval kunnen we spreken over opslag in databanken, in het eerste geval over de staat binnen een serie van operaties.

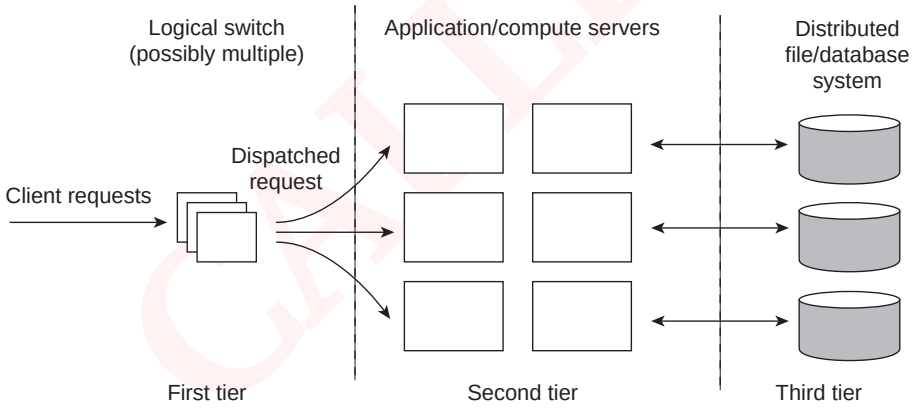
Cookies in webbrowsers worden initieel verstuurd van de server naar clients. Elke keer nu een client een website bezoekt zal deze de cookie meegeven.

3.4.2 Server Clusters

Een server cluster is niets meer dan een verzameling van machines die verbonden zijn via een netwerk.

In de meeste gevallen worden server clusters logisch georganiseerd in drie lagen. De eerste laag zal clients requests dispatchen naar één van de servers.

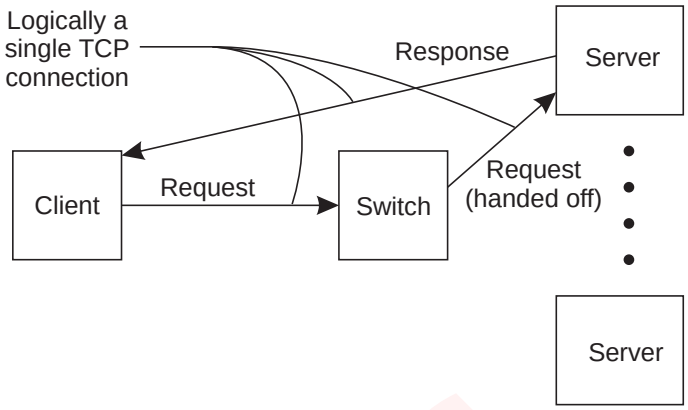
Voor schaalbaarheid en beschikbaarheid kunnen de clusters ook meerdere access points hebben.



Figuur 3.9: De algemene organisatie van een three-tiered server cluster

In het geval van *transport-layer switches* zal de switch een inkomende TCP connectie aanvaarden en deze doorgeven aan één van de servers. Dit noemt men *TCP handoff*. De server zal de request afhandelen en zal een akcnowledgment terug sturen naar de client, maar met als source field de

switch. Hierbij zal de dispatcher of de switch de bottleneck zijn –vermits al de requests via de switch moet lopen.



Figuur 3.10: Het principe van TCP handoff

In het ander geval kan de switch de poortnummer en IP adres meegegeven van de machine die de request zal afhandelen, maar dit brengt communication latency met zich mee door de extra connecties.

3.5 Summary

Processes play a fundamental role in distributed systems as they form a basis for communication between different machines. An important issue is how processes are internally organized and, in particular, whether or not they support multiple threads of control. Threads in distributed systems are particularly useful to continue using the CPU when a blocking I/O operation is performed. In this way, it becomes possible to build highly-efficient servers that run multiple threads in parallel, of which several may be blocking to wait until disk I/O or network communication completes.

Organizing a distributed application in terms of clients and servers has proven to be useful. Client processes generally implement user interfaces, which may range from very simple displays to advanced interfaces that can handle compound documents. Client software is furthermore aimed at achieving distribution transparency by hiding details concerning the communication with servers, where those servers are currently located, and whether or not servers are replicated. In addition, client software is partly responsible for hiding failures and recovery from failures.

Servers are often more intricate than clients, but are nevertheless subject to only a relatively few design issues. For example, servers can either be iterative or concurrent, implement one or more services, and can be stateless or stateful.

Special attention needs to be paid when organizing servers into a cluster. A common objective is hide the internals of a cluster from the outside world. This means that the organization of the cluster should be shielded from applications. To this end, most clusters use a single entry point that can hand off messages to servers in the cluster. A challenging problem is to transparently replace this single entry point by a fully distributed solution.

Hoofdstuk 4

Communicatie

In dit hoofdstuk bekijken we drie wijdgebruikte modellen voor communicatie: RPC, MOM en data streaming.

Inhoudsopgave

4.1 Fundamentals

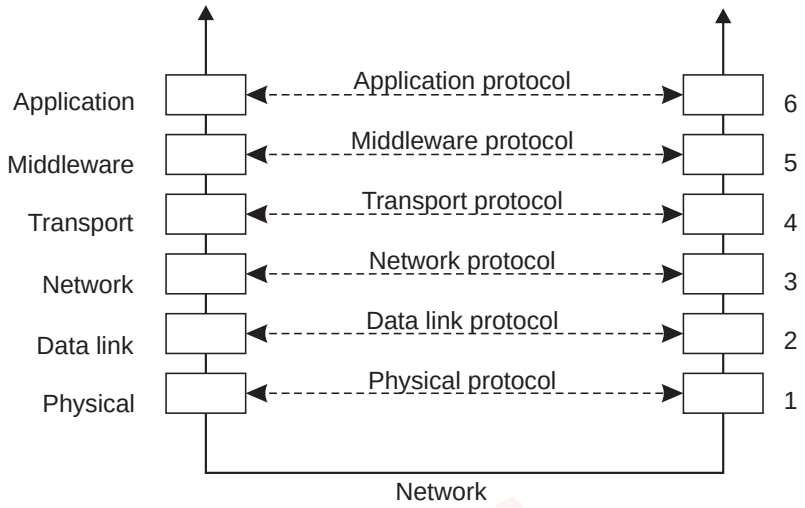
4.1.1 Layered Protocols

De Internetprotocollen per laag worden hier besproken.

Lower-Level Protocols

De fysische laag zend enkel bits. De datalink laag zal instaan voor het detecteren en verbeteren van fouten van de frames[1]. De netwerklaag zal ervoor zorgen dat de pakketten hun weg vinden naar de juiste ontvanger, dit is routing. Het meest gebruikte netwerk protocol is het connectionless IP, waarbij elke IP pakket onafhankelijk van de ander pakketten worden verzonden naar hun bestemming.

[1] Een frame is een groep bits die worden bekeken op het niveau van de datalink laag.



Figuur 4.1: Een bewerkt referentie model voor communicatie over netwerken.

Transport Protocols

De transport laag is de laatste laag van een basis netwerk protocol stack. Vanaf de transport laag kunnen applicatie ontwikkelaars het onderliggende netwerk gebruiken. Reliable transport connecties kunnen wordne gebouwd bovenop connection-oriented of connectionless netwerk diensten. De transportlaag moet ervoor zorgen dat alle pakketten geordend en volledig zijn. Dit zorgt voor een end-to-end communicatie gedrag.

De internet connection-oriented en connectionless transport protocollen zijn respectievelijk TCP en UDP.

Higher-Level Protocols

In de praktijk worden de drie bovenste lagen versmolten tot één laag, i.e. de applicatie laag. Voorbeelden van applicatie protocollen zijn FTP en HTTP.

Middleware Protocols

Middleware is een applicatie dat zich logisch in de applicatielaag bevindt. Deze bevat protocollen voor high-level communicatie en voor het starten van middleware services.

Protocollen voor het opzetten van middleware services Voorbeelden hiervan zijn authenticatie/autorisatie van gebruikers en processen, commit protocollen waarbij alles of niets wordt uitgevoerd (dit wordt ook wel eens atomicity genoemd) en gedistribueerde locking protocollen.

High-level communicatie protocollen In dit hoofdstuk wordt er gesproken over de communicatie protocollen binnen gedistribueerde systemen: RPC, message queuing, media streams en multicasting.

4.1.2 Types of Communication

De types van communicatie geleverd door een middleware kan worden onderverdeeld in persistente of transiënte communicatie enerzijds, en asynchrone of synchrone communicatie anderzijds.

Persistent vs. Transient Communication

Elektronische mail is een voorbeeld van persistente communicatie, waarbij een bericht is opgeslagen door de communicatie middleware. Dit in tegenstelling tot transiënte communicatie waarbij berichten enkel worden opgeslagen als de entiteiten actief zijn. Transport-level communicatie zullen enkel transiënte communicatie ondersteunen, e.g. store-and-forward routers.

Asychronous vs. Synchronous Communication

Bij asynchrone communicatie zal de zender onmiddelijk verder uitvoeren na transmissie. Bij synchrone communicatie zal de zender geblocked zijn tot zijn request is geaccepteerd door de middleware of ontvanger, of door een antwoord ontvangen te hebben van de bestemming. Wat neert komt op: blokkeer tot het bericht verzonden, aangekomen of afgehandeld is.

Combinaties van types van communicaties

Volgende combinaties zijn veel gebruikt:

- Persistentie en synchronisatie tot request indiening
  Dit is een gebruikt schema voor message-queuing systemen.
- Transiënte communicatie en synchronisatie tot request afhandeling
  Dit schema wordt gebruikt voor RPC.

4.2 Remote Procedure Call

RPC wordt gebruikt om access transparantie te verwezenlijken. Met RPC is het mogelijk om remote methodes te gaan oproepen, waarbij het lijkt alsof de methode-oproepen lokaal gebeurt. Andere machines beschikken misschien over meer processing power of de databank staat op een server waardoor het nodig is om te communiceren met andere machines.

We ondervinden hierdoor problemen: verschillende machines hebben niet dezelfde adres ruimte en machine-failure heeft impact op de executie.

4.2.1 Basic RPC Operation

We bekijken eerst de conventionele procedure call om dan uit te leggen hoe deze kan worden opgesplitst in een client en server gedeelte.

Conventional Procedure Call

Parameters kunnen worden doorgegeven via de volgende mechanismen:

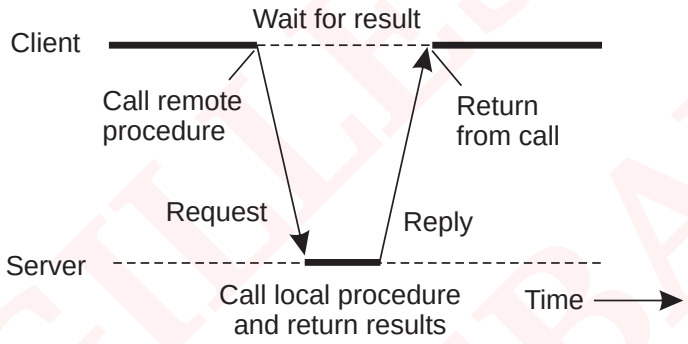
`Call-by-value`
Hierbij wordt de waarde van een parameter doorgegeven.

`Call-by-reference`
De referentie van een parameter wordt doorgegeven, i.e. het adres van de variabele.

`Call-by-copy/restore`
In dit mechanisme wordt er eerst een `call-by-value` parameter gebruikt, maar na het uitvoeren van de procedure wordt er opnieuw een kopie gemaakt van de parameter. Dit kopie zal de oude waarde overschrijven. Wat hetzelfde effect heeft als de `call-by-reference`.

Client and Server Stubs

De client-stubs zullen procedure oproepen voor andere machines omzetten naar berichten die dan worden verstuurd naar de server. De servers OS zal dan deze berichten geven aan de server-stub. Deze stub zal dan de inkomende requests omzetten naar lokale procedure calls. Dit met de bedoeling om RPC te laten gelijken op een lokale procedure call.



Figuur 4.2: Het principe van RPC tussen een client en server programma.

Een remote procedure call gaat als volgt:

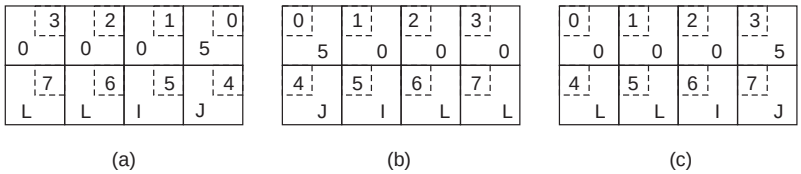
1. The client method calls the *client stub* normally.
2. The *client stub* builds a message and calls the local OS.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the *server stub*.
5. The *server stub* unpacks the parameters and calls the server.
6. The server does the work and returns the result to the *stub*.
7. The *server stub* packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the *client stub*.
10. The *stub* unpacks the result and returns to the client.

4.2.2 Parameter Passing

Passing Value Parameters

Parameters inpakken in een bericht wordt *parameter marshaling* genoemd.

Er moet rekening gehouden worden met de formaten, zoals big of little endian, en welke bitsequenties string en integers voorstellen. Het verschil tussen big en little endian wordt weergegeven op figuur 4.3.



Figuur 4.3: (a) Het originiele bericht op een Pentium. (b) Het bericht na transmissie op een SPARC. (c) Het bericht achter inversie van (b). (De kleine nummers in de kotjes geven het adres van elke byte weer.)

Passing Reference Parameters

Er zijn enkele strategieën om om te gaan met referentie parameters:

- Verbieden van pointers als parameters
- Gebruik maken van `copy/restore` ter vervanging van referenties
- Pointers die worden doorgegeven naar de server worden in de server-stub eerst omgezet. De server vraagt de data van de pointer aan de client.

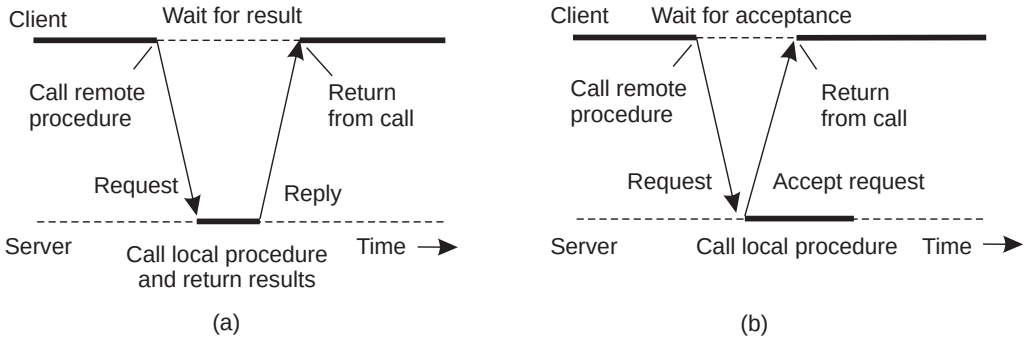
Parameter Specification and Stub Generation

De caller en callee moeten een formaat voor de parameters en message exhange protocol afspreken.

Client en server stubs kunnen worden gegenereerd op basis van een interface definition language (IDL).

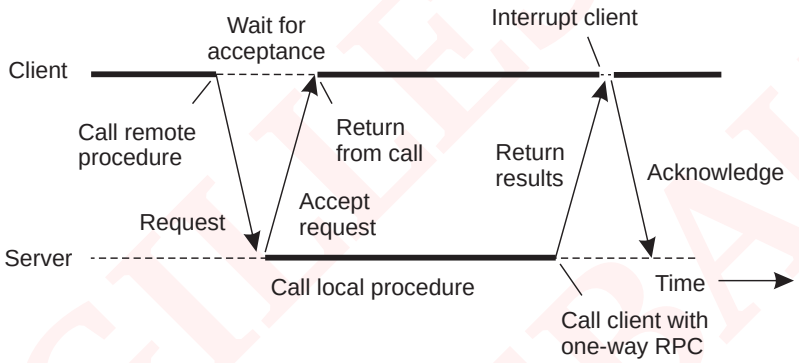
4.2.3 Asynchronous RPC

Met asynchrone RPCs zal de server direct een antwoord terugsturen naar de client vanaf het RPC request is aangekomen.



Figuur 4.4: (a) De interactie tussen client en server in een traditionele RPC. (b) De interactie door asynchrone RPC.

Een andere mogelijkheid voor asynchrone RPC is dat de client ondertussen andere zaken kan afhandelen voor het antwoord van de server. Dit noemt men *deferred synchronous RPC* en is weergegeven op figuur 4.5.



Figuur 4.5: Een client en server interactie door twee asynchrone RPCs (*deferred synchronous RPC*).

Als laatste is het ook mogelijk om niet te wachten op een *acknowledgment* van de server, dit noemt men *one-way RPC*.

4.2.4 DCE RPC (Example)

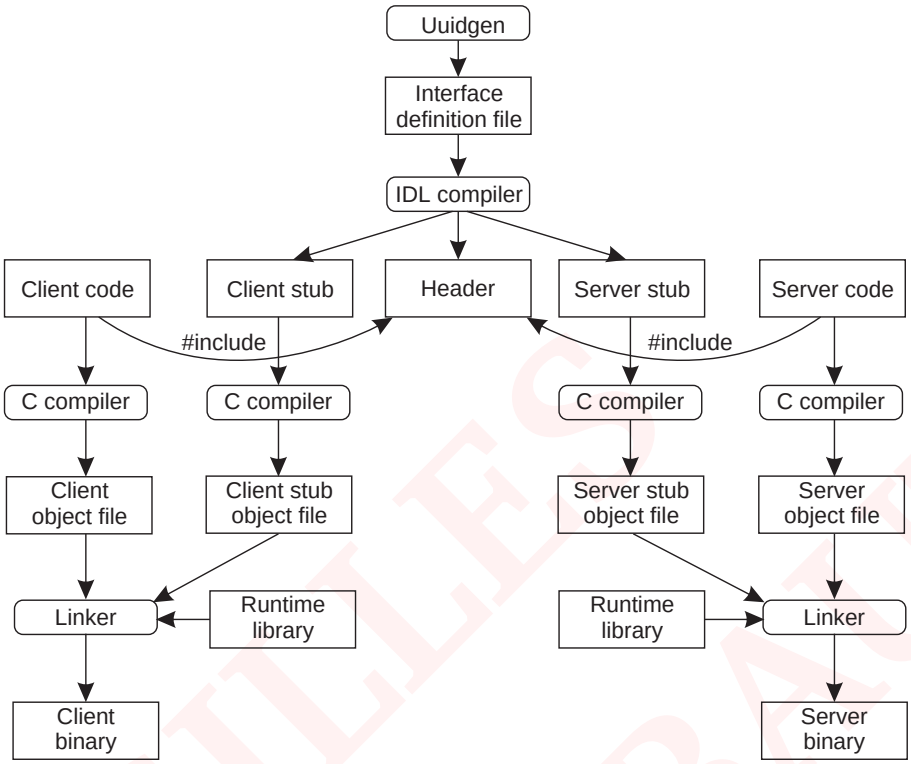
Distributed Computing Environment, een specifiek RPC systeem, wordt in deze sectie als voorbeeld besproken.

Dit systeem biedt enkele middleware services aan:

- Distributed file service (transparante access van bestanden)
- Directory service (bijhouden locatie van resources)
- Security service (autorisatie/authenticatie)
- Distributed time service (globale synchronisatie van klokken)

Writing a Client and a Server

In een client-server systeem is alles verbonden via de interface definition, zoals gespecificeerd in de IDL.



Figuur 4.6: Stappen om een client en een server in DCE RPC te schrijven.

Elk IDL bestand heeft een globale unieke identifier[2] voor een interface. De client stuurt deze identifier mee met de eerste RPC om zeker te zijn dat hij de correcte server aanspreekt.

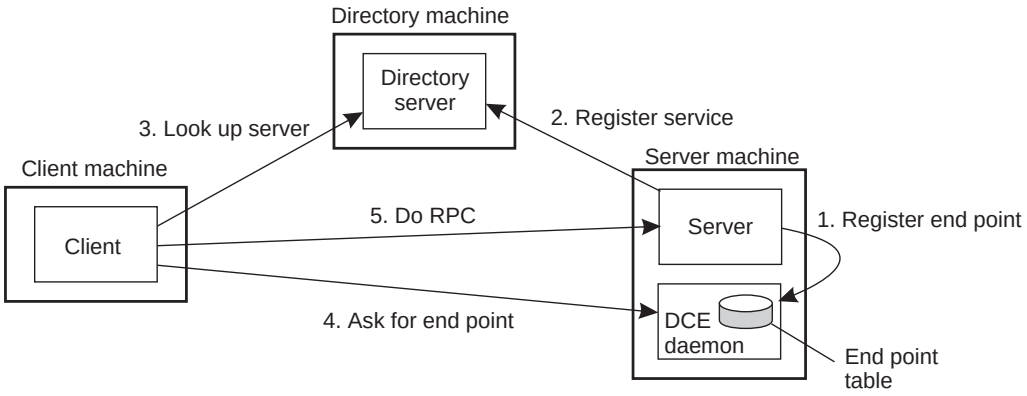
Binding a Client to a Server

Een client met de machine van de server en het correcte proces op die machine weten te vinden.

De *DCE daemon* houdt een mapping tussen de server en zijn end points bij. De server moet zich eerst registreren bij de daemon alvorens inkomende requests te kunnen ontvangen.

De server registreert zich ook bij de *directory service*, waarbij hij zijn netwerk adres en naam meegeeft. Via die naam kan de server opgezocht worden.

[2] De unieke identificatie wordt gegeneerd door `uuidgen` op basis van de tijd van creatie en de locatie.



Figuur 4.7: Client-to-server binding in DCE.

De volgende stappen worden ondernomen door de client die een server zoekt:

1. Geeft de server naam door aan de *directory server*, deze geeft het netwerk adres van de machine terug. Deze server gedraagt zich als een look up service.
2. De client contacteert dan de DCE daemon van die machine (met een gekende end-point). De daemon zokt in zijn *end point table* het end-point op van de server.
3. De client kan nu de server contacteren, i.e. de RPC starten.

4.3 Message-Oriented Communication

Via MOM is het mogelijk om af te stappen van de assumptie dat beiden partijen actief moeten zijn. Bij message-oriented communication kan er een onderscheidt gemaakt worden tussen transiënte en persistente communicatie.

4.3.1 Message-Oriented Transient Communication

Gedistribueerde systemen kunnen worden opgebouwd bovenop het message-oriented model die de transportlaag aanbiedt. Transient staat hier voor het feit dat beide partijen actief moeten zijn, i.e. client en server kunnen berichten ontvangen/versturen.

Berkeley Sockets

Een socket is een communicatie end-point naar waar een applicatie data kan schrijven/lezen door gebruik van het onderliggend netwerk.

De socket primitieven voor TCP/IP:

`socket()` creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.

`bind()` is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address.

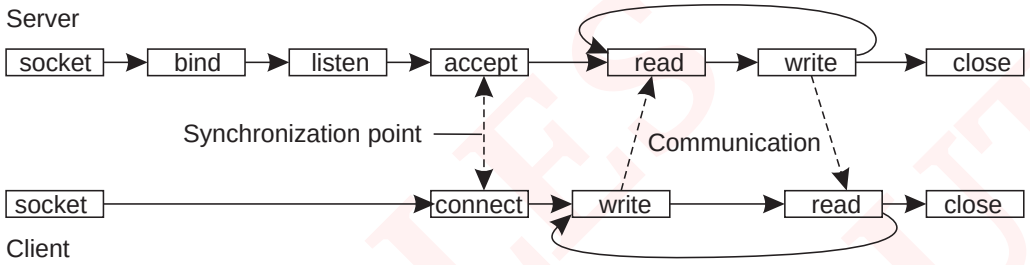
`listen()` is used on the server side, and causes a bound TCP socket to enter listening state.

`connect()` is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.

`accept()` is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.

send() and `recv()`, or `write()` and `read()`, or `sendto()` and `recvfrom()`, are used for sending and receiving data to/from a remote socket.

`close()` causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.



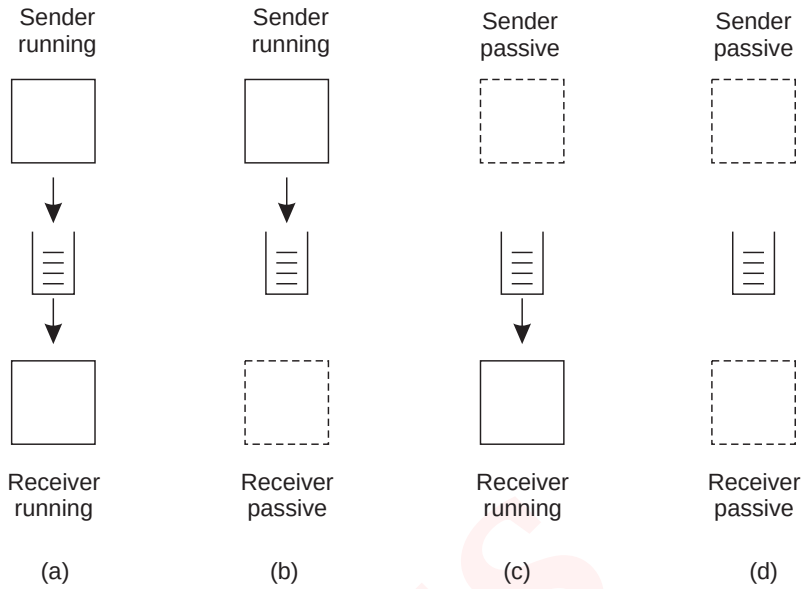
Figuur 4.8: Connection-oriented communicatie patroon door gebruik van sockets.

4.3.2 Message-Oriented Persistent Communication

MOM of anders genoemd *message-queuing systems*, laten toe dat de clients of servers niet actief hoeven te zijn. De communicatie is hierdoor persistent en asynchroon.

Message-Queuing Model

Door MOM zullen berichten direct in een queue moeten terecht komen, waarbij er geen garantie is van de leveringstijd. Hierdoor worden systemen los-gekoppeld in de tijd.



Figuur 4.9: Vier combinaties voor los-gekoppelde communicatie door gebruik van queues.

De basis interface voor een wachtrij in een message-queuing systeem:

`Notify` Install a handler to be called when a message is put into the specified queue.

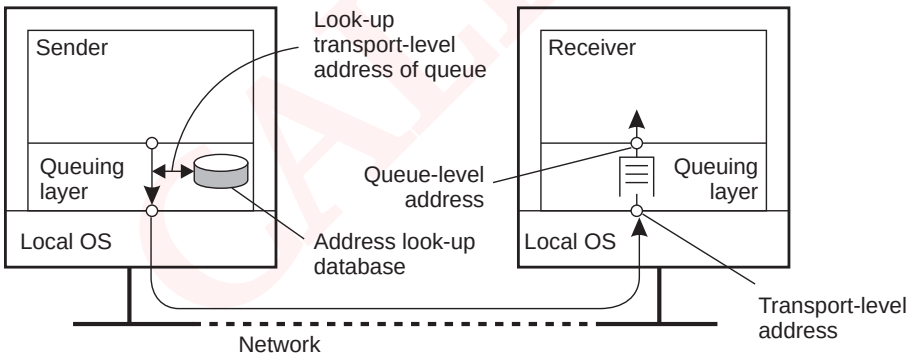
`Poll` Check a specified queue for messages, and remove the first. Never block.

`Get` Block until the specified queue is nonempty, and remove the first message.

`Put` Append a message to a specified queue.

General Architecture of a Message-Queuing System

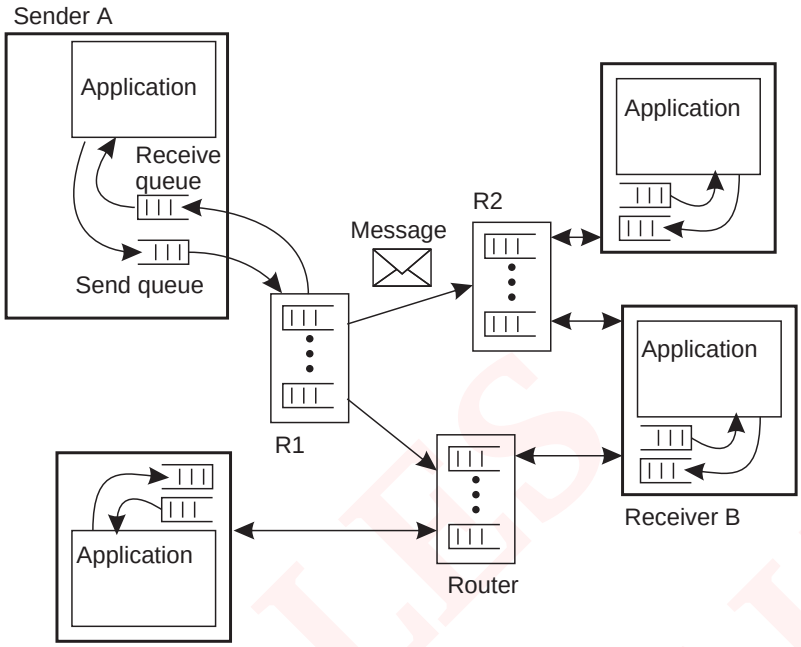
De MOM moet een mapping bijhouden van queues en netwerklocaties, i.e. een databank van *queue names* en netwerk locaties.



Figuur 4.10: De relatie tussen queue-level addressing en network-level addressing.

Als de destination queue niet bereikbaar is, moet men gebruik maken van *relays*.Dit wordt weergegeven in figuur 4.11 op de volgende pagina. Deze relays kunnen gebruikt wordne voor multicasting

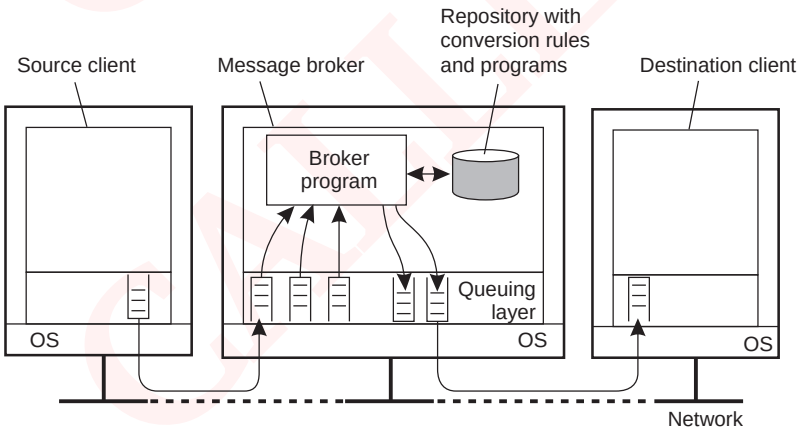
waarbij inkomende berichten gewoon in elke send queue wordt gestoken. Als laatste kan een relay ook zorgen voor het omzetten van het formaat van berichten zodat die kan worden verstaan door de gebruiker, dit noemt men *message brokers*.



Figuur 4.11: Een algemene organisatie van een message-queuing systeem met routers.

Message Brokers

Men zal een vast formaat van berichten vastleggen, die dan kunnen worden omgevormd door *message brokers*.



Figuur 4.12: Een algemene organisatie van een message broker in een MOM.

4.4 Summary

Having powerful and flexible facilities for communication between processes is essential for any distributed system. In traditional network applications, communication is often based on the low-level message-passing primitives offered by the transport layer. An important issue in middleware systems is to offer a higher level of abstraction that will make it easier to express communication between processes than the support offered by the interface to the transport layer.

One of the most widely used abstractions is the Remote Procedure Call (RPC). The essence of an RPC is that a service is implemented by means of a procedure, of which the body is executed at a server. The client is offered only the signature of the procedure, that is, the procedure's name along with its parameters. When the client calls the procedure, the client-side implementation, called a stub, takes care of wrapping the parameter values into a message and sending that to the server. The latter calls the actual procedure and returns the results, again in a message. The client's stub extracts the result values from the return message and passes it back to the calling client application.

RPCs offer synchronous communication facilities, by which a client is blocked until the server has sent a reply. Although variations of either mechanism exist by which this strict synchronous model is relaxed, it turns out that generalpurpose, high-level message-oriented models are often more convenient.

In message-oriented models, the issues are whether or not communication is persistent, and whether or not communication is synchronous. The essence of persistent communication is that a message that is submitted for transmission, is stored by the communication system as long as it takes to deliver it. In other words, neither the sender nor the receiver needs to be up and running for message transmission to take place. In transient communication, no storage facilities are offered, so that the receiver must be prepared to accept the message when it is sent.

In asynchronous communication, the sender is allowed to continue immediately after the message has been submitted for transmission, possibly before it has even been sent. In synchronous communication, the sender is blocked at least until a message has been received. Alternatively, the sender may be blocked until message delivery has taken place or even until the receiver has responded as with RPCs.

Message-oriented middleware models generally offer persistent asynchronous communication, and are used where RPCs are not appropriate. They are often used to assist the integration of (widely-dispersed) collections of databases into large-scale information systems. Other applications include e-mail and workflow.

Na dit hoofdstuk wordt de lezer verwezen naar hoofdstuk 10 en 12. Deze hoofdstukken bespreken case-studies die toepasbaar zijn bij de net geziene leerstof.

Hoofdstuk 5

Naming

Inhoudsopgave

Name resolution laat een proces toe om de entiteit te benaderen. In een gedistribueerde omgeving zal het benamingssysteem vaak zelf gedistribueerd zijn.

In dit hoofdstuk concentreren we ons op drie verschillen types van *naming services*.

1. Organisatie en implementatie van *human-friendly* namen, e.g. bestandssystemen en WWW
2. Locatie bepalingen m.b.v. plaats-onafhankelijke benamingen, e.g. mobiele telefonen en distributed hash table
3. Benamingen door de karakteristieken te beschrijven

5.1 Names, Identifiers and addresses

Entiteiten, e.g. gebruikers, processen, connecties, printers, kunnen worden benaderd door een access point. Via deze access point kunnen er operaties worden uitgevoerd op de entiteiten. Vermits access points kunnen wijzigen moeten we de naam en het adres van elkaar koppelen, zo'n naam noemt men *location independent*.

Een *true identifier* is een naam die de volgende eigenschappen heeft:

1. Een *identifier* refereert naar slechts één entiteit

2. Elke entiteit wordt maar door maximaal één *identifier* verwezen
3. Een *identifier* refereert altijd naar dezelfde entiteit, i.e. het wordt nooit hergebruikt

Adressen en *identifiers* zijn twee belangrijke types van namen die lek gebruikt worden voor verschillende doelen.

Gedistribueerde systemen behouden een name-to-address binding die in zijn simpelste vorm een tabel is met `[name, address]` paren. Maar gedistribueerde systemen bespannen vaak grote netwerken die veel *resources* moeten benamen. Een gecentraliseerde *table* is dus niet mogelijk!

5.2 Flat naming

Een belangrijke eigenschap van flat naming is dat deze geen informatie bevatten over de locatie van het access point van de geassocieerde entiteit.

5.2.1 Simple solutions

Deze twee simpele oplossingen zijn enkel toepasbaar binnen LAN netwerken.

Broadcasting and Mulitcasting

Broadcasting laat toe om aan alle machines –binnen een LAN– te vragen wie de gewenste entiteit bevat. Deze zend dan een bericht terug met het adres van zijn access point. Een voorbeeld hiervoor is Address Resolution Protocol (ARP) die het data-link adres van een machine terugstuurt op basis van een IP adres.

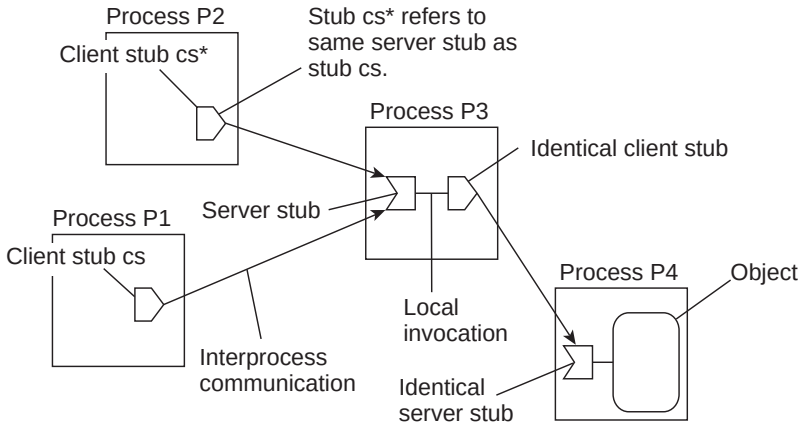
Broadcasting schaalt slecht, waardoor we over moeten schakelen op multicasting, die een bepaalde groep aanspreekt.

Multicasting maakt gebruik van hosts die zich hebben gevoegd tot een multicast groep. zulke groepen worden geïdentificeerd door een multicast adres. Nu zullen de *requests* worden verzonden naar een specifieke multicast groep.

Forwarding pointers

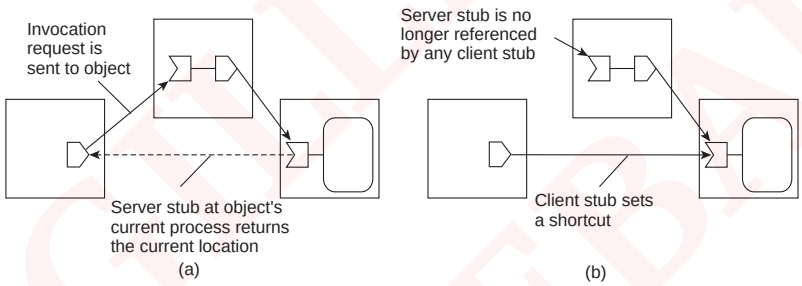
Als een entiteit verhuist van een locatie A naar een locatie B zal er in A een referentie naar B worden bijgehouden. Hierdoor is het mogelijk dat we via de referentie naar A toch bij B uitkomen. Het probleem is echter dat er veel *pointers* en dus een lange *chain* wordt gecreëerd.

SSP chains Een *pointer* wordt geïmplementeerd als een `(client stub, server stub)`. De server laat een *client stub* achter als het zich verplaatst naar een andere locatie, zoals weergegeven in figuur 5.1 op de pagina hierna.



Figuur 5.1: Het principe van forwarding pointers die gebruik maken van `(client stub, server stub)` paren

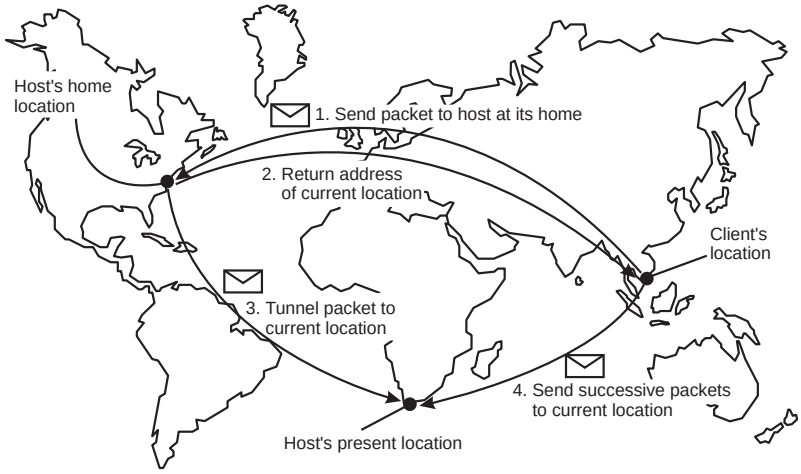
De client redirect wanneer hij informatie heeft over de nieuwe locatie (figuur 5.2). Waarbij er twee manieren zijn voor de antwoorden terug te sturen. Enerzijds kan het server object zijn antwoorden direct naar de client sturen, wat heel snel gebeurt. Ofwel zal het antwoord het omgekeerde pad volgen, waardoor alle stubs kunnen worden aangepast naar de nieuwe locatie.



Figuur 5.2: Redirectie van forwarding pointers door het opslaan van een *shortcut* in de client stub.

5.2.2 Home-Based Approaches

De home location is de LAN waarbinnen een entiteit is gecreëerd. Binnen deze zelfde LAN wordt er ook een home agent aangesteld die bijhoudt waar de entiteiten die binnen zijn LAN zijn gecreëerd gelocaliseerd zitten. Wanneer een entiteit verhuist zal hij een temporaal adres vragen (aan de LAN waar hij momenteel verblijft) en deze registreren bij de home agent, dit is de care-of address. De client kan dan communiceren via de home agent, als de host zich niet meer bevindt in zijn LAN zal de care-of address worden meegedeeld aan de client. Dit is te zien in figuur 5.3 op de pagina hierna.



Figuur 5.3: Het principe van Mobile IP.

Het wordt echter wel een probleem wanneer een entiteit zich permanent heeft verplaatst. Als oplossing hiervoor zal de home location van de entiteit moeten worden gewijzigd. Deze home location kan dan worden opgezocht bij een traditionele *naming service*.

5.2.3 Distributed Hash Tables

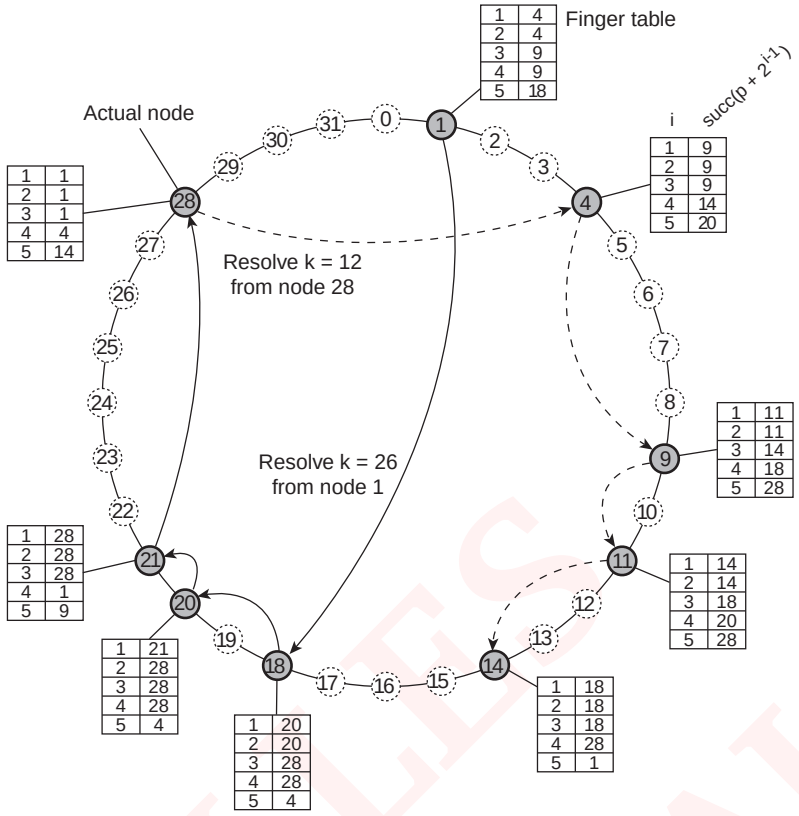
Er zal een *random indentifier* worden gegeven aan nodes, `id`. Elk *item* zal een *random* sleutel toegekend krijgen, `k`. Een item zal moeten worden opgeslagen bij node waarvoor `id==succ(k)`. In het Chord systeem zal er een finger table (FT) worden bijgehouden met *entries* van een aantal andere nodes.

$$FT_p[i] = succ(p + 2^{i-1})$$

Als er een look-up moet gebeuren voor sleutel k zal node p deze forwarden naar node q met index j in $p's$ FT waarbij:

$$FT_p[j] \leq k < FT_p[j+1]$$

Zo zal er de look-up performanter zijn dan enkel de buren bij te houden ($O(\log(N))$).



Figuur 5.4: Ophalen van sleutel 26 van node 1 en sleutel 12 van node 28 in een Chord system.

Exploiting Network Proximity

In het Chord systeem kan het zijn dat de nodes die moeten gevolgd worden om een sleutel te bereiken, ver verspreid zijn. Dit kan worden voorkomen door *id*'s te geven aan noden in de buurt, hierbij zijn er drie mogelijkheden:

1. Topology based assignment of identifiers
   De wereld wordt gemapped op een 1-dimensionale ring en zorgt dat dichtstbijliggende nodes, sleutels hebben die ook dicht bij elkaar zitten. Het nadeel hieraan is dat er geen uniforme distributie van sleutels is en er dus bij het wegvallen van netwerk, een leegte ontstaat.

2. Proximity routing
   Er wordt hier een lijst bijgehouden van alternatieven per entrie. Wat ook mogelijk maakt dat een node failure niet altijd zal leiden tot een lookup failure.

3. Proximity routing
   De dichtste node wordt gekozen als buur.

5.3 Attribute-based naming

Attribute-based naming is een naam-systeem die een entiteit beschrijft in termen van `(attribute,value)` paren. De client kan zo via het instellen van de *value* een set van *cons-*

traints op de gezochte *entities* opleggen, zodat hij enkel *entities* krijgt die hem interesseren.

5.3.1 Directory Services

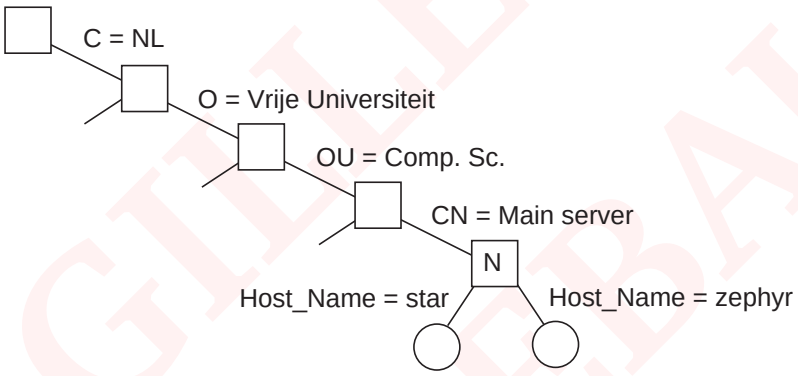
Attribute-based naming systemen zijn ook gekend onder de noemer, directory services.

In tegenstelling tot structurele benaming systemen, zal er hier exhaustief doorheen de *descriptors* moeten worden gezocht. Hierbij zijn er meerde implementatie mogelijk voor het aanpakken van dit probleem,. Hieronder worden er twee besproken, i.e. hierarchical en decentrilazed implementaties.

5.3.2 Hierarchical Implementations: LDAP

LDAP is een protocol die gebruikt wordt voor een LDAP directory service aan te bieden, deze bestaat een een aantal *records*, *directory entries*. Een *directory entry* bestaat uit een collectie van `(attribute, value)` paren met elk een geassocieerd type. Een collectie van alle *directory entries* in een LDAP *directory service* noemt men een Lightweight Directory Access Protocol (DIB).

Via een Relative Distinguished Name (RDN) kunnen we een globale unieke naam bekomen, dit is een sequentie van *naming attributes* in elke *record*.



Figuur 5.5: Een gedeelte van een DIT

5.4 Summary

Names are used to refer to entities. Essentially, there are three types of names. An address is the name of an access point associated with an entity, also simply called the address of an entity. An identifier is another type of name. It has three to only one entity, and is never assigned to another entity. Finally, human-friendly names are targeted to be used by humans and as such are represented as character strings. Given these types, we make a distinction between flat naming, structured naming, and attribute-based naming.

Systems for flat naming essentially need to resolve an identifier to the address of its associated entity. This locating of an entity can be done in different ways. The first approach is to use broadcasting or multicasting. The identifier of the entity is broadcast to every process in the distributed system. The process offering an access point for the entity responds by providing an address for that access point. Obviously, this approach has limited scalability.

A second approach is to use forwarding pointers. Each time an entity moves to a next location, it leaves behind a pointer telling where it will be next. Locating the entity requires traversing the path of forwarding pointers. To avoid large chains of pointers, it is important to reduce chains periodically.

A third approach is to allocate a home to an entity. Each time an entity moves to another location, it informs its home where it is. Locating an entity proceeds by first asking its home for the current location.

A fourth approach is to organize all nodes into a structured peer-to-peer system, and systematically assign nodes to entities taking their respective identifiers into account. By subsequently devising a routing algorithm by which lookup requests are moved toward the node responsible for a given entity, efficient and robust name resolution is possible.

A fifth approach is to build a hierarchical search tree. The network is divided into nonoverlapping domains. Domains can be grouped into higher-level (nonoverlapping) domains, and so on. There is a single top-level domain that covers the entire network. Each domain at every level has an associated directory node. If an entity is located in a domain D, the directory node of the next higher-level domain will have a pointer to D. A lowest-level directory node stores the address of the entity. The top-level directory node knows about all entities.

Structured names are easily organized in a name space. A name space can be represented by a naming graph in which a node represents a named entity and the label on an edge represents the name under which that entity is known. A node having multiple outgoing edges represents a collection of entities and is also known as a context node or directory. Large-scale naming graphs are often organized as rooted acyclic directed graphs.

Naming graphs are convenient to organize human-friendly names in a structured way. An entity can be referred to by a path name. Name resolution is the process of traversing the naming graph by looking up the components of a path name, one at a time. A large-scale naming graph is implemented by distributing its nodes across multiple name servers. When resolving a path name by traversing the naming graph, name resolution continues at the next name server as soon as a node is reached implemented by that server.

More problematic are attribute-based naming schemes in which entities are described by a collection of (attribute, value) pairs. Queries are also formulated as such pairs, essentially requiring an exhaustive search through all descriptors. Such a search is only feasible when the descriptors are stored in a single database. However, alternative solutions have been devised by which the pairs are mapped onto DHT-based systems, essentially leading to a distribution of the collection of entity descriptors.

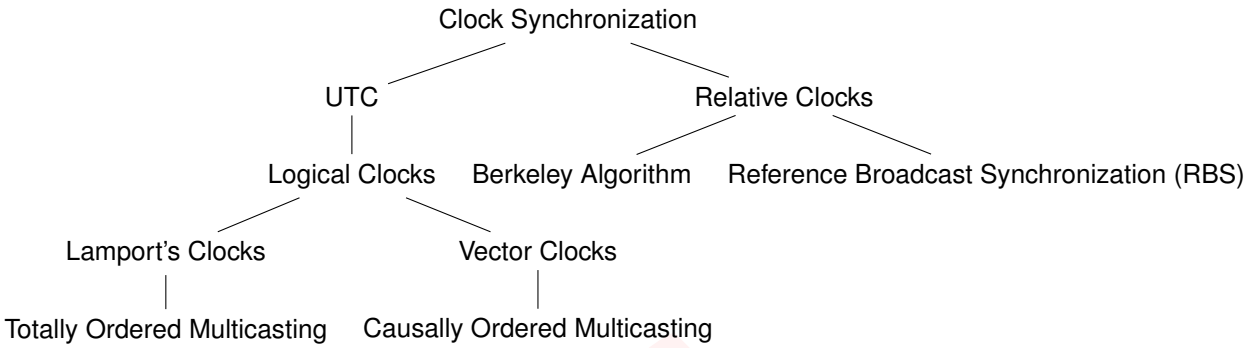
Related to attribute-based naming is to gradually replace name resolution by distributed search techniques. This approach is followed in semantic overlay networks, in which nodes maintain a local Est of other nodes that have semantically similar content. These semantic lists allow for efficient search to take place by which first the immediate neighbors are queried, and only after that has had no success will a (limited) broadcast be deployed.

Hoofdstuk 6

Synchronisatie

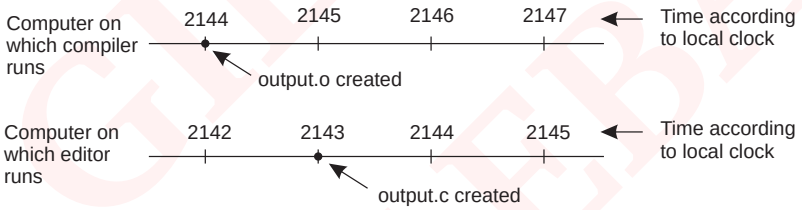
Inhoudsopgave

6.1 Overzicht



6.2 Clock synchronisation

In figuur 6.1 is er een voorbeeld geïllustreerd waarbij de source bestanden en de gecompileerde bestanden zich niet op eenzelfde machine bevinden. Men hoeft enkel aangepast source bestanden opnieuw te compileren. Als de klokken niet gelijk lopen kan het zijn dat de compiler denkt dat we te maken hebben met de nieuwste versie van de gecompilde source code, maar eigenlijk zitten we nog met een oudere versie.



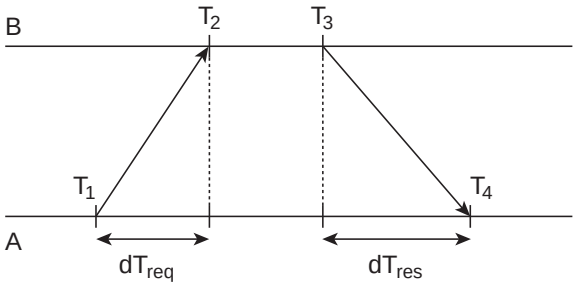
Figuur 6.1: Als elke machine zijn eigen klok heeft, kan een event die gebeurt achter een ander event gezien worden alsof het toch eerder was gebeurt.

6.2.1 Clock Synchronisation algorithms

Universal Coordinated Time (UTC) wordt verzonden d.m.v. radiostations en ontvangen via WWV, d.i. de antennes voor radio te ontvangen. Hierbij kunnen er 2 afwijklingen plaatsvinden eerst een afwijking tijdens het versturen en ontvangen van het signaal, en daarna een afwijking tijdens het propageren van de UTC tijd.

Network Time Protocol (NTP)

Als we een *time server* contacteren die de correcte tijd kent, kunnen we deze tijd overnemen rekening houdend met *delays*. Dit is weergegeven in figuur 6.2 op de pagina hierna.



Figuur 6.2: De huidige tijd van een tijd-server berekenen

De klok wijkt af δ, deze is te berekenen:

$$\delta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

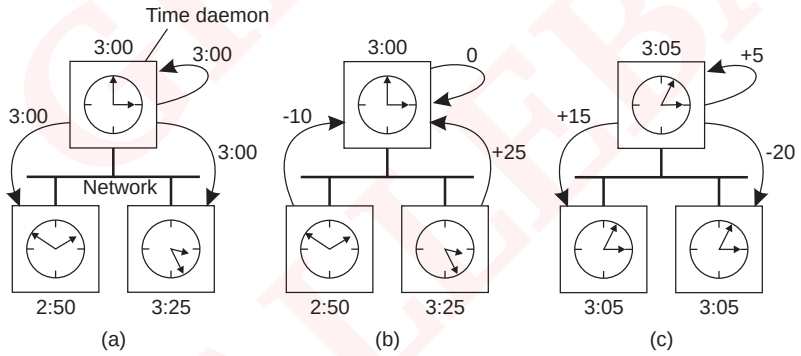
In de praktijk zullen klokken die voorlopen worden vertraagd, maar niet in de tijd worden teruggedraaid.

Een UTC klok heeft *stratum-0*, machines met een WWV receiver zijn *stratum-1* machines, ...

De *time server* is passief dus, deze geeft de tijd enkel op aanvraag van een andere machine.

6.2.2 The Berkeley Algorithm

De *time deamon* zal nu, in tegenstelling tot een *time server*, actief pollen naar de tijd op de andere machines. Het gemiddelde tussen deze klokken wordt berekenend en wordt doorgestuurd. Dit werkt ook als de machine de UTC tijd niet kennen.



Figuur 6.3: (a) De time deamon die de klok-waarden vraagt van alle andere machine. (b) De machines antwoorden. (c) De time daemon vertelt iedereen hoe ze hun klok moeten aanpassen.

- ☺ Er is geen WWV *receiver* nodig
- ☹ De correcte (echte) tijd is niet gekend
- ☹ Onnauwkeurig door de, nog steeds aanwezige, offsets

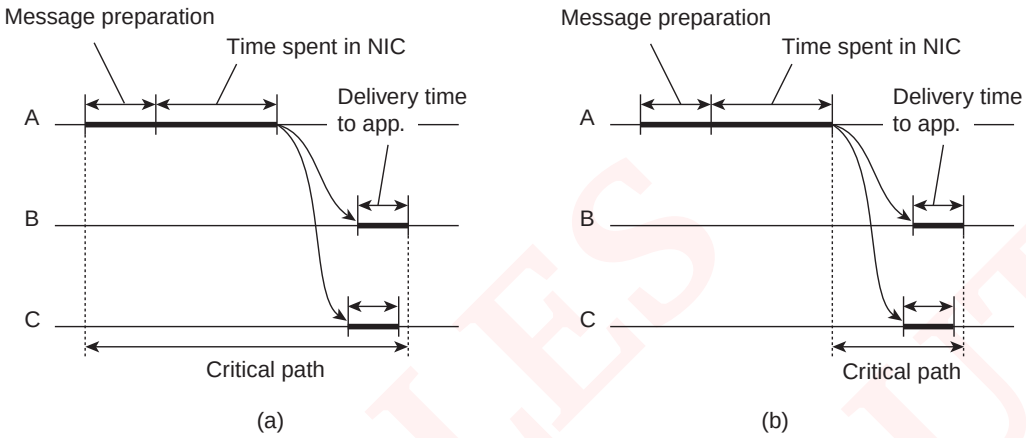
6.2.3 Clock Synchronization in Wireless Networks

Reference Broadcast Synchronization (RBS) is een klok synchronisatie protocol dat enkel intern alle klokken wil synchroniseren, zoals ook het geval is bij het Berkeley Algorithm. Het Berkeley

Algorithm is een two-way protocol (zowel ontvanger als zender zal worden gesynchroniseerd) in tegenstelling tot RBS die enkel de ontvangers zal synchroniseren.

Als de zender de tijd doorstuurd naar zijn ontvangers, en we ervan uitgaan dat we te maken hebben met een vaste delay dan verkrijgen we volgende twee voordelen:

1. Bericht preparation time va nde klok token is niet relevant
2. Tijd die de token doorbrengt in het Netwerk Interface is niet relevant



Figuur 6.4: (a) Het gewoonlijke kritieke pad om de network delays te bepalen. (b) Het kritieke pad in het geval van RBS

Het idee:
Wanneer een node een *reference message* m *broadcast* zal elke ontvangende node de tijd bijhouden waarop hij m heeft ontvangen, $T_{p,m}$. Deze tijd is afgelezen van de lokale klok van p. Als twee nodes elkaars *delivery times* krijgen kunnen ze hun relatieve offset berekenen:

$$offset[p,q] = \frac{\sum_{k=1}^{M}(T_{p,k} - T_{q,k})}{M}$$

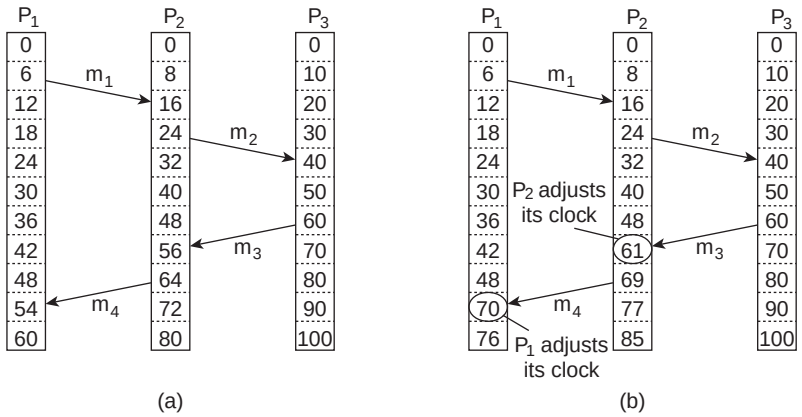
6.3 Logical clocks

In tegenstelling tot vorige klokken, hoeven we ons nu niet te baseren op een gezamelijke tijd. Met logische klokken willen we enkel de volgorde weten waarin *events* voorkomen. We moeten dus geen tijd afspreken tussen verschillende nodes.

6.3.1 Lamport's Logical Clocks

Het algoritme van Lamport werkt op basis van een *happens-before* realtie.

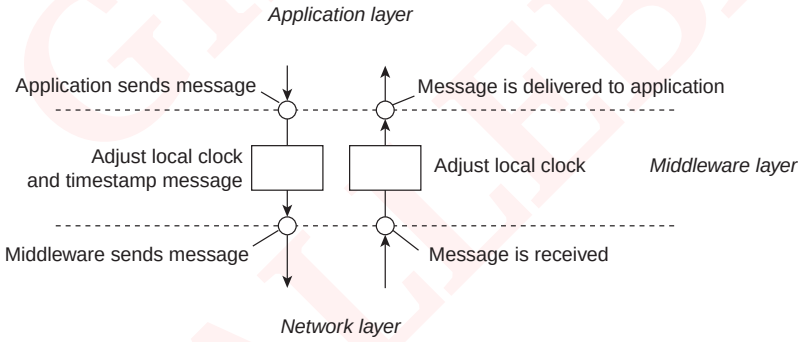
In figuur 6.5 op de volgende pagina worden drie processen met hun eigen klokken voorgesteld, elke klok tikt met een andere snelheid. Het algoritme van Lamport zal tijden aan *events* koppelen en de klokken corrigeren.



Figuur 6.5: (a) Drie processen met elk hun eigen klok. De klokken draaien op verschillende snelheden. (b) Lamport's algorithm corrigeert de klokken.

De implementatie van Lamport's logical clocks is als volgt:
Elk process P_i houdt een lokale *counter* C_i bij en worden geupdate als volgt:

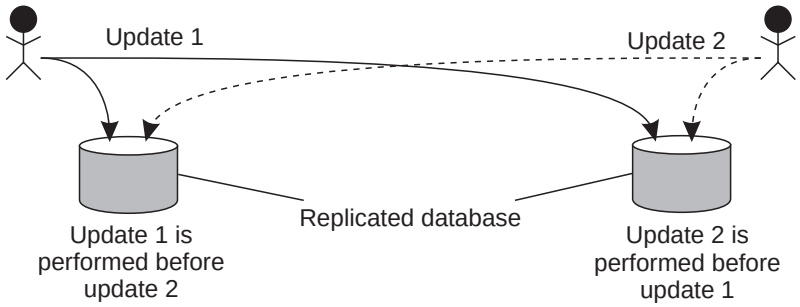
- Voor het uitvoeren van een *event* verhoog de counter met 1:
  C_i++
- Zend het bericht met een *timestamp* C_i
- Bij het ontvangen van het bericht corrigeer de lokale klok van P_j:
  $C_j = max\{C_j, timeStamp(m)\}$



Figuur 6.6: De positionering van Lamport's logical clocks in gedistribueerde systemen

Totally Ordered Multicasting

Lamport's clocks kunnen gebruikt wordne om algoritme van Lamport te implementeren in een gedistribueerde omgeving. Hierbij wordt er verzekerd tot *events* allemaal in dezelfde volgorde toekomen bij alle ontvangers.



Figuur 6.7: Voorbeeld: Updaten van een gerepliceerde databank op een incosistente manier

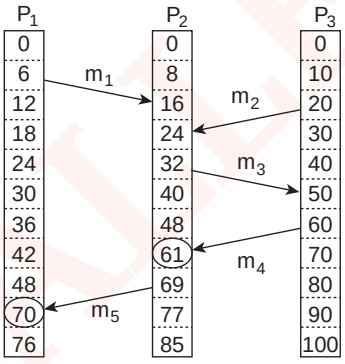
Aanpak:

1. Stuur een bericht naar ontvangers met de lokale *timestamp*
2. Alle ontvangers houden een *queue* bij van ontvanger berichten gesorteerd op de ontvangen *timestamps*. De ontvangers krijgen zo allemaal dezelfde *queue*. De ontvanger zal een *ACK* multicasten met een hogere *timestamp*.
3. Het bericht wordt doorgegeven aan de applicatie-laag als het bericht door elke andere ontvanger werd ge-*ACK*'ed.

Opmerking: We kunnen een unieke volgorde afwdingen door het `MAC` of `Process ID` toe te voegen aan de *timestamp*.

6.3.2 Vector Clocks

Lamport's klokken zeggen niets over causaliteit, i.e. het is niet omdat $C(a) < C(b)$ dat *event* a voor b is gebeurt.



Figuur 6.8: Parallele bericht transmissie d.m.v. logische klokken

Dit wordt opgelost door vector klokken, hierbij zal $VC(a) < VC(b)$ garanderen dat *event* a voor b heeft plaatsgevonden.

Aanpak:
Elk proces P_i houdt een vector VC_i bij met de volgende eigenschappen:

1. $VC_i[i] =$ aantal *events* die al gebeurt zijn bij P_i. $VC_i[i]$ is dus de logische klok van P_i

2. Als $VC_i[j] = k$ dan weet P_i dat er al k *events* zijn gebeurt in P_j.

Stappen ondernomen om de vectoren aan te passen:

1. Voor het uitvoeren van een *event* verhoog de eigen counter met 1:
   $VC_i[i]$++
2. Zend het bericht met een *timestamp vector* VC_i
3. Bij het ontvangen van het bericht corrigeer de lokale klok van P_j:
   $VC_j[k] = max\{VC_j[k], timeStamp(m)[k]\}$ dit voor elke k

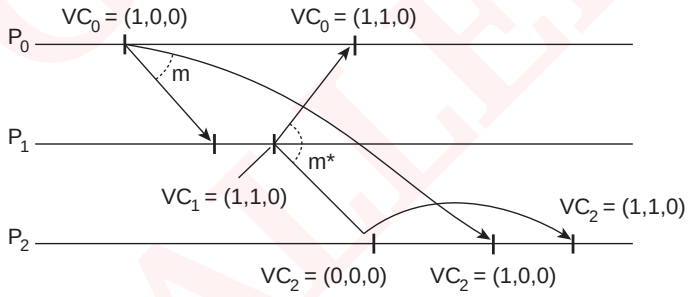
Enforcing Causal Communication

Vector klokken kunnen causale communicatie afdwingen, e.g. causally-ordered multicasting. Hierbij wordt er enkel rekening gehouden met gerelateerde berichten, dit in tegenstelling tot algoritme van Lamport. Ongerelateerde berichten kunnen dus worden ontvangen in verschillende volgorde bij de nodes.

Werking:
Als P_i een bericht verstuurd naar P_j dan wordt het bericht vertraagd in P_j (d.w.z. wachten tot het geven van het bericht aan de applicatie-laag):

- $ts(m)[i] == VC_j[i] + 1$
  m is het volgende bericht dat P_j verwachtte van P_i

  en
- $ts(m)[k] \leq VC_j[k]$ voor alle $k \neq i$
  P_j heeft alle berichten gezien dat ook werden gezien door P_i.



Figuur 6.9: Afdwingen van causale communicatie

6.4 Mutual exclusion

6.4.1 Overview

Token-based solutions

Een speciaal bericht tussen processen, i.e. de *token*, wordt rondgestuurd. Degene die de *token* in handen heeft zal toegang krijgen tot*shared resource*. Dit heeft echter als nadeel dat als een

machine crasht waarbij de *token* zat zal verloren geraken of zal moeten worden terug gehaald worden.

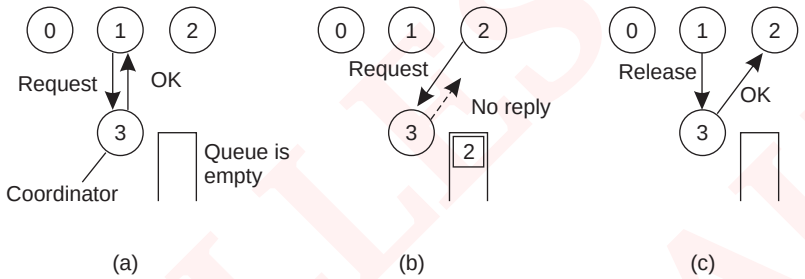
Voordelen zijn dan weer het vermijden van *starvation* en *deadlocks*.

Permission-based approaches

Er zal permissie moeten worden gevraagd aan de andere processen alvorens een *shared resource* te benaderen. In gedistribueerde systemen zal men vaker terug grijpen naar permission-based approaches, deze worden hieronder dus ook verder besproken onder de vorm van algoritmen.

6.4.2 A Centralized Algorithm

Er wordt één coördinator uitgekozen uit de processen, als een proces een resource wilt benaderen zla deze eerst toestemming moeten vragen aan de coördinator die deze requests FIFO behandeld.



Figuur 6.10: (a) Proces 1 vraagt de coordinator voor permissie om een shared resource te benaderen. Deze permissie is toegelaten. (b) Proces 2 zal ook permissie vragen. De coordinator antwoord niet. (c) Wanneer proces 1 de resource vrijlaat, zal de coordinator antwoorden op proces 2.

Eigenschappen zijn:

- ☺ Eerlijk
- ☺ Gemakkelijk te implementeren
- ☹ Moeilijk te recoveren van een crash
- ☹ Performance bottleneck

6.4.3 A Decentralized Algorithm

Hierbij zullen er N coördinators worden aangewezen voor elke resource, i.e. de resource zal N keer worden gerepliceerd. Het is de bedoeling om de permissie te krijgen van $M > \frac{N}{2}$ coördinators. Degenen die de resource wil benaderen zal telkens geïnformeerd worden over de beslissing van elke coördinator (YES/NO).

Eigenschappen zijn:

- ☺ Werkt correct met een zeer hoge probabiliteit
- ☺ Geen single-point-of-failure

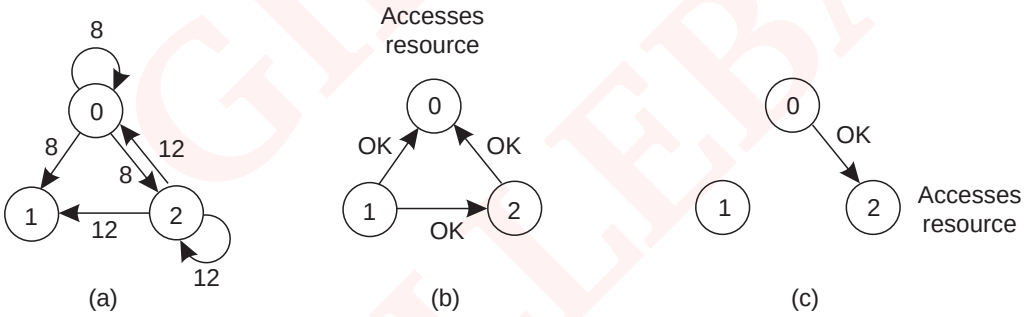
- ☹ Inefficiëntie als vele nodes een resource willen benaderen (geef dus de coördinators voorrang)
- ☹ Probabilistisch algoritme (*deadlock* is mogelijk)

6.4.4 A Distributed Algorithm

Wanneer een proces een gedeelde *resource* willen benaderen zalt het een bericht opstellen met de naam van *resource*, proces nummer en de momentele logische klok. Dit zend hij naar alle andere processen. De processen zijn dus zelf coördinators. Bij het ontvangen van een *request* bericht kunnen er drie verschillende *cases* worden onderscheiden:

1. return OK
   Als de ontvanger de *resouce* niet heeft of nodig zal hebben.
2. *Reqeust* in queue
   Als de ontvanger de resource al heeft.
3. Vergelijken van *timestamps*
   Als de ontvanger de *resource* ook wil benaderen (en dit niet al heeft gedaan) zal hij de *timestamps* vergelijken en zal op basis van die *timestamp case* 1 of 2 uitvoeren.

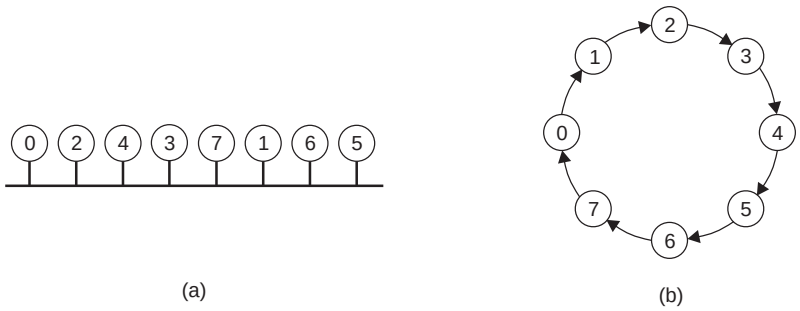
Het proces zal dus blijven wachten tot hij de permissie krijgt van alle andere processen. Als hij klaar is met die *resource* dan zend hij een ACK naar alle processen in zijn *queue* en wordt deze *queue* leeggemaakt.



Figuur 6.11: (a) Twee processen willen eenzelfde resource op hetzelfde moment. (b) Proces 0 heeft de laagste timestamp, en wint dus. (c) Wanneer proces 0 klaar is, zal hij een ACK verzenden naar 2 dat hij mag.

Als de *timestamps* dezelfde zijn, kan er nog altijd een onderscheid gemaakt worden via het processID.

6.4.5 A Token Ring Algorithm



Figuur 6.12: (a) Een groep van processen (zonder orde). (b) Een logische ring die geconstrueerd is in software.

Wanneer de ring wordt geintialiseerd dan krijgt proces 0 de *token*. De token blijft ronddraaien tot iemand de *resource* wil bemachtigen. Wanneer een node een resource wil benaderen zal hij moeten wachten tot hij de *token* overhandigd krijgt en zal deze pas vrijgeven na het vrijgeven van de *resource*.

Nadelen:

- ☹ Single point of failure
- ☹ Node failure, er moet dan een detectie worden gemaakt en een recovery om de kring terug rond te maken

6.4.6 Comparison of the Four Algorithms

Algorithm	Berichten per entry/exit	Delay bij elke entry (in message times)	Problems
Centralized	3	2	coordinator crash
Decentralized	2 m k + m, k = 1,2,...	2 m k	Starvation, low efficiency
Distributed	2 (n-1)	2 (n-1)	Crash of any process
Token ring	1 tot oneindig	1 tot (n-1)	Lost token, process crash

Tabel 6.1: Een vergelijking tussen vier mutual exclusion algoritmes

6.5 Election algorithms

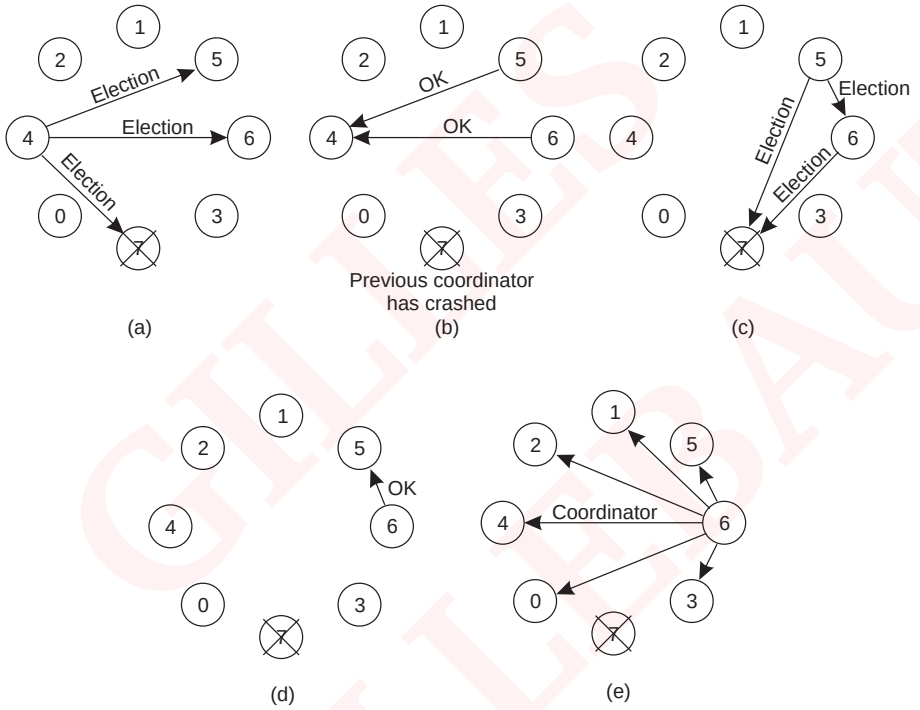
De verder besproken algoritmes hebben als doel een coördinator aan te stellen door verkiezingen te 'organiseren'. Dit is het proces met het hoogste `processID`.

Assumpties:

- 1 proces per machine
- Een uniek `processID` per proces
- Elke proces kent de ID van alle andere processen

6.5.1 The Bully Algorithm

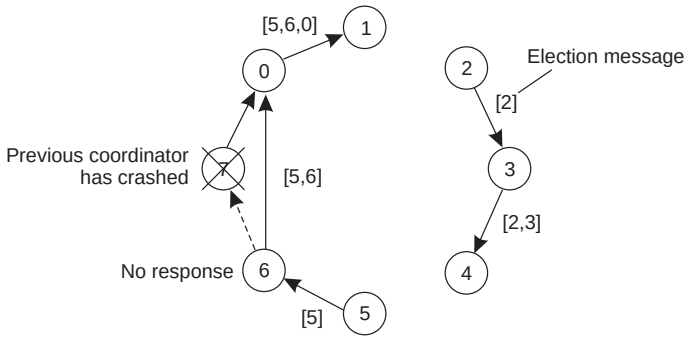
- P zend een `election` bericht naar alle processen met een hoger nummer.
- Als niemand antwoord dan zal P winnen en coördinator worden.
- Als iemand van de hogere antwoord, zal deze overnemen en is P's job over.



Figuur 6.13: De bully election algorithm. (a) Process 4 houdt een verkiezeing. (b) Proces 5 en 6 antwoorden, zeggen dat 4 moet stoppen. (c) Nu zal 5 en 6 een verkiezeing houden. (d) 6 en wint en vertelt het aan iedereen.

6.5.2 The Ring Algorithm

- Het `election` bericht gaat heel de ring rond vanaf de node die door had dat de coördinator was gecrasht
- Na de volledige toer zal een `coördinator` bericht worden verstuurd om iedereen op de hoogte te brengen van de nieuwe coördinator.



Figuur 6.14: Verkiezingsalgoritme met gebruik van een ring.

6.6 Summary

Strongly related to communication between processes is the issue of how processes in distributed systems synchronize. Synchronization is all about doing the right thing at the right time. A problem in distributed systems, and computer networks in general, is that there is no notion of a globally shared clock. In other words, processes on different machines have their own idea of what time it is.

There are various way to synchronize clocks in a distributed system, but all methods are essentially based on exchanging clock values, while taking into account the time it takes to send and receive messages. Variations in communication delays and the way those variations are dealt with, largely determine the accuracy of clock synchronization algorithms.

Related to these synchronization problems is positioning nodes in a geometric overlay. The basic idea is to assign each node coordinates from an rn-dimensional space such that the geometric distance can be used as an accurate measure for the latency between two nodes. The method of assigning coordinates strongly resembles the one applied in determining the location and time in GPS.

In many cases, knowing the absolute time is not necessary. What counts is that related events at different processes happen in the correct order. Lamport showed that by introducing a notion of logical clocks, it is possible for a collection of processes to reach global agreement on the correct ordering of events. In essence, each event e, such as sending or receiving a message, is assigned a globally unique logical timestamp C(e) such that when event a happened before b, $C(a) < C(b)$. Lamport timestamps can be extended to vector timestamps: if $C(a) < C(b)$, we even know that event a causally preceded b.

An important class of synchronization algorithms is that of distributed mutual exclusion. These algorithms ensure that in a distributed collection of processes, at most one process at a time has access to a shared resource. Distributed mutual exclusion can easily be achieved if we make use of a coordinator that keeps track of whose turn it is. Fully distributed algorithms also exist, but have the drawback that they are generally more susceptible to communication and process failures.

Synchronization between processes often requires that one process acts as a coordinator. In those cases where the coordinator is not fixed, it is necessary that processes in a distributed computation decide on who is going to be that coordinator. Such a decision is taken by means of election algorithms. Election algorithms are primarily used in cases where the coordinator can crash. However, they can also be applied for the selection of superpeers in peer-to-peer systems.

Hoofdstuk 7

Consistentie en replicatie

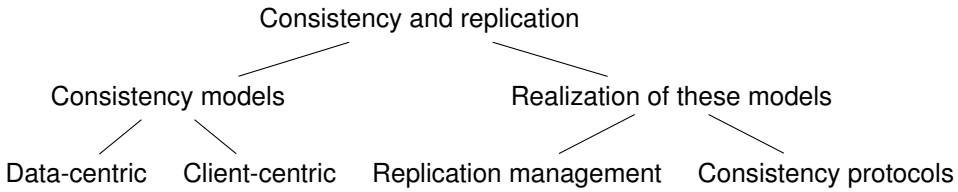
Inhoudsopgave

7.1 Inleiding

Data wordt gerepliceerd om de betrouwbaarheid en de performantie van een systeem te verhogen. Replicatie en/of caching kunnen gebruikt worden om de schaalbaarheid van een systeem te verhogen, onder de vorm van een verhoogd aantal van systemen of verschillende replica's geografisch te verspreiden.

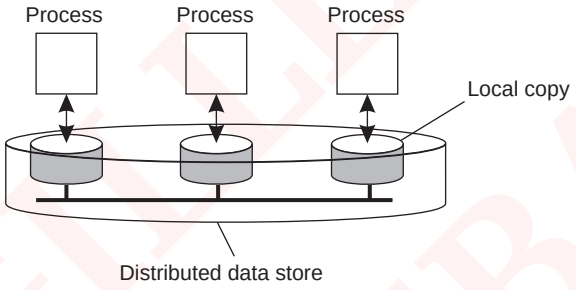
Dit is echter niet triviaal vermits er nood is aan een globale synchronisatie, wat gepaard gaat met een communicatie kost. De replica's moeten ook dezelfde data bevatten, wat een performance penalty met zich meebrengt. De oplossing hiervoor is het relaxeren van de consistentie beperkingen.

Een *consistency model* is een een contract tussen processen en de *data store* waarbij volledige consistentie niet mogelijk is.

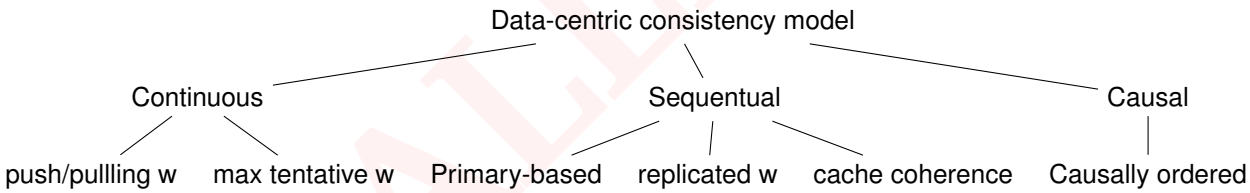


7.2 Data-centric consistency models

Problemen komen boven wanneer een proces van een databank de meest recente versie wilt lezen en bij het schrijven moeten de kopies propageren naar alle *data stores*.



Figuur 7.1: De algemene organisatie van een logische *data store*, fysisch gedistribueerd en gerepliceerd over meerdere processen.



7.2.1 Continuous Consistency

Afwijkingen binnen replica's mogen soms voorkomen:

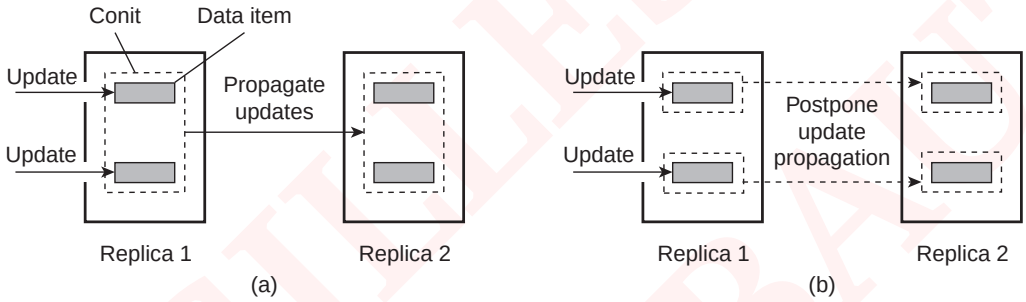
- Numerical deviations
  - Prijzen van twee kopieën mogen niet meer afwijken dan 2 cent.
  - Het aantal updates die niet nog niet gezien zijn door andere replica's mag niet groter zijn dan een bepaalde waarde.

- Staleness deviation
  Relateert met de laatste update tijd van een replica
  - Replica's moeten binnen 2 uur geüpdatet worden bij bijvoorbeeld weersvoorspellingen.
- Ordering deviation
  Update een replica zonder rekening te houden met een opgelegde volgorde. Hierbij moet er wel *rollback* en *reorder* mechanismes worden voorzien.

Conit

Een conit is een *unit* waarover consistentie wordt gemeten. Deze kunnen worden opgedeeld in *coarse-grained* en *fine-grained* categorieën.

Bij *coarse-grained* conits zullen de replica's sneller in een staat van inconsistentie zijn. Dit in tegenstelling tot *fine-grained* conits waarbij er wel meer info moet worden bijgehouden. Dit is weergegeven in figuur 7.2 waarbij de conits niet meer dan 1 update mogen verschillen. Hierdoor zullen *coarse-grained* conits sneller zorgen voor een update propagatie dan *fine-grained* conits.



Figuur 7.2: Keuze voor de gepaste granulariteit voor een conit. (a) Twee updates leiden tot een update propagatie. (b) Er is (tot nu toe) nog geen update propagatie nodig.

7.2.2 Consistent Ordering of Operations

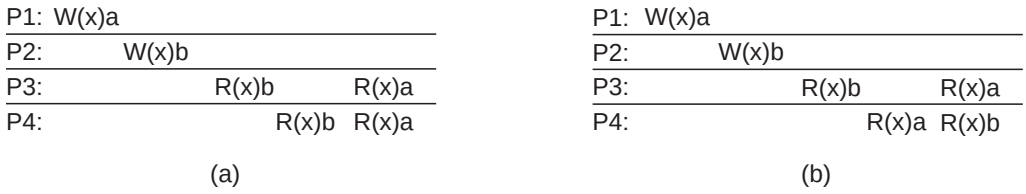
Het systeem zal hier een consistente volgorde moeten overeenkomen voor elke update.

Sequential Consistency

Een *data store* is sequentieel consistent als het aan de volgende conditie voldoet:

The results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program

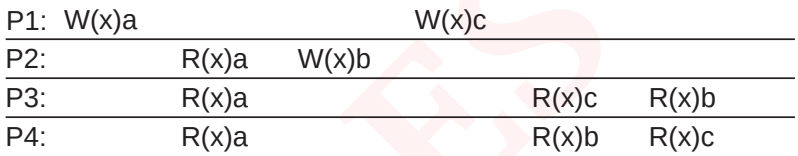
Wat eigenlijk neerkomt op het volgende; zolang elk proces in dezelfde volgorde de operaties ziet is het in orde, er wordt niets gezegd over de meest recente. Dus de gezien volgorde moet niet de werkelijke volgorde zijn.



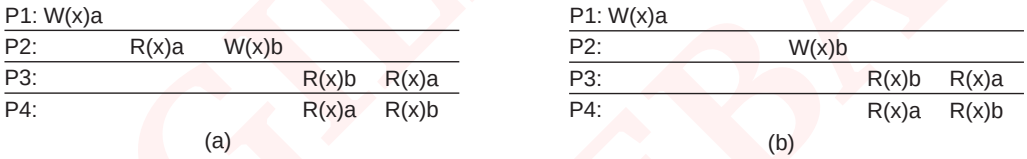
Figuur 7.3: (a) Een sequentieel consistente data store. (b) een data store die niet sequentieel consistent is.

Causal Consistency

Dit is een verzwakking van sequentiële consistentie, want hierbij moeten enkel gebeurtenissen die met elkaar in verband kunnen gebracht worden op elkaar volgen. Dus *writes* die mogelijks causaal gerelateerd zijn moeten in dezelfde volgorde gezien worden bij alle processen. Concurrente *writes* mogen dus wel in verschillende volgorde gezien worden.



Figuur 7.4: Deze sequentie mag bij een causaal-consistente store, maar niet bij een sequentieel consistente store.



Figuur 7.5: (a) Een schending van een causaal-consistente store. (b) Een correcte volgorde van gebeurtenissen in een causaal-consistente store.

Grouping Operations

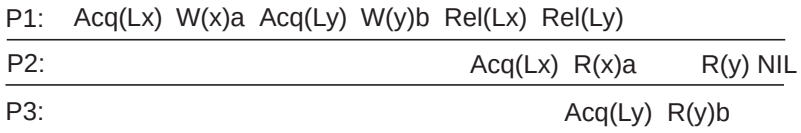
Op applicatieniveau is het vaak nodig dat lees en schrijf operaties beveiligd zijn tegen parallelle toegang. Hierdoor zullen we gebruik moeten maken van synchronisatie variabelen die worden genomen bij het intreden van een *critical section* en bij het verlaten.

Hierbij moet er aan drie criteria worden voldaan:

1. `Acq(l)` door een proces is niet toegelaten tot alle updates van de gedeelde data uit is uitgevoerd, i.e. manipuleer enkel up-to-datet data.

2. *Exclusive mode access* voor L is toegestaan als er geen enkel ander proces de synchronisatie variabele heeft, i.e. andere processen mogen de gedeelde data niet updaten.

3. *Nonexclusive mode access* voor L mag enkel uitgevoerd worden, rekeninghoudend met de eigenaar van de variabele, i.e. verkrijg eerst de meest recente versie van de data.



Figuur 7.6: Een correcte volgorde van gebeurtenissen voor entry consistency.

7.2.3 Korte intermezzo

Voor we over gaan naar client-centric consistency bespreken we nog even kort de voorgaande mechanismes.

Data-centric consistency is gelijkaardig aan concurrent programming en kent volgende mechanismes:

- Sequentiële consistentie is gebaseerd op het delen van hoofdgeheugen op een systeem
- Causale consistentie is op hetzelfde principe gebaseerd als Sequentiële consistentie, maar zal in de praktijk minder vaak worden gebruikt
- Grouping operaties zijn eerder een buitenstaander vermits die zijn gebaseerd op een set van instructies die 'atomair' moeten uitgevoerd worden

De client-centric consistency is dan eerder gebaseerd op minder parallelle schrijven. Deze kent ook twee mechanismes:

- Eventual consistency zal ervoor zorgen dat de client altijd dezelfde replica zal benaderen
- Dan bestaan er nog mechanismes die wel toelaten dat de client mobiel kan 'bewegen'

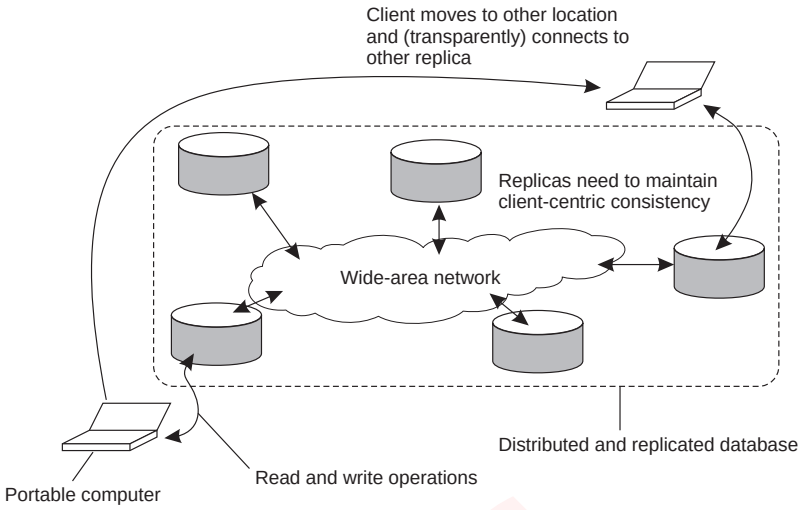
7.3 Client-centric consistency models

7.3.1 Eventual Consistency

Bij systemen waar er weinig data updates gebeuren kunnen we gebruik maken van updates in een *lazy fashion*. Zo zal één proces een kopie updaten en daarna deze update verder laten propageren. Er kunnen geen write-write conflicts optreden vermits enkel geautoriseerde processen hun data mogen bewerken. Enkelm read-write confilcts kunnen optreden, maar dit is eenvoudig op te lossen vermits het vaak toelaatbaar is dat een reader een oudere variant leest terwijl de data wordt geüpdatet.

Voorbeelden van zo systemen zijn DNS en caching van website aan de client kant.

Er is echter een probleem als gebruikers zich op verschillende replica's zal begeven en dat de verplaatsing gebeurt op een korte tijd. In dat geval kan de gebruiker een oudere waarde lezen op een andere replica. Dit probleem wordt geschetst in figuur 7.7 op de volgende pagina.



Figuur 7.7: Het principe van een mobiele gebruiker die verschillende replica's van een gedistribueerde databank probeert te benaderen.

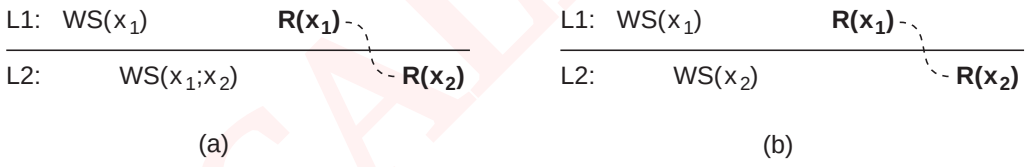
Om dit op te lossen gaan we over naar client-consistency modellen, die garanderen dat een enkele client een consistente benadering van data verkrijgen.

Er zullen hier verder vier client-centric consistency modellen bespreken:

1. Monotonic Reads (MR)
2. Monotonic Writes (MW)
3. Read Your Writes (RYW)
4. Writes follow Reads (WFR)

7.3.2 Monotonic Reads

Voorwaarde: *Als een proces een waarde van data item x leest, zal elke lees operatie erna op x van dat proces altijd dezelfde waarde of een nieuwere waarde teruggeven.*

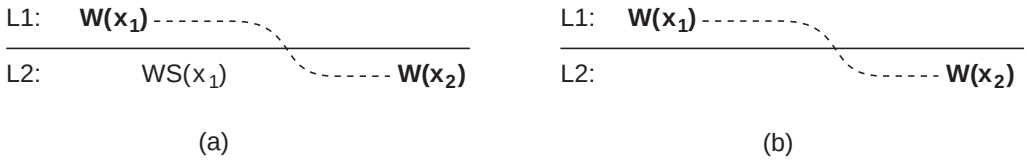


Figuur 7.8: De lees operatie uitgevoerd door een proces P op twee verschillende lokale kopies van dezelfde data store. (a) Een monotonic-read consistent data store. (b) Een data store die niet voldoet aan monotonic reads.

Een voorbeeld waar dit model zou inpassen is bij gerepliceerde mailboxen.

7.3.3 Monotonic Writes

Voorwaarde: *Een schrijfoperatie van een proces op een data item x, zal volledig uitgevoerd zijn voor er een opvolgende schrijf operatie op x is door datzelfde proces*

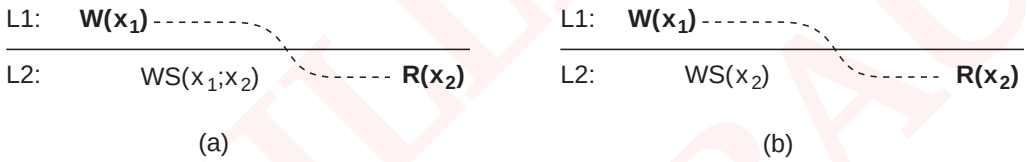


Figuur 7.9: De schrijf operatie uitgevoerd door een proces P op twee verschillende lokale kopies van dezelfde data store. (a) Een monotonic-write consistent data store. (b) Een data store die niet voldoet aan monotonic write, want update x_1 gebeurt niet.

Een voorbeeld waar dit model zou inpassen is bij het updaten van een SW library (version control).

7.3.4 Read Your Writes

Voorwaarde: *Het effect van een schrijf operatie door een proces op een data item x zal altijd worden gezien door opvolgende lees operaties op x door hetzelfde proces*

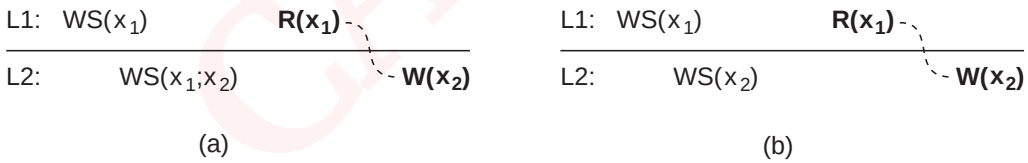


Figuur 7.10: (a) Een data store die read-your-writes consistentie ondersteunt. (b) Een data store die read-your-writes consistentie NIET ondersteunt

Een voorbeeld waar dit model zou inpassen is bij het updaten een paswoord of website.

7.3.5 Writes follow Reads

Voorwaarde: *Een schrijf operatie door een proces op een data uitem x gevolgd door een vorige lees operatie op diezelfde x door hetzelfde proces vindt gegarandeerd plaats op dezelfde of een meer recente versie van x dat werd gelezen.*



Figuur 7.11: (a) Een data store die writes-follow-reads consistentie ondersteunt. (b) Een data store die writes-follow-read consistentie NIET ondersteunt

Een voorbeeld waar dit model zou inpassen is bij reacties op een nieuwsgroep die enkel gepost wordt op een lokale kopie voor het originele bericht werd geschreven. Dus een reactie op een

nieuwspost wordt enkel maar gepropageerd of gelezen nadat op diezelfde replica de nieuwspost werd geschreven.

7.4 Consistency protocols

Consitency protocollen zijn de implementaties van consistentie modellen.

7.4.1 Protocols for continuous consistency

Deze werden niet uitvoerig besproken in de les, meer info kan er worden teruggevonden op pagina 306 in het handboek.

- Bounding Numerical Deviation: pushing nieuwe schrijf operaties door de bron
- Boudning Staleness Deviations: pulling nieuwe schrijf operaties door de replica op bepaalde tijdsintervallen
- Boudning Ordering Deviations

7.4.2 Protocols for sequential consistency

Primary-Based Protocols

Remote write protocol Bij het remote write protocol kan er gebruik gemaakt worden van blocking en non-blocking requests.

- Bij blocking requests zal de client moeten wachten tot de update is uitgevoerd op alle servers.
- In het geval van non-blocking, zal de client wachten tot de eerste `ack`
  - ☺ Wat de schrijf operaties versnelt
  - ☹ Er is nood aan fault tolerance (wat besproken wortd in het volgende hoofdstuk)

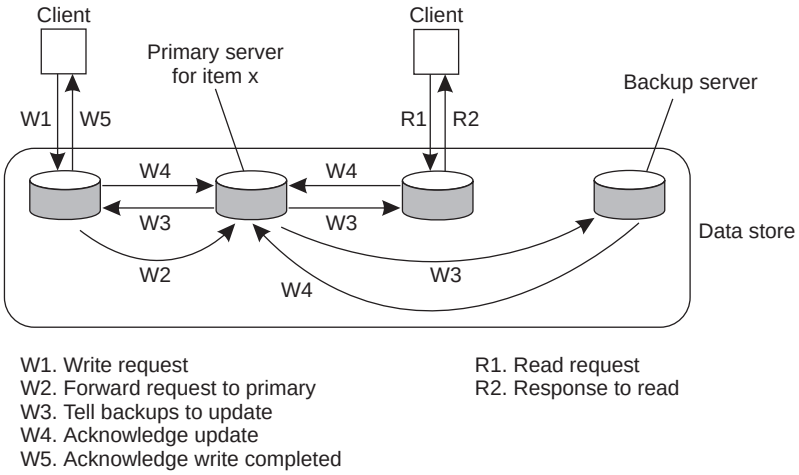
Local write protocol Dit protocol is zeer gelijkend op het voorgaande, hier zal enkel de *master* of *primary server* verplaatsen naar het proces die de update wil uitvoeren.

☺ Dit is goed voor een *burst* aan schrijf operaties op dezelfde kopie als het gata over non-blocking requests.

Dit kan in de praktijk gebruikt worden bij laptops die verbonden zijn en hun data delen. Wanneer de laptop zich loskoppelt zal hij de master moeten worden zodat hij toch lokaal alles kan updaten.

Replicated-write Protocols

Active Replication



Figuur 7.12: Het principe van een primary-backup protocol.

Unordered approach Lokaal proces zal zelf een update versturen naar alle servers. We zorgen dat de sequentie correct verloopt d.m.v. *totally ordered multicasting* (lamports klokken).

Ordered approach De client stuurt de update naar een *sequencer* en vraagt dus eigenlijk een volgnummer. De *sequencer* zal de nummer toevoegen en de update doorsturen naar de andere servers. Dit is eer gelijkend aan primary-based protocols.

Quorum-Based Protocols Werkt op het principe van 'verkiezingen'. Er moet permissie worden gevraagd aan de andere servers alvorens te schrijven of lezen. We stellen het lees quorum voor door N_R en het schrijf quorum door N_W.

De eigenschappen zijn dan:

1. $N_R + N_W > N$
2. $N_W > N/2$

Voor het lezen moeten er dus N_R servers akkoord gaan, het is analoog voor het schrijven.

De eerste voorwaarde zorgt ervoor dat er geen lees-schrijf conflicten voorkomen, de tweede voorwaarde zorgt dat er geen schrijf-schrijf conflicten voorkomen. In het eerste geval zou het anders kunnen voorkomen dat we lezen op een foute kopie.

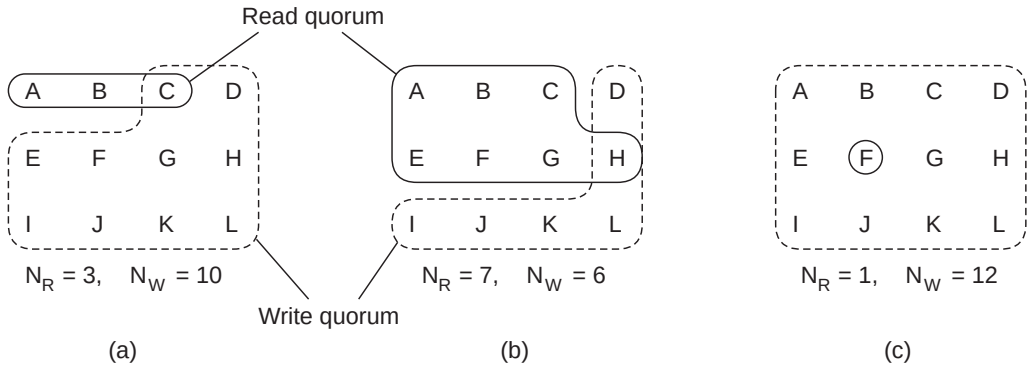
In figuur 7.13 op de pagina hierna wordt er geïllustreerd hoe het algoritme werkt.

Cache-Coherence Protocols

Controleer of de lokale kopie dezelfde is als deze op een server kopie op basis van aangewezen versie nummer van het data item.

Coherence detection strategies Er zijn drie strategieën mogelijk:

1. Doe de coherence check, vervolg met de transactie



Figuur 7.13: Drie voorbeelden van het voting algoritme. (a) Een correcte keuze van lees en schrijf operatie set. (b) Een keuze dat kan leiden tot schrijf-schrijf confilcten. (c) Een correcte keuze, ook wel ROWA genaamd.

2. Voer parallel de transactie en coherence check door, wat optimistisch is
3. Check pas als de transactie *commits* met de mogelijkheid dat er moeten worden gestopt

Coherence enforcements strategies Ook hier zijn drie mogelijkheden:

1. Verbied lokale caching, wat niet performant is
2. De server zend invalidaties naar de caches
3. De server propageert updates

Client interference Wat als de client de cached data veranderd?

- Write-through cache
  De client fungeert hier als tijdelijke primary met exclusieve schrijfpermissies, zie local-write protocol.
- Write-back cache
  Vertraag het aantal schrijfoperaties zodat er meerdere schrijf operaties mogelijk zijn alvorens de servers te informeren.

7.4.3 Implementing Client-Centric Consistency

De server zal hier een unieke identifier koppelen aan elke schrijf opdracht. De client moet zo een lees en schrijf set bijhouden. In de lees set zullen de relevante schrijfopdrachten staan voor de leesoperaties. In de schrijf set zullen de schrijfopdrachten die uitgevoerd zijn door de client bevatten.

Monotomic Reads

De client zal de lees set geven aan de server, de server zal de schrijfopdrachten in de lees set uitvoeren en antwoorden op de client.

Monotomic Writes

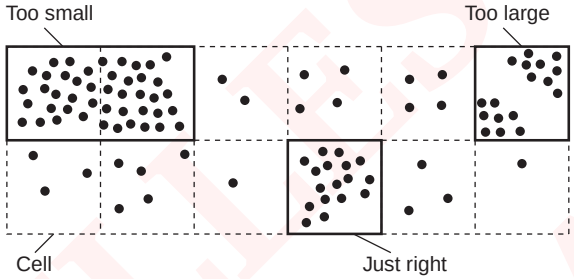
De client zal de schrijf set geven aan de server, de server zal de schrijfopdrachten in de schrijf set uitvoeren en zal de nieuwe schrijfopdracht uitvoeren.

7.5 Replica Management

Replica management houdt zich bezig met de vraag: Waar plaatsen we replica servers en welke data moeten ze bevatten?

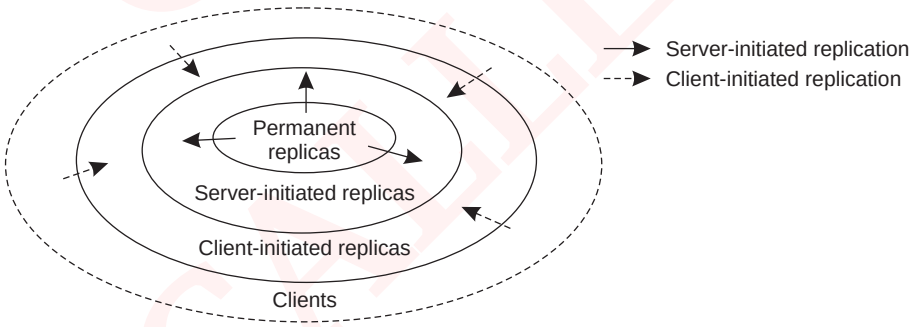
7.5.1 Replica-Server Placement

De keuze zal gebaseerd meoten worden op de afstand van de clients tot de locaties, waardoor we de ruimte moeten opsplitsen in cellen, zoals weergegeven op figuur 7.14.



Figuur 7.14: Het kiezen van een goede cel grootte voor server placement

7.5.2 Content Replication and Placement



Figuur 7.15: De logische organisatie van verschillende kopieën van een data store in drie concentrische ringen.

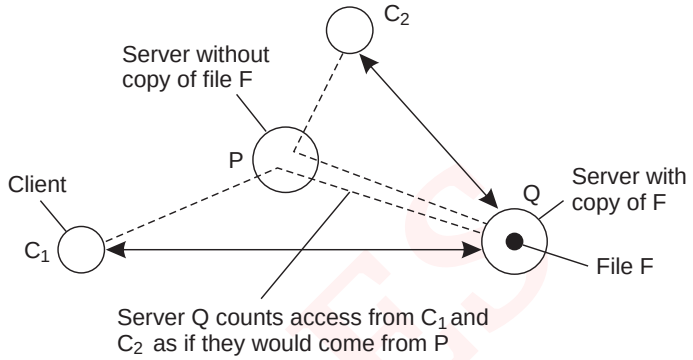
Permanent Replicas

- De eerste set van replica's zullen bestaan uit verschillende replica servers die vast staan op een vaste locatie

- Er kan ook worden gedaan aan mirroring die een kopie is van een server die geografisch verspreid liggen

Server-Initiated Replicas

Als er een *burst*is van inkomende requests op een bepaalde plaats moet er een replica kunnen worden toegevoegd. Dit wordt bijvoorbeeld gebruik bij web hosting services en wordt geïllustreerd op figuur **??** op pagina ??.



Figuur 7.16: Het tellen van access requests van verschillende clients.

Elke server Q houdt een teller bij van die weergeeft hoeveel keer de server Q werd gecontacteerd voor een bestand F door een client dichtbij server P, `count_Q(P,F)`.

We kunnen dan tresholds definiëren voor het verwijderen en voor de replicatie van een bestand op een server, respectievelijk `Del(S,F)` en `Rep(S,F)`.

Het algorimte gaat dan als volgt:

- Als `count(S,F) < del(S,F)` verwijder kopie, tenzij er maar één meer over is.
- Als `del(S,F) < count(S,F) < rep(S,F)` migreer het bestand
- Als `count(S,F)` $\geq$ `rep(S,F)` repliceer het bestand

Client-Initiated Replicas

Zijn eigenlijk client caches wat de access time bevorderd. Deze caches kunnen zich lokaal bevinden of op een machine in hetzelfde LAN.

7.5.3 Content Distribution

State versus Operations

Er zijn drie ook hier drie strategieën mogelijk:

1. Propageer enkel een notificatie van een update, wat neerkomt op een invalidation protocol. Dit is voordelig als er veel schrijfopdrachten zijn in vergelijking met leesopdrachten en bespaart bandbreedte.

2. Transfereer de data van één kopie naar de andere, wat voordelig is als de read-to-write ratio hoog is.

3. Propageer de update operatie naar de andere kopies, wat neerkomt op een active replication. Dit is voordelig als de data groot is en de parameters van de operatie klein. Het nadeel is echter dat de processing moet gebeuren op elke replica.

Pull versus Push Protocols

Server-based protocols gebruiken een push-based approach die invalidaties of data kunnen pushen tussen verschillende servers.

Client-based protocols gebruiken eerder een pull-based approach die de response time verhogen in het geval van een cache mis.

7.6 Summary

There are primarily two reasons for replicating data: improving the reliability of a distributed system and improving performance. Replication introduces a consistency problem: whenever a replica is updated, that replica becomes different from the others. To keep replicas consistent, we need to propagate updates in such a way that temporary inconsistencies are not noticed. Unfortunately, doing so may severely degrade performance, especially in large-scale distributed systems.

The only solution to this problem is to relax consistency somewhat. Different consistency models exist. For continuous consistency, the goal is set to bounds to numerical deviation between replicas, staleness deviation, and deviations in the ordering of operations.

Numerical deviation refers to the value by which replicas may be different. This type of deviation is highly application dependent, but can, for example, be used in replication of stocks. Staleness deviation refers to the time by which a replica is still considered to be consistent, despite that updates may have taken place some time ago. Staleness deviation is often used for Web caches. Finally, ordering deviation refers to the maximum number of tentative writes that may be outstanding at any server without having synchronized with the other replica servers.

Consistent ordering of operations has since long formed the basis for many consistency models. Many variations exist, but only a few seem to prevail among application developers. Sequential consistency essentially provides the semantics that programmers expect in concurrent programming: all write operations are seen by everyone in the same order. Less used, but still relevant, is causal consistency, which reflects that operations that are potentially dependent on each other are carried out in the order of that dependency.

Weaker consistency models consider series of read and write operations. In particular, they assume that each series is appropriately ”bracketed”by accompanying operations on synchronization variables, such as locks. Although this requires explicit effort from programmers, these models are generally easier to implement in an efficient way than, for example, pure sequential consistency.

As opposed to these data-centric models, researchers in the field of distributed databases for mobile users have defined a number of client-centric consistency models. Such models do not consider the fact that data may be shared by several users, but instead, concentrate on the consistency that an individual client should be offered. The underlying assumption is that a client connects to

different replicas in the course of time, but that such differences should be made transparent. In essence, client-centric consistency models ensure that whenever a client connects to a new replica, that replica is brought up to date with the data that had been manipulated by that client before, and which may possibly reside at other replica sites.

To propagate updates, different techniques can be applied. A distinction needs to be made concerning what is exactly propagated, to where updates are propagated, and by whom propagation is initiated. We can decide to propagate notifications, operations, or state. Likewise, not every replica always needs to be updated immediately. Which replica is updated at which time depends on the distribution protocol. Finally, a choice can be made whether updates are pushed to other replicas, or that a replica pulls in updates from another replica.

Consistency protocols describe specific implementations of consistency models. With respect to sequential consistency and its variants, a distinction can be made between primary-based protocols and replicated-write protocols. In primary-based protocols, all update operations are forwarded to a primary copy that subsequently ensures the update is properly ordered and forwarded. In replicated-write protocols, an update is forwarded to several replicas at the same time. In that case, correctly ordering operations often becomes more difficult.

Hoofdstuk 8

Fout tolerantie

Inhoudsopgave

8.1 Introduction to fault tolerance

8.1.1 Basic concepts

Er zijn wat vereisten voor afhankelijke systemen:

- Availability
  Dit is de kans dat het systeem op een correcte manier werkt op elk moment. We spreken dan over grote ordes van interrupties van 1 seconde per dag.

- Reliability
  Dit is de kans dat het systeem op een correcte manier werkt op een groot interval. We spreken in dit geval in termen van tijdsintervallen in tegenstelling tot tijdseenheden. bijvoorbeeld: een interrupt van twee weken om het jaar.
- Safety
  Spreekt over de situatie waarbij dat het tijdelijk uitvallen van het systeem niet resulteert in catastrofale zaken.
- Maintainability
  Hoe gemakkelijk kan het systeem worden onderhouden/hersteld.

Types of Faults

- Transient faults
  Deze fouten komen eenmalig voor en verdwijnen dan, als de operatie opnieuw wordt gestart dan zal de fout weggaan. Voorbeeld: een vogel die een transmittor beam doorkruist, hierdoor zullen er bits verloren gaan, maar deze kunnen herzonden worden.
- Intermittent fault
  Dit zijn fouten die herhaaldelijk voorkomen en vanzelf terug verdwijnen. Bijvoorbeeld: een los contact bij een connector.
- Permanent fault
  Zijn fouten die in het systeem blijven tot de fout-component vervangen is. Bijvoorbeeld: een doorgebrande chip, software bugs,. . .

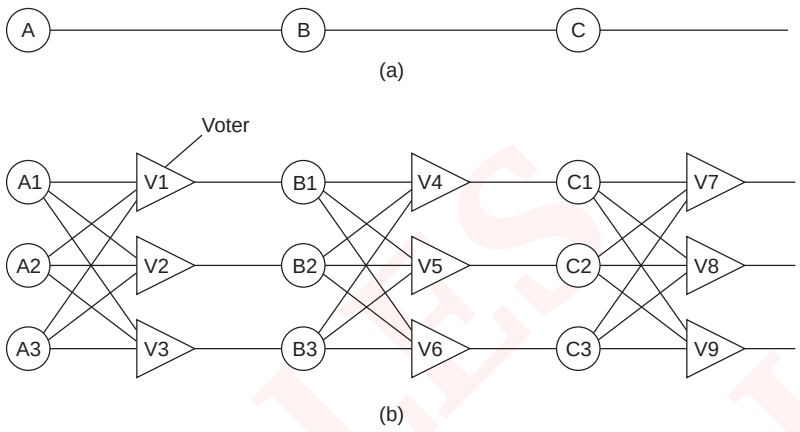
8.1.2 Failure Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

8.1.3 Failure Masking by Redundancy

Als we redundancy inbouwen in het systeem kunnen we fouten maskeren. We kunnen een onderscheidt maken tussen verschillende soorten redundancy:

- Information redundancy
  f.i. Hamming codes
- Time redundancy
  f.i. berichten herverzenden
- Physical redundancy
  f.i. HW of SW componenten repliceren, dit is weergegeven in figuur 8.1, i.e. TMR (Triple Modular Redundancy)



Figuur 8.1: Triple modular redundancy – Een mechanisme om fysische fout-componenten te maskeren. Zowel de *voters* als de elementen kunnen fouten bevatten, daarom dat er gebruik wordt gemaakt van drie *voters* per stage.

8.2 Process resilience

8.2.1 Design Issues

We kunnen dus, zoals al eerder gezegd, fouten vermijden door te doen aan *grouping*.

Flat Groups versus Hierarchical Groups

Flat groups

- Symmetrisch

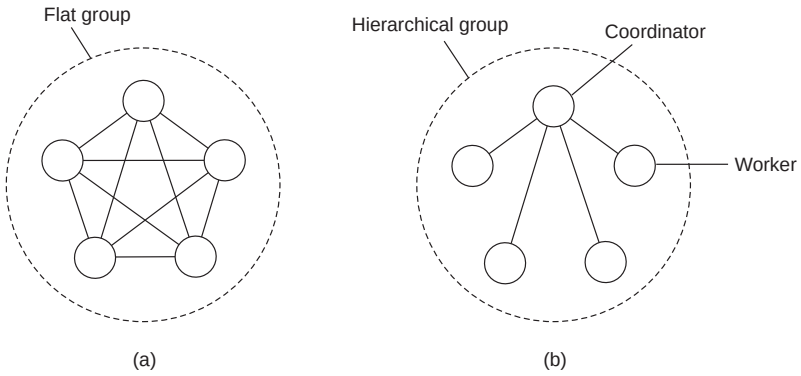
☺ Geen Single-point-of-failure

☹ Complexe coördinatie

Hierarchical groups

☺ Gemakkelijkere coördinatie

☹ Single-point-of-failure



Figuur 8.2: (a) Communicatie in een flat group. (b) Communicatie in een simpele hiërarchie.

Group Membership

Om groepen te onderhouden (e.g. members toelaten, groepen aanmaken) kunnen we gebruik maken van twee mechanismes: enerzijds een gecentraliseerde en anderzijds een gedistribueerde aanpak.

- Gecentraliseerde aanpak
  Maakt gebruik van een group server die de exacte membership tabellen bijhoudt. Er is wel weer S-P-o-F mogelijk.
- Gedistribueerde group membership algoritmes
  Zijn meer complex. De kandidaten kunnen zich aanmelden om lid te worden van de groep via een multicast bericht naar de groep te sturen.

8.2.2 Failure Masking and Replication

We kunnen een vulnerable process maskeren door deze in een fout tolerante groep te steken. Deze zijn gerepliceerde processen, en kunnen op twee manieren worden gerepliceerd: via primary-based of replicated-write protocollen.

- Primary-based protocollen, worden ook wel eens primary backop protocollen genoemd en werken op basis vaneen electie algoritme wanneer de primary server crasht.
- Replicated-write protocollen, zijn actieve replicaties en zijn gebaseerd op Quorom based protocollen, wat werkt op basis van het vragen van permissies aan andere processen.

How much replication is needed?

Als we spreken over een k fout tolerant systeem, spreken we over een systeem die kan 'overleven' met k falende componenten.

Als we kijken naar *silent failures* dan moeten we $k+1$ componenten hebben. Want er kunnen dan k componenten stoppen en er kan toch een output geleverd worden.

Als we willen beveiligen op byzantijnse *failures* dan moeten we $2 \cdot k + 1$ componenten voorzien. Hierbij spreken we over k zieke componenten die foute outputs geven. Als er k componenten foute

info geeft kunnen we de correcte info eruit halen door de $k+1$ overige componenten, er is dus een meerderheid aan correcte info.

8.2.3 Failure Detection

Een *failure* kan worden gevonden a.d.h.v. een pull of push-based approach. In het eerste geval, *pinging* zal er worden gepolled met de vraag of je nog `alive?` bent. In het laatste geval, *gossiping*, zal er worden verkondigd dat je nog steeds 'leeft'.

We kunnen zowel *failures* hebben bij nodes als op het netwerk zelf.

8.3 Reliable client-server communication

8.3.1 Point-to-Point Communication

Door het gebruik van retransmissies en acks is het mogelijk om hier de *omossion failures* te maskeren.

Het crashen zal echter niet kunnen gemaskeerd worden. Dit kan enkel maar gemaskeerd worden door een nieuwe connectie op te starten met een andere server in het gedistribueerd systeem.

8.3.2 RPC Semantics in the Presence of Failures

We bespreken eerst vijf fouten die kunnen voorkomen in RPC systemen, om daarna apart de verschillende problemen uit te diepen en oplossingen te bieden.

De vijf *failures*:

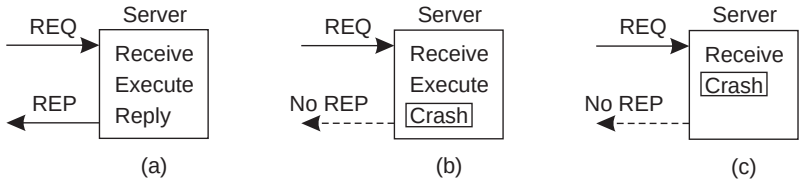
1. De client kan de server niet bereiken
2. Het request bericht van de client naar de server is niet toegekomen
3. De server crasht achter het ontvangen van een request
4. Het reply bericht van de server naar de client is niet toegekomen
5. De client crast na het evrzenden van een request

Client Cannot Locate the Server

We kunnen exceptions toevoegen aan het client proces zodat de client weet wanneer hij de serverniet kan bereieken. We wouden wel dat *remote procedure* hetzelfde oogde als een lokale, hier moeten we dan inboeten.

Lost Request Messages

De server kan hier een `ACK` sturen, en een timer bijgehouden op de client die dan een bericht kan terug zenden bij een time-out.



Figuur 8.3: Een server in een client-server communicatie. (a) De normale situatie. (b) Crash achter uitvoering. (c) Crash voor uitvoering.

Server Crashes

We kunnen geen semantiek vinden die voor zowel (b) als (c) een oplossing biedt in figuur 8.3.
Er zijn drie events mogelijk:

1. $M \rightarrow$ verzenden van *complete* bericht
2. $P \rightarrow$ tekst printen
3. $C \rightarrow$ crash

Deze events kunnen gebeuren in zes verschillende volgordes:

1. $M \rightarrow P \rightarrow C$
2. $M \rightarrow C(\rightarrow P)$
3. $P \rightarrow M \rightarrow C$
4. $P \rightarrow C(\rightarrow M)$
5. $C(\rightarrow P \rightarrow M)$
6. $C(\rightarrow M \rightarrow P)$

Client	Server					
	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

Figuur 8.4: Verschillende combinaties van client en server strategieën bij server crashes.

Er bestaat dus geen combinatie van strategieën die correct zal werken voor alle sequentie van events.

Lost Reply Messages

Dit is geen probleem voor idempotent berichten, i.e. berichten die zonder consequenties meerdere keren mogen uitgevoerd worden.

Het is wel een probleem bij geldtransfers.

We kunnen dit oplossen door de requests in een idempotente manier te structurenen. De server slaat dan staat informatie op en de client zend het verloren request bericht terug.

Client Crashes

Wat leidt tot *orphans*, i.e. *remote procedures* zonder *parent*. Mogelijke strategieën zijn:

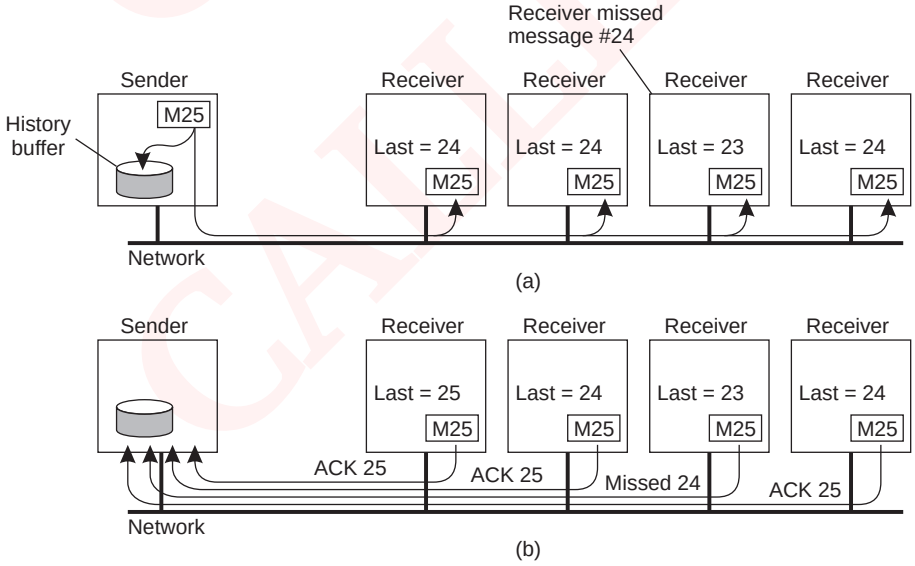
1. Extermination, *orphans* verwijderen achter reboot
2. Reincarnation, starten van een nieuwe epoch achter reboot
3. Expiration, laten vervallen achter een bepaalde tijd

8.4 Reliable group communication

8.4.1 Basic Reliable-Multicasting Schemes

Het is de bedoeling om alle berichten bij de members te krijgen.

Een oplossing:
Gebruik van een *history buffer* bij de zender.
Elk bericht krijgt een sequentienummer en wordt lokaal opgeslagen in die buffer tot iedereen heeft ge-ACK'ed. Als blijkt dat een ontvanger de vorige berichten nog niet heeft ontvangen zal hij deze vragen aan de zender.



Figuur 8.5: Een simpele oplossing voor betrouwbare multicast waarbij alle ontvangers gekend zijn en geacht niet te crashen. (a) Bericht transmissie. (b) Reporting feedback.

In de praktijk zullen we echter te maken hebben met bursts van berichten en kunnen dan de transimissie verlagen door het *piggyback*en van ACKs.

8.4.2 Scalability in Reliable-Multicasting

De zender wordt overspeoeld met ACKs als er veel ontvangers zijn, want voor N ontvangers zullen er N ACKS worden verstuurd.

De basisch oplossing zou zijn dat we enkel negatieve ACKs versturen, i.e. als het bericht niet is toegekomen. Wat resulteert in een grote *history buffer* bij de zender.

Het doel is eigenlijk de feedback berichten te reduceren, hiervoor bestaan er twee strategieën: nonhierarchical feedback en hierarchical feedback control.

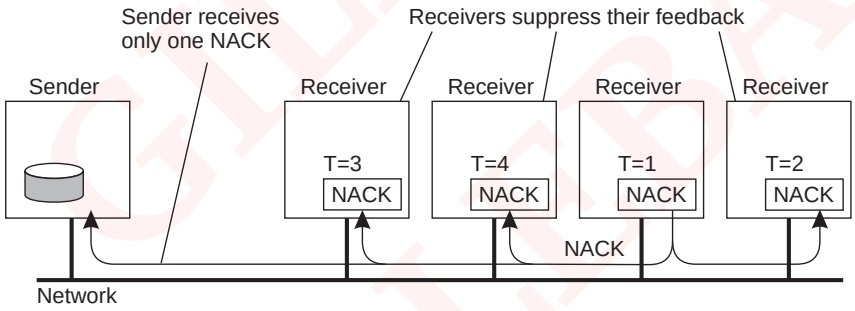
Nonhierarchical Feedback Control

Dit biedt een oplossing bij het flooden van de zender door NACKs. De strategie gaat als volgt:

- Elke ontvanger wacht een random tijd om feedback te sturen
- de NACK wordt dan gemulticast naar de rest van de processen in de groep

Door het random wachten zullen andere ontvangende processen gestopt worden om ook hun feedback te sturen wat resulteert in het niet overspoelen van de verzender.

Het nadeel is echter wel dat alle andere ontvangers een NACK krijgen.



Figuur 8.6: Verschillende ontvangers hebben een request voor transmissie gescheduled, maar de eerste retransmissie request zal zorgen dat de rest wordt onderdrukt.

Hierarchical Feedback Control

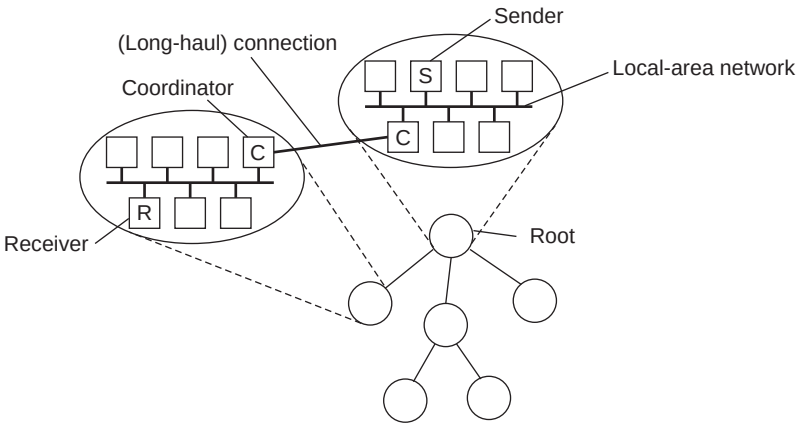
In deze aanpak zal elke lokale coördinator zijn eigen *history buffer* hebben.

Dus we splitsen een groter netwerk op in kleinere groepen waarbij betrouwbare multicast voor kleine groepen wordt gebruikt.

8.4.3 Atomic Multicast

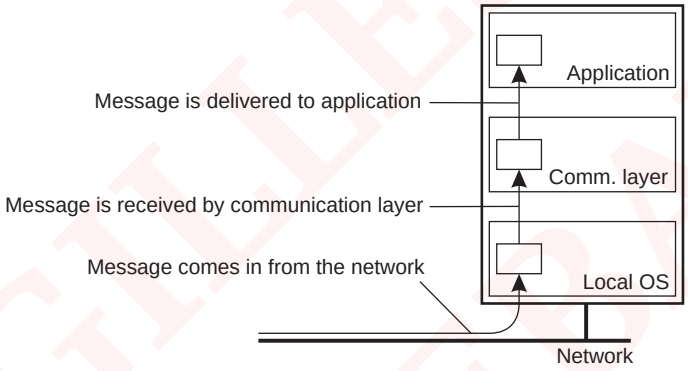
Hierbij worden alle berichten afgeleverd aan alle processen en in dezelfde volgorde of ze worden helemaal voor niemand afgeleverd.

Een voorbeeld is het up-to-date houden van gerepliceerde databanken.



Figuur 8.7: De essentie van hiërarchische betrouwbare multicasting. elke lokale coördinator stuurt het bericht door naar zijn kinderen en zal later de requests retransmissie afhandelen.

Virtual Synchrony



Figuur 8.8: De logische organisatie van een gedistribueerd systeem om een onderscheidt te kunnen maken tussen het ontvangen van een bericht en het bezorgen van een bericht (aan de applicatie-laag). [Totally ordered Multicasting]

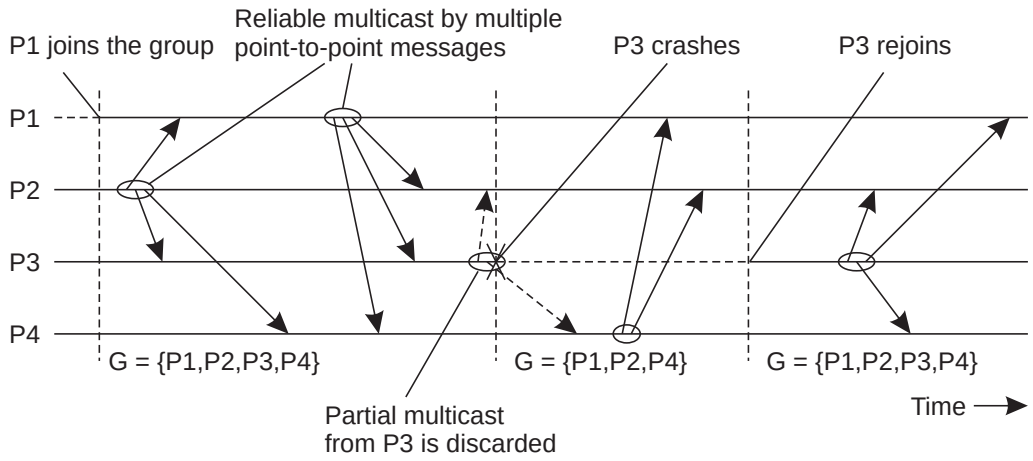
Als de zender crasht tijdens de multicast dan zullen de berichten ofwel ontvangen worden door alle processen of worden genegeerd door iedereen.

De multicasts vinden plaats in epochs, dit is een tijdsinterval met een set van members, zoals weergegeven op figuur 8.9 op de volgende pagina.

Message Ordering

Zender:

1. Unordered multicast
2. FIFO ordered multicast
3. Causally ordered multicast, geïmplementeerd m.b.v. vector timestamps
4. Totally-ordered multicast, ontvangen bij alle processen in dezelfde orde



Figuur 8.9: Het principe van virtuele synchrone multicast.

Ontvanger:
Atomic multicast:

1. Conditie 1 : Totally-ordered delivery, maakt niet uit hoe ze zijn ontvangen als ze maar allemaal in dezelfde manier worden afgeleverd aan de applicatie-laag.
2. Conditie 2: Virtual synchrony

Unordered multicast Hier maakt het niet uit in welke volgorde de berichten worden ontvangen.

Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

Figuur 8.10: Drie communicerende processen uit dezelfde groep die communiceren in een ongeordende manier.

FIFO ordered multicast We zien dat de ontvangen berichten tussen P2 en P3 verschillend zijn, wat mag. Enkel moeten de volgordes van versturen door 1 proces dezelfde zijn. Dus moeten bij beiden m1 voor m2 komen en m3 voor m4, zolang hier aan voldaan is, dan geldt FIFO-ordered multicasting.

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

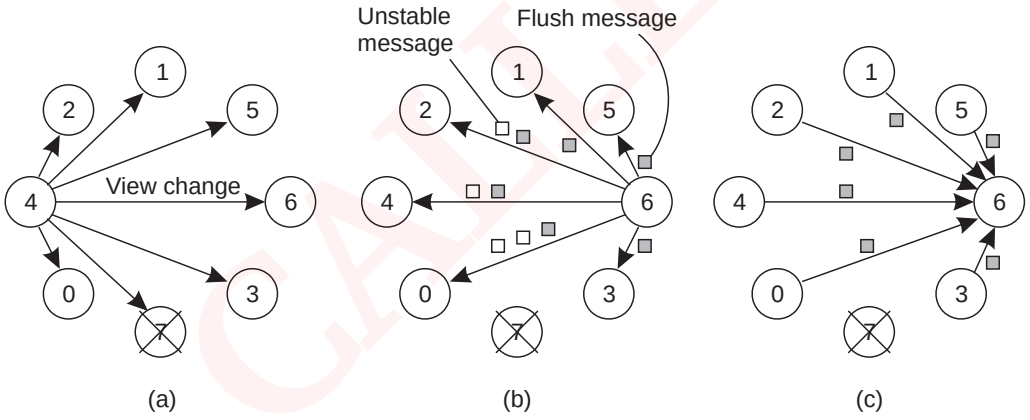
Figuur 8.11: Vier processen uit eenzelfde groep, met twee verschillende zenders die ontvangstorder uitvoeren m.b.v. FIFO-ordered multicasting.

Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Figuur 8.12: Zes verschillende versies van virtuele synchrone betrouwbare multicasting.

Implementing Virtual Synchrony

Een stabiel bericht is een bericht dat door alle members van een view G is gezien. Enkel deze, stabiele berichten, mogen doorgegeven worden aan de applicatie-laag.



Figuur 8.13: (a) Proces 4 ziet dat proces 7 is uitgevallen en zend een *view change*. (b) Proces 6 zend al zijn onstabiele berichten, gevolgd door een *flush message*. (c) Proces 6 installeert een nieuwe *view* wanneer het een flush bericht ontvangt van alle andere.

8.5 Distributed commit

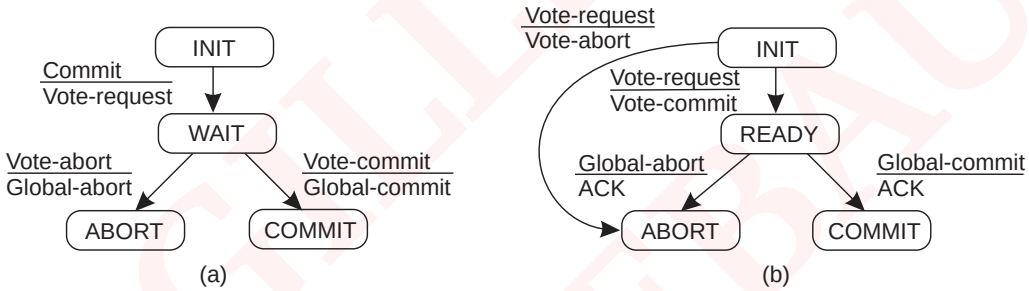
Een gedistribueerde commits is een operatie dat door iedereen of door niemand wordt uitgevoerd (binnen één groep).

8.5.1 One-Phase commit

De coördinator zegt aan alle members dat de operatie moet worden uitgevoerd. Het probleem hierbij is dat als een proces die operatie niet kan uitvoeren, dat de coördinator dit niet kan weten.

8.5.2 Two-Phase commit

- Coordinator sends `VOTE_REQUEST` to all participants
- Participant returns `VOTE_COMMIT` or `VOTE_ABORT`
- Coordinator collects votes, and sends `GLOBAL_COMMIT` or `GLOBAL_ABORT`
- Participant locally commits or locally aborts



Figuur 8.14: (a) Het finite state machine voor de coordinator in 2PC. (b) De finite state machine voor een deelnemer.

Het probleem hierbij is dat als een proces P aan het wachten is in `READY` en de coördinator crasht, dan zal hij de andere processen moeten vragen in welke staat zij verkeren. Als iedereen in de `READY` staat zit, dan weten ze niet of de coördinator al heeft beslist en wat hij heeft beslist.

Maar vermits deze situatie niet snel zal voorkomen blijft men bij de two-phase in tegenstelling tot de zeer complexe three-phase, die we dus hier niet gaan bespreken.

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Figuur 8.15: Acties die kunnen worden ondernomen door deelnemer P wanneer deze zich in de READY toestand bevindt en een ander proces Q zijn toestand vraagt.

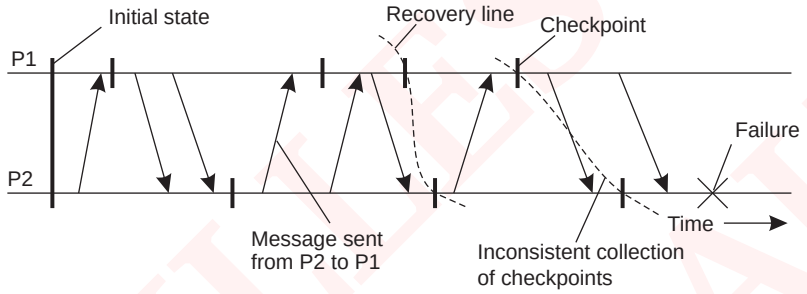
8.6 Recovery

Er bestaan twee manieren op te recoveren:

1. Backward recovery, waarbij we de vorige (de meest recente) staat terugbrengen a.d.h.v. checkpoints. De staat wordt dus opgenomen op bepaalde tijdstippen, en de verzonden pakketten worden dan opnieuw verzonden.
2. Forward recovery, zal tot een nieuwe correcte staat leiden. Dus we kwamen in een foute staat terecht en nu zal er alles worden gedaan om terug in een correcte staat te geraken, door verschillende operaties.

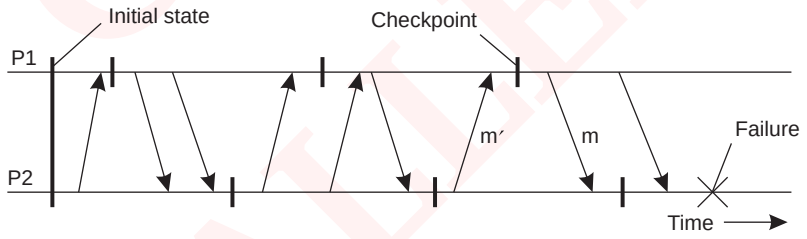
In he eerste geval worden dus gedistribueerde snapshots genomen, dit is een opname van een consistente globale staat die bestaat uit een collectie van checkpoints.

De recovery line is meest recente snapshot, wat dus slaat op de meest recente consistente collectie van checkpoints, wat weergegeven is in figuur 8.16.



Figuur 8.16: Een recovery line

als we onafhankelijk checkpoints beginnen zetten kan dit leiden tot een domino-effect als we terug willen gaan naar een vorige staat. Dit wordt geïllustreerd in figuur 8.17.



Figuur 8.17: Het domino-effect.

Als we over gaan naar gecoördineerde checkpointing dan zullen dus alle processen gesynchroniseerd hun staat lokaal gaan opslaan. Hierdoor krijgen we een globale consistente staat. Maar dit zal weer een coördinator vergen en dus overhead introduceren.

8.7 Summary

Fault tolerance is an important subject in distributed systems design. Fault tolerance is defined as the characteristic by which a system can mask the occurrence and recovery from failures. In other words, a system is fault tolerant if it can continue to operate in the presence of failures.

Several types of failures exist. A crash failure occurs when a process simply halts. An omission failure occurs when a process does not respond to incoming requests. When a process responds too soon or too late to a request, it is said to exhibit a timing failure. Responding to an incoming request, but in the wrong way, is an example of a response failure. The most difficult failures to handle are those by which a process exhibits any kind of failure, called arbitrary or Byzantine failures.

Redundancy is the key technique needed to achieve fault tolerance. When applied to processes, the notion of process groups becomes important. A process group consists of a number of processes that closely cooperate to provide a service. In fault-tolerant process groups, one or more processes can fail without affecting the availability of the service the group implements. Often, it is necessary that communication within the group be highly reliable, and adheres to stringent ordering and atomicity properties in order to achieve fault tolerance.

Reliable group communication, also called reliable multicasting, comes in different forms. As long as groups are relatively small, it turns out that implementing reliability is feasible. However, as soon as very large groups need to be supported, scalability of reliable multicasting becomes problematic. The key issue in achieving scalability is to reduce the number of feedback messages by which receivers report the (un)successful receipt of a multicasted message.

Matters become worse when atomicity is to be provided. In atomic multicast protocols, it is essential that each group member have the same view concerning to which members a multicasted message has been delivered. Atomic multicasting can be precisely formulated in terms of a virtual synchronous execution model. In essence, this model introduces boundaries between which group membership does not change and which messages are reliably transmitted. A message can never cross a boundary.

Group membership changes are an example where each process needs to agree on the same list of members. Such agreement can be reached by means of a commit protocol, of which the two-phase commit protocol is the most widely applied. In a two-phase commit protocol, a coordinator first checks whether all processes agree to perform the same operation (i.e., whether they all agree to commit), and in a second round, multicasts the outcome of that poll. A threephase commit protocol is used to handle the crash of the coordinator without having to block all processes to reach agreement until the coordinator recovers.

Recovery in fault-tolerant systems is invariably achieved by checkpointing the state of the system on a regular basis. Checkpointing is completely distributed. Unfortunately, taking a checkpoint is an expensive operation. To improve performance, many distributed systems combine checkpointing with message logging. By logging the communication between processes, it becomes possible to replay the execution of the system after a crash has occurred.

Hoofdstuk 10

Gedistribueerde Object Gebaseerde systemen

Clients verkrijgen diensten en resources door objecten aan te roepen.

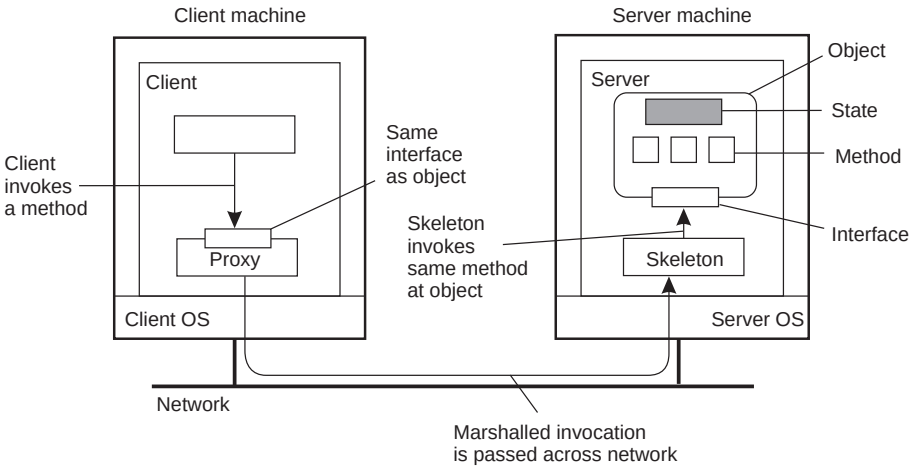
Inhoudsopgave

10.1 Architecture

10.1.1 Distributed Objects

Een object encapsuleerd data, de staat van het object,en de operaties op die data, de methodes. De methodes zijn openbaar gemaakt door een interface.

Het is zo mogelijk om de interface op een machine te plaatsen, maar dat het object zelf op een andere machine bestaat, wat weergegeven is in figuur 10.1 op de pagina hierna. Dit wordt een *distributed object* genoemd.



Figuur 10.1: Een organisatie van een *remote object* met een client-side proxy.

Net zoals bij RPC systemen zullen stubs zorgen dat de berichten (requests en replies) resulteren in methode operoepen of antwoorden. In dit geval spraken we bij clients over een proxy, die het objects interface bevat, en bij de server over een skeleton. De proxy en skeleton zijn dus analoog aan respectievelijk de client- en server-stub bij RPC systemen.

De *remote objects* geven enkel hun interface ter beschikking, de staat wordt echter niet gedeeld.

Compile-Time versus Runtime Objects

Compile-time objects In dit geval zijn objecten instanties van klassen, zoals in Java, C++, ... Het is mogelijk om de interface van die klassen in de client- en server-stub te compileren. Dit is hoe RMI werkt.

Het nadeel echter is dat deze methode programmeertaalafhankelijk is.

Runtime objects Er wordt gebruik gemaakt objecten die in verschillende programmeertalen kunnen geschreven zijn. *Distributed objects* worden zo expliciet tijdens runtime gemaakt. Een *object adapter* kan fungeren als een *wrapper* rond de implementatie om het een uitzicht te geven van een object. Een object wordt zo gedefinieerd als een interface die ze implementeren.

Persistent and Transient Runtime Objects

Persistent objects Deze objecten kunnen opgeslagen en ingeladen worden, i.e. hun levensduur is groter dan de leeftijdsduur van de server. Een voorbeeld voor dit soort objecten zijn objecten die de winkelwagen of stock regelen van een online-shop.

Transient objects Wanneer objecten 'sterven' met hun server dan spreken we over *transient objects*.

10.1.2 Example: Enterprise Java Beans

De JBE taal voorziet een runtime voor gedistribueerde objecten te ondersteunen. Een EJB is een Java object die wordt gehost door een speciale server die clients de mogelijkheid geeft om het object op verschillende manieren te benaderen. De server ondersteunt ook de separatie tussen de applicatie en de systeem-georiënteerde functionaliteiten.

Applicatie-georiënteerd Deze functies worden geïmplementeerd door de EJB, ook wel een `bean` genoemd.

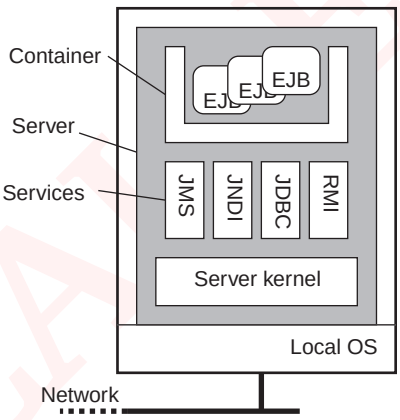
Systeem-georiënteerd De functies bestaan uit het opzoeken van objecten (JNDI), opslag van objecten (JDBC), communicatie: remote method invocation (RMI) en messaging (JMS), ...

De EJBs worden ingesloten door een container die een interface biedt naar onderliggende diensten die zijn geïmplementeerd in de applicatie server. Deze diensten zijn de functies die hierboven zijn beschreven bij de systeem-georiënteerde functionaliteiten.

EJB types

De programmeer moet een onderscheidt maken tussen vier verschillende types van beans, om optimaal gebruik te kunnen maken van de onderliggende functionaliteiten.

1. Stateless session beans
2. Stateful session beans
3. Entity beans
4. Message-driven beans



Figuur 10.2: Algemene architectuur van een EJB server.

Stateless session beans zijn *transient objects* die eenmalig worden benaderd, waarna ze worden vernietigd als hun taak erop zit. Er wordt telkens een nieuwe instantie –bij elke methode invocatie– gecreëerd. Er zal in de praktijk echter gewerkt worden met een pool van herbruikbare objecten. Een voorbeeld is het behandelen van een SQL query.

Stateful session beans zullen echter wel een client-state onthouden. Een elektronische shoppingcart bijvoorbeeld, waarbij de klant zaken kan toevoege, verwijderen om nadien de aankopen te bevestigen. Wanneer de klant klaar is zal het object worden vernietigd.

Entity beans zijn persistente objecten. Deze objecten zullen vaak opgeslagen zitten in databanken en zullen bestaan tijdens transacties.

Message-driven beans reageren op berichten volgens het publish-subscribe patroon. Deze beans worden niet rechtreeks aangesproken maar zijn gebonden met specifies soorten berichten. Na het afhandelen van de berichten worden ze vernietigd en zijn zo dus ook stateless.

10.2 Processes

10.2.1 Object Servers

Object servers zijn speciaal ontworpen om gedistribueerde objecten te ondersteunen. Deze servers bieden zelf geen diensten aan, de diensten bevinden zich in de objecten. Het is eenvoudig om diensten te wijzigen door gewoon objecten toe te voegen of te verwijderen.

Alternatives for Invoking Objects

Om het op een object aan te roepen maakt men best gebruik van verschillende *policies*, zo'n aanpak is flexibel.

Beslissingen hoe een objecten te benaderen worden *activation policies* genoemd.

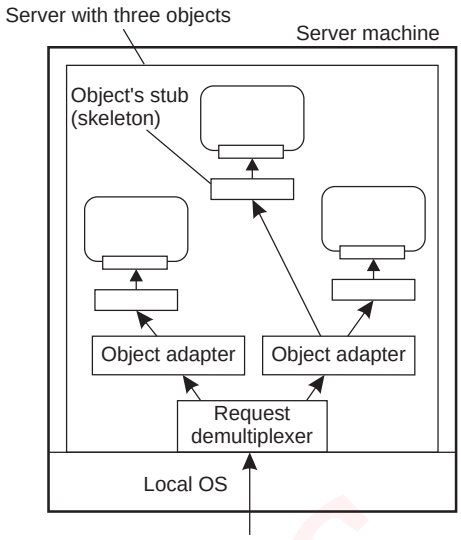
Enkele activatie policies worden hieronder besproken:

- Transiënte objecten worden opgeslagen in het RAM en worden vernietigd als de clients niet meer verbonden zijn met deze objecten. Het voordeel is dat de objecten geen resources innemen als ze niet nodig zijn. Dit kan echter ook nadelig zijn wanneer er veel methode invocaties zijn op dat object, in dat geval blijven deze objecten best in het RAM.
- Objecten kunnen zich bevinden in een aparte segmenten. Zo delen objecten geen code en data. Dit kan nodig zijn voor veiligheidsredenen.
- Er kan gewerkt worden met één server thread, die de controle regelt. In het andere geval kan men werken met een thread voor elk van zijn objecten. Waardoor er automatisch beveiligd is tegen synchronisatie problemen. Als men zou werken met een thread per invocatie zal dit de concurrency verhogen.

Object Adapter

Een object adapter of *object wrapper* zal objecten groeperen met eenzelfde policy.

Wanneer een invocation request arriveert zal de server de request eers dispatchen naar de gepaste object adapter, wat is weergegeven in figuur 10.3 op de volgende pagina.



Figuur 10.3: Een organisatie van een object server met verschillende activatie policies.

Deze adapters kennen de interfaces van hun objecten niet, anders zouden ze niet generisch kunnen zijn.

De adapter zal de invocation request naar de server-side stub van het object sturen. Die dan deze request unmarshalt en de methode uitvoert.

10.3 Communication

10.3.1 Binding a Client to an Object

Het is mogelijk om impliciet of expliciet een client te binden aan een object. Zo ontstaat er een globale referentie naar een gedistribueerd object over het gehele systeem.

Implementation of Object References

Een referentie moet uiteraard genoeg informatie bevatten om een client te kunnen binden met een object. Een referentie kan het volgende omvatten:

- Netwerk adres van de machine
- Poortnummer
- Indicatie over welk object het gaat

Een gedeelte van deze informatie zal worden geleverd door een object adapter.

Er zijn echter twee grote nadelen verbonden aan dit mechanisme. Ten eerste zal de server niet op een andere poort kunnen draaien en ten tweede zal de server ook nooit op een andere machine kunnen draaien.

Net zoals bij DCE waar we een client met een server konden binden zullen we dit mechanisme ook gebruiken om een client met een object te binden.

Een daemon zal het beheer regelen op de machines zodat de server wel achter een andere poort kan draaien.

Een *location server* zal de locaties van de gedistribueerde objecten bijhouden, elke referentie bevat zo het adres van de locatie server en een globale *identifier* voor die server.

10.3.2 Static versus Dynamic Remote Method Invocations

In tegenstelling tot RPC zal RMI object-specifieke stubs gebruiken. Om te genieten van RMI ondersteuning zal er gebruik moeten gemaakt worden van een IDL.

```
object.method(input, output)
```

Static invocation

De object interfaces kunnen worden gegenereerd op basis van de gekende IDL. Hierdoor zal de client moeten worden ge-hercompileerd als de interface van het object wijzigt.

```
invoke(object, method, input, output)
```

Dynamic invocation

Een alternatief is *dynamic invoation*, waarbij een methode invocatie *at runtime* wordt uitgevoerd. Dit kan bijvoorbeeld uitgelezen worden uit een configuratie bestand.

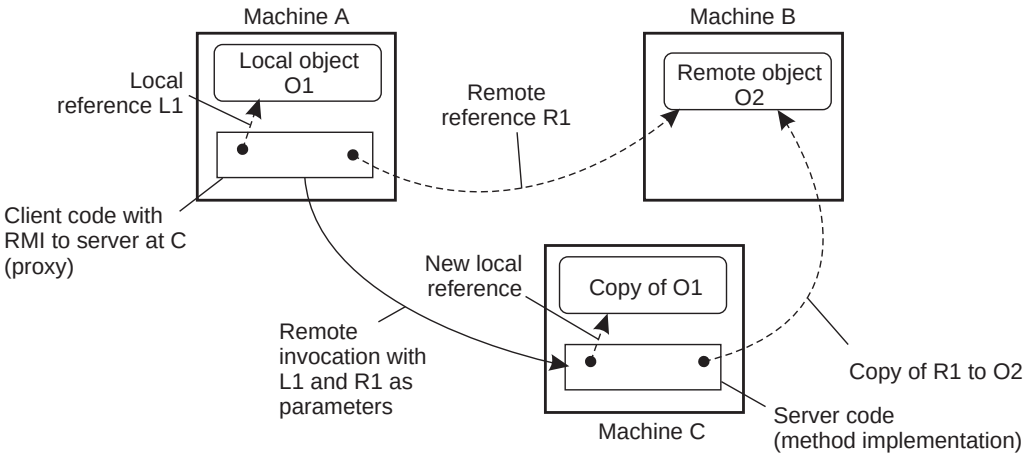
10.3.3 Parameter Passing

Er kunnen twee mogelijke parameters worden doorgegeven, namelijk een referentie naar een *remote* of naar een *locale* object.

In het geval van een *remote* object zal de parameter worden doorgegeven als een `call-by-reference`.

In het andere geval, waar het object zich in de adresruimte van de machine bevindt, zal het object volledig worden gekopieerd en worden mee verzonden met de invocatie, d.i. `call-by-value`.

Deze twee situaties zijn weergegeven op figuur 10.4 op de pagina hierna.



Figuur 10.4: De situatie wanneer er een object wordt doorgegeven als referentie of als waarde.

10.3.4 Example: Java RMI

In Java zijn gedistribueerde objecten geïntegreerd in de programmeertaal.

The Java Distributed-Object Model

Objecten kunnen enkel gekopieerd worden door de server. Aan de client kant zullen de *proxies* dus nooit worden gekopieerd.

Java Remote Object Invocation

Enkel objecten die *serializable* zijn kunnen worden verstuurd via RMI. Typisch zullen platform afhankelijke objecten zoals *file descriptors* en *sockets* niet *serializable* zijn.

Parameter passing Zoals eerder besproken en te zien op figuur 10.4, worden lokale objecten verzonden `by-value` en *remote* objecten `by-reference`.

De referentie van zo'n *remote object* zal bestaan uit een adres, end-point van de server en een *locale identifier* voor het object.

Proxy In Java zijn *proxies serializable*. Zo is het mogelijk om een *proxy* door te sturen naar andere processen. Een proxy kan zo gebruikt worden als een referentie naar een *remote* object.

Implementation handles worden gecreëerd om code te kunnen doorgeven. Zo is het mogelijk om de proxy code apart te downloaden (een andere server host de proxy) waardoor het *unmarshalling* van de code zelf veel efficiënter gaat. De staat van het object is echter wel nog ge*marshalled*.

10.3.5 Object-Based Messaging

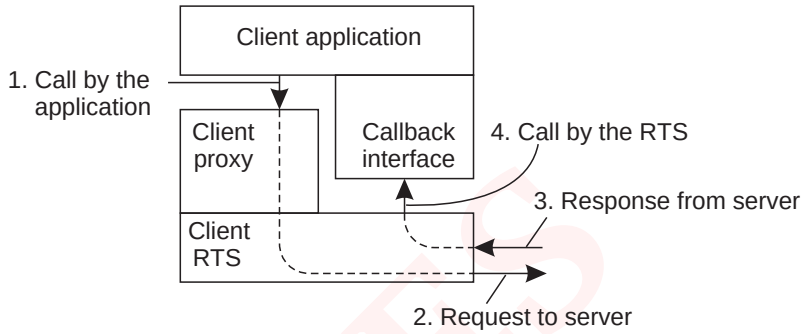
CORBA combineert methode invocatie met *message-oriented* communicatie. Een asynchrone methode invocatie is analoog aan een asynchrone RPC. CORBA ondersteunt twee modellen die de asynchrone methode invocatie uitvoeren: het callback model en het polling model.

CORBA's Callback Model

In het callback model, weergegeven in figuur 10.5, zal een client een object implementeren die die callback methodes bevat.

Deze methoden zullen door de onderliggende communicatielaag worden gebruikt om het resultaat van een asynchrone invocatie door te geven.

De server ziet dit als een synchrone invocatie, maar de client zorgt er expliciet voor dat die wordt omgevormd tot een asynchrone invocatie.



Figuur 10.5: CORBA's callback model voor asynchrone methode invocatie.

Er kan bijvoorbeeld gebruik worden gemaakt van een methode:
`int add(in int i, in int j, out in k)`
Deze synchrone methode invocatie zal moeten omgezet worden naar een asynchrone methode invocatie door de methode te splitsen:

```
void sendcb_add(in int i, in int j); //Downcall by client

void replycb_add(in int ret_val, in int k); //Upcall to the client
```

CORBA's Polling Model

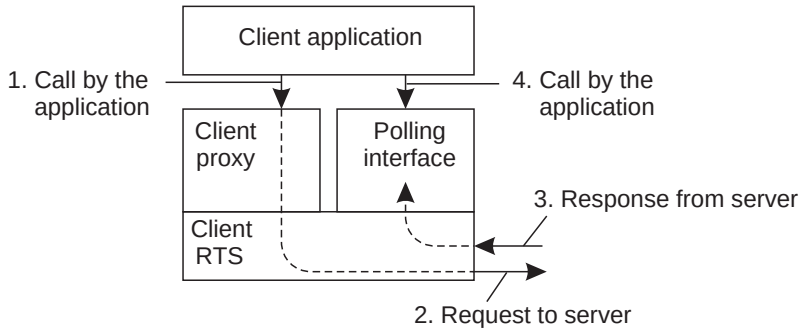
In het polling model zal de client de mogelijkheid hebben om zijn locale RTS voor inkomende resultaten te pollen.

Als we het voorbeeld van hierboven bekijken, resulteert de opsplitsing in deze twee methodes:

```
void sendpoll_add(in int i, in int j); //Called by client

void replypoll_add(in int ret_val, in int k); //Also called by the
client
```

De `replypoll_add` methode zal hier moeten worden geïmplementeerd door de client's RTS.



Figuur 10.6: CORBA's polling model voor asynchrone methode invocatie.

10.4 Summary

Most object-based distributed systems use a remote-object model in which an object is hosted by server that allows remote clients to do method invocations. In many cases, these objects will be constructed at runtime, effectively meaning that their state, and possibly also code is loaded into an object server when a client does a remote invocation.

To support distributed objects, it is important to separate functionality from extra-functional properties such as fault tolerance or scalability. To this end, advanced object servers have been developed for hosting objects. An object server provides many services to basic objects, including facilities for storing objects, or to ensure serialization of incoming requests. Another important role is providing the illusion to the outside world that a collection of data and procedures operating on that data correspond to the concept of an object. This role is implemented by means of object adapters.

When it comes to communication, the prevalent way to invoke an object is by means of a remote method invocation (RMI), which is very similar to an RPC. An important difference is that distributed objects generally provide a systemwide object reference, allowing a process to access an object from any machine. Global object reference solve many of the parameter-passing problems that hinder access transparency of RPCs.

There are many different ways in which these object references can be implemented, ranging from simple passive data structures describing precisely where a remote object can be contacted, to portable code that need simply be invoked by a client. The latter approach is now commonly adopted for Java RMI.

Hoofdstuk 12

Gedistribueerde Web gebaseerde systemen

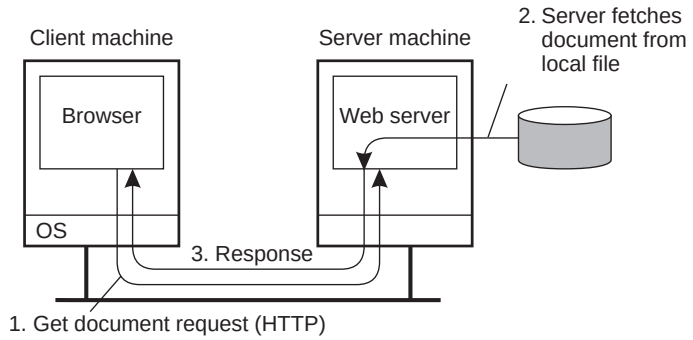
Inhoudsopgave

12.1 Architecture

Het idee van gedistribueerde documenten groeide uit tot het ondersteunen van dynamische documenten met actieve elementen, om uiteindelijk te komen tot *services* die worden aangeboden,i.e. *web services.*

12.1.1 Traditional Web-based Systems

De gemakkelijkste manier om naar een document te refereren is door een Uniform Resource Locater (URL).



Figuur 12.1: De organisatie van een traditionele Web site.

Web Documents

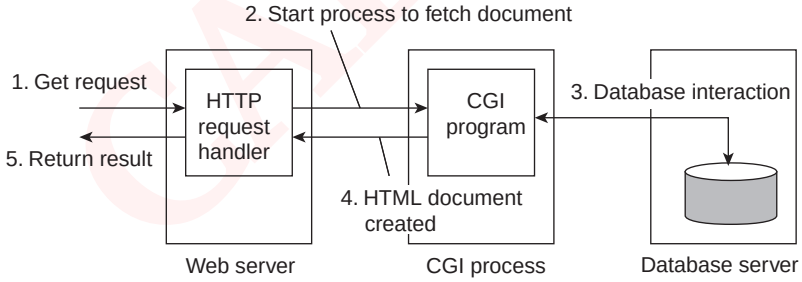
Op het Web wordt alle informatie opgeslagen onder de vorm van een document. Dit document kan worden opgedeeld in twee delen: een hoofdgedeelte die zorgt voor een template voor het tweede gedeelte, die bestaat uit verschillende zaken die samen het document vormen die wordt voorgesteld in de browser.

- Het hoofdgedeelte staat geschreven in een markup language
  Waarbij HyperText Markup Language (HTML) de meest gebruikte is. Een ander belangrijkemarkup language is eXtensible Markup Language (XML).
- Het tweede gedeelte bestaat uit *embedded documents*
  Er moet geweten zijn wat het type is en hoe de browser die data, van dat type, moet behandelen. Elk (*embedded*) document heeft een geassocieerde Multipurpose Internet Mail Exchange (MIME) type.

  Het applicatie type kan worden uitgevoerd door een apart programma of door een plugin die de browser uitbreidt.

Multitiered Architectures

Eén van de eerste verbetringen aan de basis architectuur was de ondersteuining voor een simpele *user interaction* door de Common Gateway Interface (CGI).



Figuur 12.2: Het principe van een server-side CGI programma.

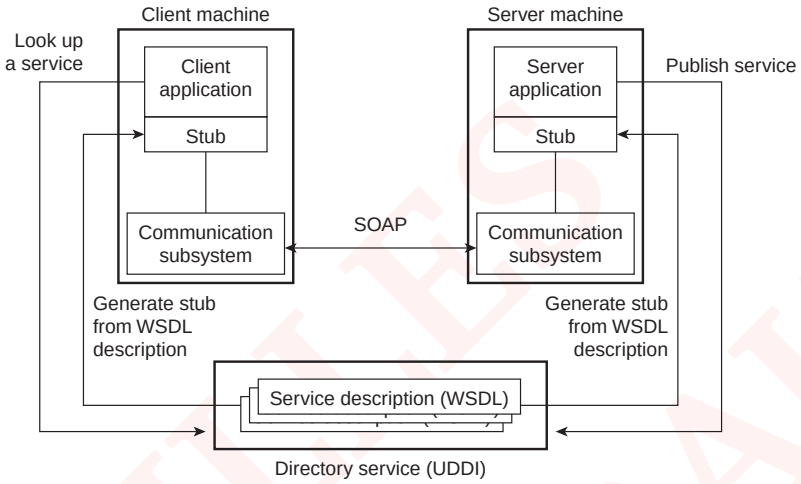
De gebruiker stuurt de CGI-naam en parameters door, hierdoor wordt de CGI script uitgevoerd op de server en wordt er een document terug gestuurd.

12.1.2 Web services

We gingen hiervoor uit van browsers die zich gedragen als een interface voor gebruikers. Maar er is een grote groep van Web-based systemen die services bieden aan *remote applications* die geen interactie met gebruikers vereisen aan het opkomen. Dit leidt tot de organisatie van Web services.

Web Services Fundamentals

Een Web service moet toegankelijk zijn voor clients door het gebruik van een standaard om de service op te zoeken en te gebruiken. Dit is weergegeven in figuur 12.3.



Figuur 12.3: Het principe van een Web service.

De Universal Description, Discovery and Integration standaard (UDDI) beschrijft de layout van een databank die *service descriptions* bevat, die Web service clients toelaten om relevante services te zoeken.

De services zelf worden dan beschreven door de Web Services Definition Language (WSDL).

De communicatie wordt verzorgd door het SOAP die een framework is die de communicatie tussen twee processen standaardiseert.

Web Services Composition and Coordination

Composition Composite services, e.g. een web-based shop, zijn complexe services die meerdere diensten aanbieden a.d.h.v. transacties. Zo'n shop zal bestaan uit meerdere *providers*: de order, payment en delivery service.

Coordination Coördinatie tussen Web services worden geregeld door coordination protocols.

Er kan geopteerd worden voor *single coordination* of *gedistribueerde coordination*.

Deze zaken zijn gestandaardiseerd in een Web Service Coordination.

12.2 Processes

12.2.1 Web Clients

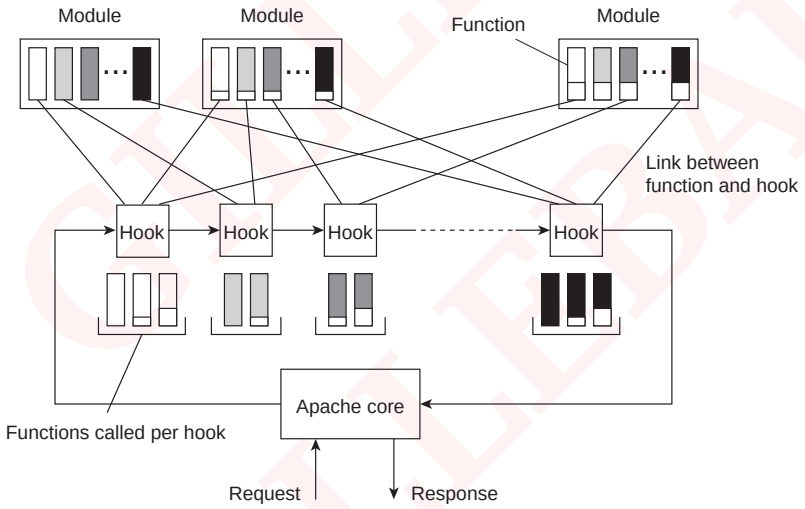
De belangrijkste Web client is een Web browser. Webbrowsers kunnen worden aangevuld met plugins die specifieke document types (MIME) kan behandelen. Deze plugins worden dynamisch in de browser geladen.

Een andere client-side process is een Web proxy. Een voorbeeld hiervoor is een FTP proxy. Om een bestand te versturen naar een FTP server, zal de browser een HTTP request sturen naar een lokale FTP proxy die het bestand ophaalt en teruggeeft onder de vorm van HTTP.

12.2.2 The Apache Web server

De Apache Web server is platform onafhankelijk door een basis runtime environment te voorzien die is geïmplementeerd voor elk platform. Deze runtime noemt Apache Portable Runtime (APR).

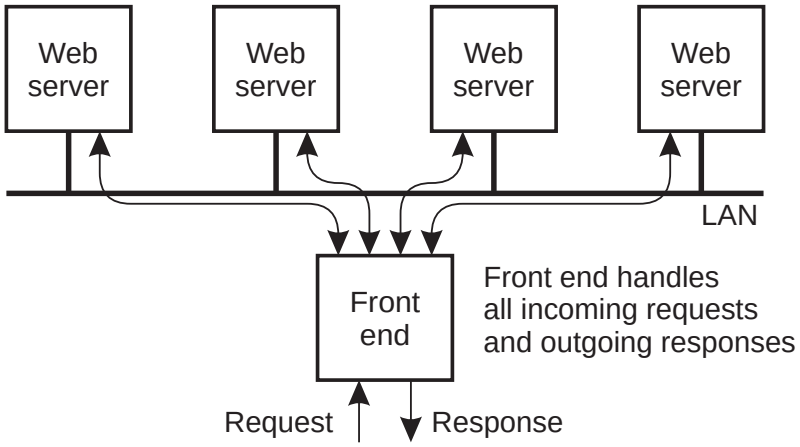
De *Apache core* werkt op basis van Apache core hooks, deze zijn *placeholders* voor specifieke groepen van functies. Elke request wordt door een aantal fasen verwerkt, waarbij elke fase bestaat uit een paar Apache core hooks. Deze Apache core hooks worden behandeld in een specifieke orde.



Figuur 12.4: De algemene organisatie van de Apache Web server.

12.2.3 Web server clusters

Web server clusters worden gebruikt om replica's te voorzien zodat de web servers niet worden overbelast, dit is een voorbeeld van *horizontal distribution*.



Figuur 12.5: Het principe van een server cluster in combinatie met een front end (Web service).

De front end kan gebruikt worden om de client requests om te leiden naar één van de replica's. Hierbij kan er een onderscheidt gemaakt worden tussen front ends die zich gedragen als transport laag switches of deze die optreden op de applicatie laag. Deze twee kunnen dan worden gecombineerd tot een hybride vorm.

Front end at transport layer

- Een eerste optie zou zijn dat de HTTP request wordt door verstuurd van de 'switch' naar een server. Deze server stuurt de HTTP response dan terug naar de 'switch' die op zijn beurt de reponse terugstuurt naar de client. Wat neerkomt op het forwarden van HTTP requests/responses.
  - ☹ De switch wordt een communicatie bottleneck
  - ☹ De switch opereert op basis van de belasting en niet op de inhoud van de requests
- Als optimalisatie kan gebruik gemaakt worden van TCP handoff.
  - ☺ De switch zal geen communicatie bottleneck meer zijn
  - ☹ De switch opereert nog steeds op basis van de belasting en niet op de inhoud van de requests

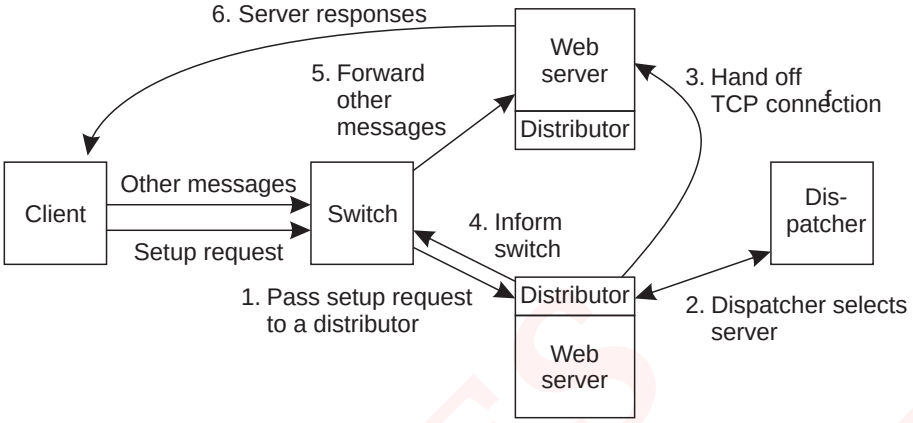
Front end at application layer

Een betere aanpak is om content-aware request distribution te gebruiken.

- ☺ Het bestand kan worden gecached als telkens hetzelfde document op dezelfde server wordt aangevraagd
- ☺ Volledige replicatie zal ook niet meer verplicht zijn, vermits de documenten gedistribueerd kunnen worden over de verschillende servers
- ☹ De front end zal een bottleneck vormen door de verhoogde processing
- ☹ Er is geen TCP handoff mogelijk, als men enkel op de applicatielaag opereert

Hybrid architecture

De hyrbide oplossing bestaat uit een transportlaag switch waarbij de processing uit de switch is gehaald, dit is de distributor. Hierdoor is het ook mogelijk om aan TCP handoff te doen. Dit is weergegeven in figuur 12.6.

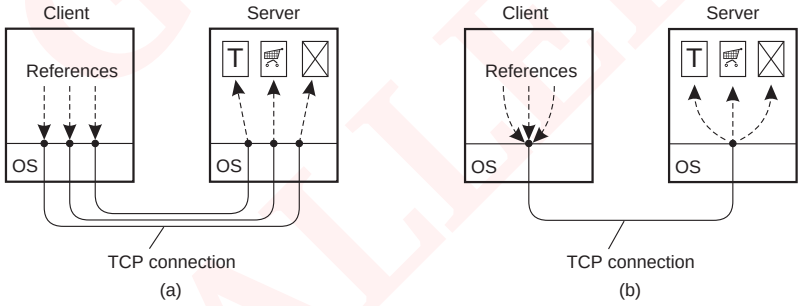


Figuur 12.6: Een schaalbare content-aware cluster van Web servers.

12.3 Communication

Het communicatie protocol voor traditionele Web systemen is HTTP en SOAP voor Web services.

12.3.1 HTTP



Figuur 12.7: (a) non-persistente connecties. (b) peristente connecties.

12.3.2 SOAP

SOAP berichten zijn grotendeels gebaseerd op XML.

Listing 12.1: SOAP Example

```
<?xml version="1.0"?>

<soap:Envelope
```

```
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
...
</soap:Header>

<soap:Body>
...
<soap:Fault>
...
</soap:Fault>
</soap:Body>

</soap:Envelope>
```

12.4 Summary

It can be argued that Web-based distributed systems have made networked applications popular with end users. Using the notion of a Web document as the means for exchanging information comes close to the way people often communicate in office environments and other settings. Everyone understands what a paper document is, so extending this concept to electronic documents is quite logical for most people.

The hypertext support as provided to Web end users has been of paramount importance to the Web's popularity. In addition, end users generally see a simple client-server architecture in which documents are simply fetched from a specific site. However, modem Web sites are organized along multitiered architectures in which a final component is merely responsible for generating HTML or XML pages as responses that can be displayed at the client.

Replacing the end user with an application has brought us Web services. From a technological point of view, Web services by themselves are generally not spectacular, although they are still in their infancy. What is important, however, is that very different services need to be discovered and be accessible to authorized clients. As a result, huge efforts are spent on standardization of service descriptions, communications, directories, and various interactions. Again. each standard by itself does not represent particularly new insights, but being a standard contributes to the expansion of Web services.

Processes in the Web are tailored to handling HTTP requests, of which the Apache Web server is a canonical example. Apache has proven to be a versatile vehicle for handling HTTP-based systems, but can also be easily extended to facilitate specific needs such as replication.

As the Web operates over the Internet, much attention has been paid to improving performance through caching and replication. More or less standard techniques have been developed for client-side caching, but when it comes to replication considerable advances have been made. Notably when replication of Web applications is at stake, it turns out that different solutions will need to co-exist for attaining optimal performance.

Woordenlijst

access point is een speciaal soort van entiteit binnen gedistribueerde systemen. De naam van een access point noemt men een adres. 43, 44

Address Resolution Protocol is een protocol binnen TCP/IP dat computers - die allemaal op hetzelfde netwerk (meer specifiek: LAN) zijn aangesloten - in staat stelt het unieke hardware-adres (MAC-adres) van een andere PC binnen dat netwerk te leren, aan de hand van het IP-adres van deze PC. 44

Apache core hooks zijn *placeholders* voor een groep van functies, die een verwerkingsfase voorstellen van een request. 63

Apache Portable Runtime is een runtime ontworpen om Apache Web servers platform onafhankelijk te maken. 63

APR Apache Portable Runtime. 63, *Glossary:* Apache Portable Runtime

ARP Address Resolution Protocol. 44, *Glossary:* Address Resolution Protocol

attribute-based naming is een naam-systeem die een entiteit beschrijft in termen van `(attribute,value)` paren. De client kan zo via het instellen van de *value* een set van *constraints* op de gezochte *entities* opleggen, zodat hij enkel *entities* krijgt die hem interesseren. 47

care-of address is, in de context van *home-based naming approaches*, het adres van een entiteit die zich buiten zijn home location bevindt. 45

CGI Common Gateway Interface. 61, *Glossary:* Common Gateway Interface

cluster computing systemen worden gebruikt om een parallel geprogrammeerd reken-intensief programma in parallel uit te voeren op verschillende machines. Deze machines hebben dezelfde karakteristieken, i.e. draaien hetzelfde OS en voeren eenzelfde taak uit binnen een netwerk. 8

Common Gateway Interface definieert een standaard waarbij Web servers een programma kunne uitvoeren met gebruikersdata als input. Meestal wordt dit gebruikt onder de vorm van *forms* waarbij de gebruiker het programma die moet worden uitgevoerd en parameters meegeeft. 61

composite services , e.g. een web-based shop, zijn complexe services die meerdere diensten aanbieden a.d.h.v. transacties. 62

content-aware request distribution is een mechanisme waarbij de front end eerst de inkomende HTTP request inspecteert, om dan te besluiten naar welke server de request wordt doorgestuurd. 64

coordination protocols coördineren Web services, i.e. dit protocol beschrijft de stappen die moeten plaatsen opdat composite services zouden slagen. 62

data-centered architectures werken op basis van het publishe/subscribe model waarbij processen los zijn gekoppeld van elkaar, i.e. referential decoupling. Processen abonneren zich op gebeurtenissen –de processen weten niet wie de taak zal uitvoeren. 14

DIB Lightweight Directory Access Protocol. 48, *Glossary:* Lightweight Directory Access Protocol

Directory Information Tree is een hierarchische collectievan *directory entries*, elke node representeert hierbij een *directory entry*, zie figuur 5.5 op pagina 57. 48

directory services is een naam-systeem die gebaseerd is op attribute-based naming. 48

distributed hash table is een soort van gedecentraliseerd distributiesysteem waarin men zoekopdrachten kan uitvoeren, gelijkaardig aan een hashtabel. In een DHT correspondeert elk item met een numerieke sleutel. Elke participerende node kan dan ook op een efficiënte wijze de waarde opzoeken voor een gegeven sleutel. De verantwoordelijkheid voor het onderhouden van de relatie tussen sleutel en waarde wordt verdeeld onder de nodes. Dit gebeurt op zulke wijze dat wanneer er een verandering plaatsvindt bij een van de deelnemers, er een minimale hoeveelheid aan storing plaatsvindt. Dit zorgt ervoor dat DHT schaalt naar zeer grote aantallen nodes en dat ze continu nieuwe, vertrekkende en falende nodes kan afhandelen. 43

distributed information systems werden gebruikt bij organisaties die geconfronteerd waren met een overvloed aan netwerk-applicaties waarbij interoperability een pijnlijke gebeurtenis was. 9

distribution transparancy in een gedistribueerd systeem slaat op het verbergen dat processen en resources verspreid zijn over meerdere fysische computers. De transparantie van een gedistribueerd systeem kan gemeten worden a.d.h.v. hun access, location, migration, replication, concurrency en failure transparancy. 4

distributor is een component binnen de hybride architectuur van een web server cluster. Deze zorgt ervoor dat het verwerken gebeurt buiten de switch. Hierdoor is het ook mogelijk om aan TCP handoff te doen. 64

DIT Directory Information Tree. 48, *Glossary:* Directory Information Tree

EAI enterprise application integration. *Glossary:* enterprise application integration

event-based architectures zijn een combinatie van event-based en data-centered architecturen. Processen zijn zo ontkoppeld in de tijd, beide processen moeten niet actief zijn tijdens de communicatie. Taken worden gedropt in een *shared repository* waarbij componenten deze taken uitvoeren wanneer zij actief worden en de mogelijkheid hebben om deze uit te voeren. 14

extensibility karakteriseert de mate waarin een systeem gemakkelijk kan worden geconfigureerd uit verschillende componenten. Deze componenten moeten ook eenvoudig kunnen worden vervangen, nieuwe componenten moeten kunnen toegevoegd worden. 5

eXtensible Markup Language is een meta-markup language die een grotere flexibiliteit toelaat om het uitzicht van een document te definiëren. XML gedefinieerd de elementen die het document opmaken. 61

finger table is een tabel van nodes die wordt bijgehouden per node in het Chord systeem. Hierdoor is het mogelijk om sneller de node te vinden die verantwoordelijk is voor een sleutel k/item. In de FT wordt er gezocht naar k, de node die k of net k ligt zal worden bezocht, dit gaande tot men een node vindt die k heeft. 46

flat naming zijn *indentifiers* die bestaan uit *random* bit strings, die ook *unstructured names* worden genoemd. Deze bevatten geen informatie over de locatie van het access point van de geassocieerde entiteit. 44

forwarding pointers zijn referenties naar *remote objects*, deze kunnen worden geplaatst in *chains* zodat een object zich kan herlocaliseren. De client hoeft zich dus geen zorgen te maken over verplaatsing van het object, de referenties worden doorheen de *chain* gevolgd naar de huidige locatie. 44, 45

FT finger table. 46, *Glossary:* finger table

gedistribueerde computing systemen worden gebruikt om hoge-performantie berekeningen te kunnen uitvoeren. gedistribueerde computing systemen kunnen worden opgedeeld in twee subgroepen. De eerste opdeling is deze waarbij er gebruik wordt gemaakt van een cluster van gelijkaardige computers, die verbonden zijn met een snelle LAN. Dit zijn de cluster computing systemen. In het grid systeem, de tweede opdeling, worden PC's opgebouwd als een federatie van computer systemen. 8

gedistribueerde pervasive systemen of ook wel embedded systemen genoemd, zijn vaak klein, aangedreven door batterijen, mobiel en beschikken enkel over draadloze connecties. Deze systemen zijn onderdeel van de omgeving. 11

gedistribueerde systemen zijn verzamelingen van onafhankelijke computers waarbij het voor de gebruikers lijkt dat het gaat over één coherent systeem. 3

grid computing systemen hebben een hoge graad van heterogeniteit, geen assumpties over hardware, besturingssystemen, netwerken, administratieve domeinen, beveiligings-policies, etc. Er wordt een virtuele organisatie gecreëerd om alle taken te volbrengen, wat leidt tot een service georiënteerde architectuur waarbij elke machine zijn eigen taak heeft. 8

home agent is, in de context van *home-based naming approaches*, host aangesteld om de locaties bij te houden van entiteiten die binnen dezelfde LAN werden gecreëerd . 45

home location is, in de context van *home-based naming approaches*, het LAN waarbinnen een entiteit werd gecreëerd . 45, 46

HTML HyperText Markup Language. 61, *Glossary:* HyperText Markup Language

HTTP HyperText Transfer Protocol. 65, *Glossary:* HyperText Transfer Protocol

HyperText Markup Language is een markup language die toelaat om links toe te voegen aan documenten. Dit is tevens de meest gebruikte markup language op het Web. 61

HyperText Transfer Protocol is het protocol voor de communicatie tussen een webclient (meestal een webbrowser) en een webserver. 65

IDL Interface Definition Language. 5, 56, *Glossary:* Interface Definition Language

Interface Definition Language beschrijft -meestal enkel- de syntax van de diensten. Dit zijn de namen van de functies, paramtertypes en return waarden van gedistribueerde systemen. 5

interoperability karakteriseert de mate waarin twee implementaties van verschillende makers naast elkaar kunnen bestaan en samenwerken waarbij ze enkel steunen op elkaars diensten die gespecificeerd staan door een gemeenschappelijke standaard. 5, 9

layered architectures zijn architecturen waarbij de componenten als lagen worden georganiseerd. Enkel aanliggende lagen kunnen met elkaar communiceren. De communicatie loopt van boven naar onder, niet omgekeerd. 13

LDAP Lightweight Directory Access Protocol. 48, *Glossary:* Lightweight Directory Access Protocol

Lightweight Directory Access Protocol is een collectie van alle *directory entries* in een LDAP *directory service*. 48

Lightweight Directory Access Protocol is een protocol die gebruikt wordt voor een LDAP directory service aan te bieden, deze bestaat een een aantal *records*, *directory entries*. Een *directory entry* bestaat uit een collectie van `(attribute, value)` paren met elk een geassocieerd type. 48

markup language is een tekstmarkeertaal waarbij de aanwijzingen instructies zijn voor de opmaak tijdens weergave van de tekst. Een gemeenschappelijke eigenschap van opmaaktalen is dat ze tekst afwisselen met opmaakinstructies. Een ander kenmerk is dat die instructies beschrijven hoe de tekst moet worden opgemaakt, en hoe ander materiaal, zoals plaatjes, wordt ingevoegd, maar niet de inhoud van dat andere materiaal beschrijven. De meest gebruikte markup language op het Web is HTML. 60, 61

Message-Oriented Middleware lost het probleem op van RMI en RPC, i.e. waarbij beide systemen *up-and-running* moeten zijn en waarbij er moet geweten zijn hoe ze naar elkaar moeten refereren. Om de *strakke koppeling* te vermijden wordt er gebruik gemaakt van het communiceren via berichten. De systemen worden opgebouwd rond het *publish/subscribe principe*. 11

MIME Multipurpose Internet Mail Exchange. 61, 62, *Glossary:* Multipurpose Internet Mail Exchange

MOM Message-Oriented Middleware. 11, *Glossary:* Message-Oriented Middleware

Multipurpose Internet Mail Exchange was origineel ontworpen om informatie te geven over de inhoud van een bericht van een elektronische mail. MIME verdeelt de types onder toplevel types en subtypes, e.g. Text (type), Plain/HTML/XML (subtype). 61

name-to-address binding binnen gedistribueerde systemen is behoudt in zijn simpelste vorm `[name, address]` paren in een tabel. 43

natuurlijke taal beschrijft wat de diensten doen van gedistribueerde systemen, d.i. de semantiek van de systemen. 5

object adapter of *object wrapper* zal objecten groeperen met eenzelfde activitatie policy. 55, 56

object-based architectures werken op basis van het client/server model waarbij objecten communiceren via (R)PC. 13

portability karakteriseert de mate waarin een applicatie ontwikkeld voor een systeem A kan worden uitgevoerd, zonder modificatie, door een ander gedistribueerd systeem B met dezelfde interface (IDL) als A. 5

proximity routing is een manier om het onderliggende netwerk in rekening te brengen in DHT-systemen, e.g. Chord systeem. De dichtste node wordt gekozen als buur. 47

proximity routing is een manier om het onderliggende netwekr in rekening te brengen in DHT-systemen, e.g. Chord systeem. Er wordt hier een lijst bijgehouden van alternatieven per entrie. Wat ook mogelijk maakt dat een node failure niet altijd zal leiden tot een lookup failure. 47

referential decoupling is het ontkoppelen in de ruimte. Componenten hoeven de referentie van de componenten die de taak uitvoert niet te kennen. 14

Remote Method Invocation is een speciaal geval van RPC, het functioneert op objecten in plaats van applicaties. 11

Remote Procedure Call is een communicatie middleware die de applicatie ontwikkelaar de mogelijkheid geeft om methodes op te roepen van applicaties die aanwezig zijn op andere machines. 11

requests idempotent Wanneer het antwoord verloren raakt dan kan het herverzenden van de request ervoor zorgen dat de operatie tweemaal wordt uitgevoerd. Als dit niet resulteert in een verkeerde situatie dan is de operatie idempotent. 15

RMI Remote Method Invocation. 11, 56, 57, 59, *Glossary:* Remote Method Invocation

RPC Remote Procedure Call. 11, 56, 58, *Glossary:* Remote Procedure Call

scalability karakteriseert de mate waarin een applicatie schaalbaar is. Dit kan worden opgemeten in drie dimensies: grote, locatie en administratie. 5, 6

Simple Object Access Protocol standaardiseert de communicatie tussen twee processen. 62

SOAP Simple Object Access Protocol. 62, 65, *Glossary:* Simple Object Access Protocol

TCP handoff is een mechanisme waarbij de TCP connectie tussen een dispatcher en een client wordt doorgegeven naar een andere server. Deze server zal de connectie verder gebruiken, wat onzichtbaar gebeurt voor de client. 64

topology based assignment of identifiers is een manier om het onderliggende netwekr in rekening te brengen in DHT-systemen, e.g. Chord systeem. De wereld wordt gemapped op een 1-dimensionale ring en zorgt dat dichtstbijliggende nodes, sleutels hebben die ook dicht bij elkaar zitten. Het nadeel hieraan is dat er geen uniforme distributie van sleutels is en er dus bij het wegvallen van netwerk, een leegte ontstaat. 47

TP monitor transaction processing monitor. 10, *Glossary:* transaction processing monitor

transaction processing monitor behandelt de gedistribueerde (of geneste) transacties om applicaties op server of databank niveau te integreren. Dit is weergegeven in figuur 1.4 op pagina 19. De hoofdtaak was om applicaties toegang te bieden voor meerdere servers/databanken –door een transactional programming model aan te bieden. 10

transaction processing systems is een type van een gedistribueerd systeem waarbij de RPCs (*Remote Procedure calls*) vaak worden ge-encapsuleerd in een transactie. `BEGIN_TRANSACTION` en `END_TRANSACTION` worden gebruikt om de scope van de transactie af te bakenen. In de body van een transactie worden vaak alle operaties uitgevoerd of geen. 9

UDDI Universal Description, Discovery and Integration standaard. 62, *Glossary:* Universal Description, Discovery and Integration standaard

Uniform Resource Locater specificeert waar een document zich bevindt, veelal door het toevoegen van de DNS naam van de geassocieerde server in combinatie met de bestandsnaam. De URL specificeert ook het protocol van de applicatie-laag voor het transfereren van bestanden, e.g. HTTP. 60

Universal Description, Discovery and Integration standaard beschrijft de layout van een databank die *service descriptions* bevat, die Web service clients toelaten om relevante services te zoeken. 62

URL Uniform Resource Locater. 60, *Glossary:* Uniform Resource Locater

Web proxy laat toe om andere applicatie-laag protocollen te gebruiken –buiten HTTP. 62

Web Service Coordination describes an extensible framework for providing protocols that coordinate the actions of distributed applications. Such coordination protocols are used to support a number of applications, including those that need to reach consistent agreement on the outcome of distributed transactions. 62

Web services zijn Web gebaseerde systemen die diensten bieden aan *remote applications* zonder vereiste van interactie met gebruikers. 61, 65

Web Services Definition Language beschrijft een service a.d.h.v. een formele taal, gelijkend om de IDL bij RPC. 62

WSDL Web Services Definition Language. 62, *Glossary:* Web Services Definition Language

XML eXtensible Markup Language. 61, 65, *Glossary:* eXtensible Markup Language