# Distributed Systems
# -
# Examenvragen

Luther D & Stanny B

# Replicatie

1. Verschil **lineariseerbaarheid** en **seq consistentie** uitleggen

Lineariseerbaarheid:

- de interleaved sequentie van operaties voldoet aan de specificatie van een kopie van de objecten
- de volgorde van de operaties in de interleaving is consistent met de echte tijdstippen waarop de operaties plaatsvonden. Vereist clock-synchronisatie.

Sequentiële consistentie

- de interleaved sequentie van operaties voldoet aan specificatie van een kopie van de objecten
- voor alle gebruikers geldt de volgorde van de operaties in de interleaving is consistent met de onderlinge volgorde van operaties van deze gebruiker (=local time order).

**Lineariseerbaarheid heeft sequentiële consistentie als rechtstreeks gevolg (maar niet andersom).**

Een seq cons. voorbeeld dat niet lineariseerbaar is.

| Client A | | Client B |
|---|---|---|
| setBalance_B(x,1) | 1 | |
| | 2 | getBalance_A(y) -> 0 (returns 0) |
| | 3 | getBalance_A(x) -> 0 (returns 0) |
| setBalance_A(y,2) | 4 | |

Een interleaving die sequentieel consistent is, is bijvoorbeeld: 2-3-1-4. Echter, dit is niet lineariseerbaar, die vereist namelijk de volgorde 1-2-3-4 (om overeen te komen met de real-time uitvoering), maar deze voldoet niet aan de specificatie van een kopie van de objecten.

2. Bespreek **passive replication** (doel, ontwerp, voor-/nadelen, varianten)

= primary-backup replication. Het doel is om een consistent correcte service te geven aan clients, zelfs indien er tot f process failures zijn (vereist f+1 RM's).

Sequence of events for handling a client request:

- Request: FE issues request with unique id to primary
- Coordination: request handled atomically in order; if request already handled, re-send response
- Execution: execute request and store response
- Agreement: primary sends updated state to backups and waits for acks
- Response: primary responds to FE; FE hands response back to client

(FE = Front End)

Er is op elk moment een Primaire Replica Manager en 1 of meerdere secundaire Replica Manager(s) (ook backups/slaves). Indien de PRM faalt, wordt 1 van de backups gepromoveerd tot PRM.

Dit systeem is lineariseerbaar want alles passeert de PRM, dus er kan een globale clock worden bijgehouden.

Wanneer de PRM crasht en een backup de nieuwe PRM wordt, zal het systeem lineariseerbaar blijven indien:

- de PRM vervangen word door een unieke backup (= zelfde voor alle clients)
- alle resterende RMs overeen komen over welke operaties al uitgevoerd waren

Deze vereisten zijn steeds voldaan als de RMs georganiseerd zijn als een groep en als de PRM view-synchronous group communication gebruikt om updates te sturen naar de backups. Dit impliceert wel veel overhead.

Wanneer de read requests ook door de backups zouden behandeld worden zou het systeem enkel nog sequentieel consistent zijn.

Niet-deterministisch gedrag van de PRM is ondersteund (bv in geval van multi-threading, de PRM stuurt het resultaat naar de slaves en niet de operaties)

3. Bespreek **active replication** (doel, ontwerp, voor-/nadelen, varianten)

Het doel is om een consistent correcte service te geven aan clients, zelfs indien er tot f process failures zijn.

Sequence of events for handling a client request:

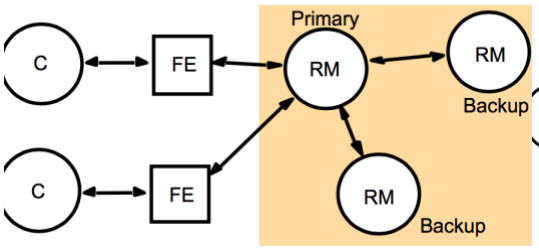
- Request: FE does reliable TO-multicast(g, <m, i>) and waits for reply
- Coordination: every correct RM gets requests in same order
- Execution: every correct RM executes the request; all RMs execute all requests in the same order
- Agreement: not needed
- Response: every RM returns result to FE; when return result to client?

Afhankelijk van de strategie wordt het antwoord doorgestuurd na 1 antwoord van een RM (crash failure-principe), of na f+1 identieke antwoorden van de RM's (Byzantian failure-principe). Als er een RM crasht heeft dit dus geen impact => vereist 2*f+1 RM's

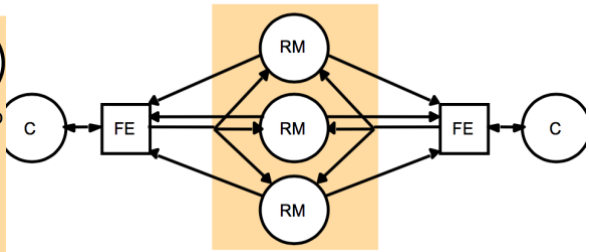
Enkel sequentiële consistentie vermits de totale orde waarin de RMs de requests processen niet per se dezelfde is als de real-time orde waarin de clients hun requests deden.

Om de performance van active replication te verhogen kunnen we eventueel de totale orde niet afdwingen wanneer geldt dat: O1 -> O2 = O2 -> O1. Ook kunnen we read-only requests doorsturen naar één RM (deze veroorzaken namelijk geen aanpassing die moet worden gepropageerd).

- Passive (primary-backup) replication




- Active replication




4. Vergelijk passive replication met active replication op vlak van:
   a. Performantie

Bij passieve replicatie gaan alle requests afgehandeld worden door de PRM. Daarna moeten alle aanpassingen nog door gepropageerd worden naar de backups. Dit zorgt voor een grote overhead tov actieve replicatie (waar alle RMs autonoom werken d.m.v. multicast).

   b. Beschikbaarheid

Beide systemen zijn fault-tolerant systems en kunnen f failures aan. Bij passieve replicatie zal de PRM, indien deze faalt, vervangen worden door een van de backups. Bij active replicatie zal een failure van een RM waarschijnlijk niet eens opgemerkt worden doordat er nog steeds n-1 antwoorden komen op de request.

   c. foutbestendigheid en consistentie

Active replicatie is enkel sequentieel consistent, vermits de totale orde waarin de RMs de requests processen niet per se dezelfde is als de real-time orde waarin de clients hun requests deden. Passieve replicatie is volledig lineariseerbaar indien de PRM goed werkt. Het blijft lineariseerbaar indien een backup PRM wordt en:

- de PRM vervangen word door een unieke backup (= zelfde voor alle clients)
- alle resterende RMs overeen komen over welke operaties al uitgevoerd waren

5. Bespreek de replicatie-aspecten van Coda (doel, ontwerp, voor/na-delen, varianten)

Is gebaseerd op AFS. Coda is vooral gefocust op het beschikbaar houden van data (availability) zorgt ervoor dat wanneer gebruikers hun toestel deconnecteren van het netwerk, of in geval van network of server failure, hun data toch (deels) lokaal beschikbaar blijft. Levert betere performance, is meer fout-tolerant met toenemende schaal, en toestellen kunnen mobiel zijn (disconnected operatie).
Het doet dit door aan AFS replicatie toe te voegen:
(file volume = set files en directories, de unit van replicatie in Coda)

- File volumes zijn nu gerepliceerd op verschillende servers
- Elk file volume heeft een Volume Storage Group (VSG) (= collectie van servers die een kopie hebben van deze file volume) en een Available Volume Storage Group (AVSG) op een bepaald tijdstip (= de servers in de VSG die de client op dat tijdstip kan contacteren).
- Wanneer de AVSG leeg is, is het volume gedeconnecteerd

Elk bestand krijgt een file version. Dit is een integer die verhoogd wordt wanneer het bestand aangepast wordt. Er wordt ook een Coda Version Vector (CVV) bijgehouden bij een kopie van een bestand op een bepaalde server. 1 int per volume/entry in VSG. Deze CVV kan gebruikt worden om conflicten te detecteren.
De beschikbaarheid van AFS is dus verhoogd. Indien een bepaalde server niet bereikbaar is, zorgt de replicatie ervoor dat het bestand toch nog beschikbaar is op een andere server.
Implementatie: bij een open:

- kies één server uit AVSG
- check CVV met alle servers in de AVSG
- de files in een replicated volume zijn beschikbaar voor alle clients die op zijn minst toegang heeft tot één van de replica's
- performance kan verhoogd worden door load sharing = verdelen van client requests voor een replicated volume over alle servers die replica's ervan houden

Bij close: - multicast files naar AVSG

- update de CVV

Manuele resolutie van conflicten kan nodig zijn
Relevante evenementen die Venus detecteert binnen T seconden m.b.v. probe messages naar alle VSG's van een gecachte file elke T seconden:

- vergroting of verkleining van AVSG
- verloren callback

Omgaan met disconnections: - Venus monitort file referenties en users kunnen een priority list specificeren => Venus probeert om deze in cache te houden indien mogelijk
Reïntegratie na disconnectie: Venus doet update operaties om file volumes van AVSG's identiek aan cache kopies te maken, conflicten worden op covolumes opgeslagen, en de user word op de hoogte gebracht.
De performantie is uiteraard minder goed dan AFS. Wanneer we 3-fold replication doen en een load van 50 users stellen, zal dit resulteren in +70% load voor Coda en +16% load voor AFS.

6. Vergelijk Coda en passieve replicatie.

Coda focust op het beschikbaar houden van data (**availability-focussed**) terwijl passieve replicatie vooral dient om met het crashen van servers om te gaan (**failure-focused**).
Coda zal er voor zorgen dat een gebruiker nog steeds toegang heeft tot bestanden door file referenties te monitoren en mbv een priority list. Zo wordt de availability behouden ook wanneer de client geen verbinding meer heeft met de servers. Dit maakt het uitermate geschikt voor mobiele toestellen.
Passieve replicatie dient voor het beschikbaar houden van de data op de servers door te zorgen dat er bij de crash van de PRM een nieuwe PRM gekozen wordt. Passieve replicatie dient dus vooral voor failure handling (van de servers).

7. Bespreek de verbeteringen die Coda aan AFS aanbrengt op vlak van performantie en beschikbaarheid. Bespreek het ontwerp en de belangrijkste ontwerpbeslissingen van Coda.

= vraag 5

# Gedistribueerde transacties

Bekijk evt slides 17-28 voor illustratie 2 concurrency problemen.

Bekijk evt slides 34-43 voor illustratie 2 recovery problemen.

1. Beschrijf het two-phase commit protocol voor gedistribueerde transacties. Je mag gebruik maken van de operaties CanCommit => Yes/No, DoAbort(Trans), DoCommit(Trans), HaveCommitted(Trans) en GetDecision(Trans)

Ontworpen zodat elke server de mogelijkheid heeft om zijn deel van de transactie te aborten => dan moet de hele transactie geabort worden. Het protocol moet correct werken, zelfs als sommige servers crashen, sommige berichten verloren gaan en servers tijdelijk niet in staat zijn te communiceren.
Het two-phase commit protocol bestaat uit 2 fasen:
Fase 1 - voting fase (servers voten voor het resultaat)
De client wil committen. Wanneer de coördinator dit bericht ontvangt stuurt hij een canCommit() naar elke participant en wacht hij op antwoorden.
Wanneer een participant een canCommit() ontvangt, antwoord hij met *Yes* of *No*. Voor *Yes* te antwoorden maakt hij zich eerst klaar (alle objecten opslaan in permanent geheugen zodat ze recoverable zijn in geval van crash).
Wanneer een participant niet kan committen antwoordt hij 'no' en abort.

Fase 2 - alle servers voeren het resultaat van de stemming uit
De coördinator verzamelt alle stemmen. Als alle stemmen *Yes* zijn, beslist hij om de hele transactie te committen en stuurt hij een doCommit(Trans) naar alle participants.
Indien er minstens 1 'no' is, kiest hij er voor om de hele transactie af te breken en stuurt hij een doAbort(Trans) naar alle participants.
Een participant wacht tot hij een bericht krijgt van de coördinator en voert de nodige stappen uit.
Indien dit een doCommit(Trans) is, maakt hij de gecommite data beschibaar, verwijdert hij zijn locks en stuurt hij haveCommitted(Trans, Participant) naar de coördinator.
Als hij een doAbort(Trans) krijgt, verwijdert hij de datastructuren en zijn locks.
Timeouts worden gebruikt om te voorkomen dat processen voor altijd blokkeren. In geval van timeout moet de gepaste actie gebeuren: - abort => als coördinator nog niet alle votes ontvangen heeft of als worker wacht op canCommit

- evt getDecision sturen naar coördinator => als worker *Yes* gevote heeft en wacht op finale beslissing

2. Waarvoor wordt getDecision gebruikt?

Wanneer een participant yes heeft geantwoord op een canCommit() vraag, kan hij in theorie oneindig lang wachten op de eindbeslissing van de coördinator. Bijvoorbeeld als er iets mis was gelopen in de communicatie. In dit geval kan hij een getDecision() sturen naar de coördinator. Als hij dan een antwoord krijgt, blijft hij wachten. Als hij geen antwoord krijgt zal de coördinator waarschijnlijk gefaald zijn en heeft hij een "uncertain state".

3. Waar zitten de timeouts? (prof vroeg op mondeling ook om deze aan te duiden op een tekening die hij maakte)

- Een participant heeft zijn operations voltooid en wacht op een canCommit()
- De coördinator wacht op de stemmen van de participants
- Een participant heeft yes geantwoord en wacht op de finale beslissing van de coördinator.

In de eerste 2 gevallen kan er dan eventueel gekozen worden voor een eenzijdige abort. In het laatste scenario kan GetDecision() gebruikt worden, de worker kan namelijk niet zomaar aborten aangezien hij al gezegd heeft dat hij zal committen.

4. Bespreek het two-phase commit protocol voor nested transactions (0,5 blz)
    a. Waarom zou je nested transactions gebruiken?
    b. Wat moet je veranderen in het two-phase commit protocol voor nested transactions?

a) nested transactions = transaction bestaande uit meerdere subtransacties
- bied modulaire aanpak om transacties te structureren in applicaties
- middel voor het beheersen van concurrency binnen een transactie: gelijktijdige subtransacties die access vragen aan shared data worden geserialised
- een "finer grained" herstel van fouten: subtransacties mislukken onafhankelijk

b) The buitenste transactie = top-level transactie, dit is de coördinator, alle andere zijn subtransacties, het ID van subtransactie is een extensie van de parent-ID en is globaal uniek. Elke child-transactie start na zijn parent en eindigt voor zijn parent.
Wanneer een subtransactie voltooid, maakt het een onafhankelijke beslissing om: - te aborten, impliceert de abort van al zijn nakomelingen of - voorwaardelijk te comitten (/= Yes van hierboven, er word niet permanent in storage opgeslagen dus bij crash zal er niet gecommit kunnen worden).
=> voor beide: status inclusief status nakomelingen word aan parent gereport.
Commit list (de lijst van alle gecommitte (sub)transactie) evenals abort list word bijgehouden.
Als alle subtransacties voltooid zijn, nemen de voorwaardelijk gecommite deel aan de 2-fase-protocol. Coördinator verzend CanCommit met transactionID en abort list naar workers in commit list en gedraagt zich zelf als een worker (=> moet in fase 2 dus zelf ook committen).
Voor workers: fase 1: - indien minstens 1 (voorw.) committed nakomeling van top-level transactie => transacties met ancestor in abort list aborten zelf, andere bereiden voor en worker zend Yes.
- indien geen (voorw.) committed nakomelingen: zend No naar coördinator.

5. Gedistribueerde transacties worden voorzien op gedistribueerde component platformen? Waar zit er nog werk voor de applicatie ontwikkelaar? Teken dit en gebruik deze termen 2PC, gedistribueerde transacties, gedistribueerde componenten, remote method invocation,.... (nog aan te vullen). Hij verwachtte hier een figuur van de slides die niet in de cursus staat, dat verschillende verbonden systemen met elk hun eigen database allemaal participant kunnen spelen en je dan een coördinator moet implementeren die alles regelt

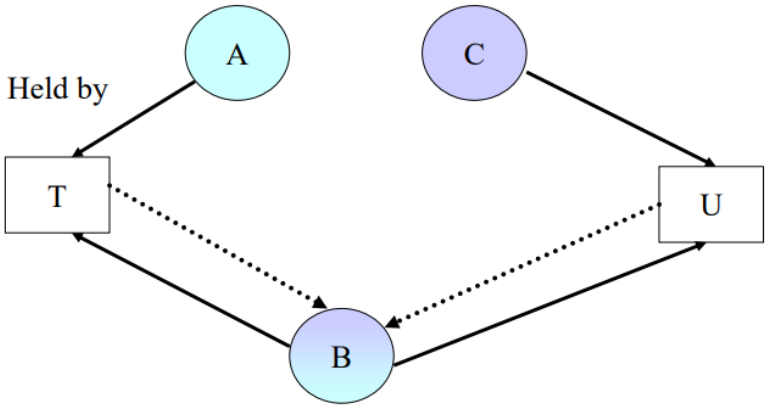
Wanneer een gedistribueerde transactie beëindigd, moeten door de atomicity eigenschap alle betrokken servers ofwel comitten ofwel aborten. Om dit te bekomen neemt 1 server de rol aan van coördinator. Deze houd alle betrokken servers bij (genaamd workers) en is verantwoordelijk voor de finale beslissing.
Zie slides 102-106

"Waar zit er nog werk voor de applicatie ontwikkelaar?" De ontwikkelaar moet controleren of de transactie effectief is uitgevoerd -> omgaan met errors.

## 6. Hoe kan een deadlock gedetecteerd worden?

Deadlock = een staat waarin elk lid van een groep van processen aan het wachten is op een ander lid om iets te unlocken => geen vooruitgang mogelijk.
Er kan een **wait-for graph** bijgehouden worden. Op een wait-for graph wordt weergeven op welke recources een process een lock heeft (edge van recourse naar process) en op welke recources een process aan het wachten is (edge van process naar recource). Wanneer er een cycle gedetecteerd wordt in de wait-for graph wilt dit zeggen dat er sprake is van een deadlock. De aanwezigheid van cycles kan periodisch gecontroleerd worden of telkens bij het toevoegen van een edge. Eens een deadlock gedetecteerd word moet de server een process selecteren en aborten. Keuze gebaseerd op: leeftijd process en in hoeveel cycles de process aanwezig is.

Voorbeeld: (process T en U zijn in deadlock over recourse B)




In een gedistribueerd systeem is het echter niet goed om een gecentraliseerde deadlock detectie (zoals globale wait-for graph) te gebruiken omdat:

- hangt af van één server (reliability)
- kost van het doorsturen van de lokale wait-for graphs

**Edge chasing** is een gedistribueerd algoritme om deadlocks te detecteren zonder het opstellen van een globale wait-for graph. Deadlocks worden gevonden door probes door te sturen die de edges van de overeenkomstige wait-for graphs volgen. Deze probes onthouden bij welke processen ze passeren. Wanneer er een cycle in de afgelegde weg van een probe is, betekent dit dat er een deadlock is.
Edge chasing in 3 stappen:

- initiation: wanneer een transactie T begint te wachten op U (en U is aan het wachten)
  - zend een nieuwe probe T->U
  - in geval dat U een lock shared, worden de probes doorgestuurd naar alle holders van het lock, ook als in de toekomst het lock geshared word.
- detection: probe (T->U) ontvangen
  - check of U nog aan het wachten is
  - als U aan het wachten is op V (en V is aan het wachten), voeg V toe aan de probe

    - probe controleren op cycle: ja -> deadlock -> resolutie
    - nee -> probe forwarden
- resolutie: abort 1 transactie

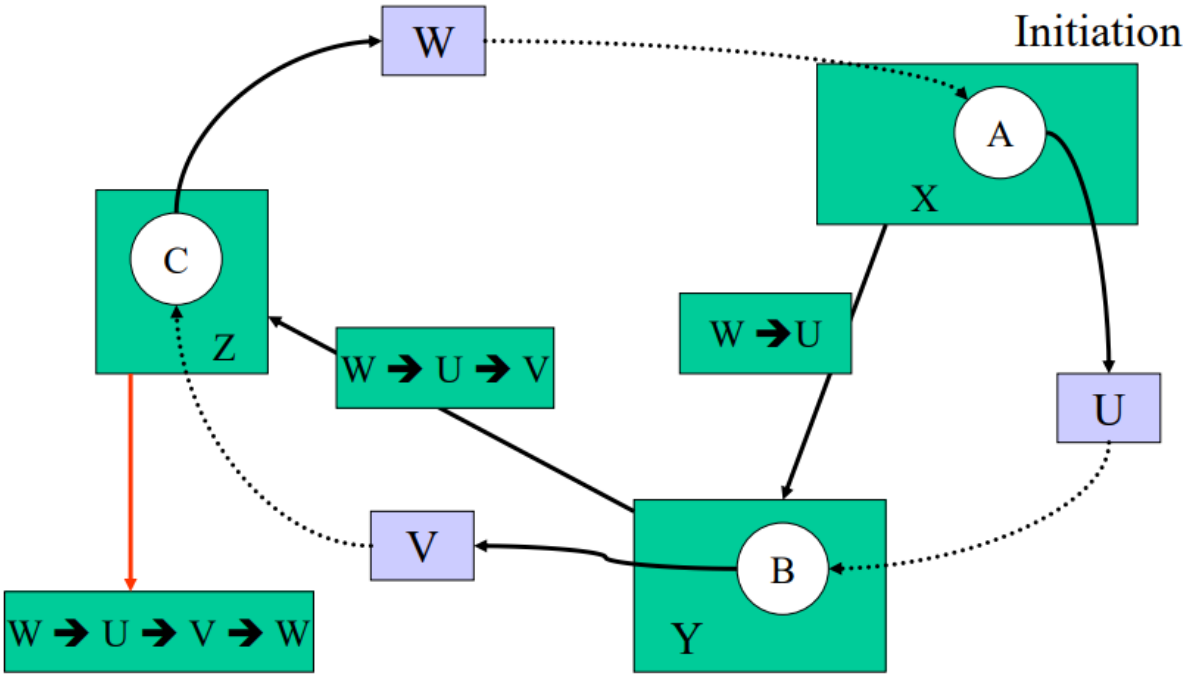
Problemen met edge chasing: - elke wachtende transactie kan een probe starten

- detectie kan op verschillende servers gebeuren => kunnen meerdere transacties aborten

Deze problemen kunnen opgelost worden met transactie prioriteiten: ken prioriteit toe aan elke transactie (e.g. via timestamps), abort transactie met laagste prioriteit, als meerdere servers dezelfde cycle detecteren zal nu dezelfde transactie geabort worden.
Verder kunnen transactie prioriteiten er ook voor zorgen dat er minder probe messages geïnitialiseerd worden (probe enkel wanneer high priority wacht op low priority) en er zijn minder doorgestuurde probe messages (probes gaan steeds van transacties met hoge prioriteit naar transacties met lage prioriteit, hiervoor zijn probe queues vereist).
(Evt oplossing deadlocks: timeouts: locks worden toegewezen voor een gelimiteerde periode.)

7. Beschrijf het edge chasing algoritme (Hier wil hij blijkbaar het algoritme in pseudocode en niet enkel een beschrijving). Hoe werkt dit concreet voor gedistribueerde transacties? Leg uit adhv een figuur.

Zie hierboven.




8. Hoe werken prioriteiten in gedistribueerde transacties.

Zie hierboven

9. Welke rol kunnen transactieprioriteiten spelen bij deadlock detection?

Zie hierboven

Kan mss ook nodig zijn bij een vraag:

- Locking rules for nested transactions
  - child transaction inherits locks from parents
  - when a nested transaction commits, its locks are inherited by its parents
  - when a nested transaction aborts, its locks are removed
  - a nested transaction can get a read lock when all the holders of write locks (on that data item) are ancestors
  - a nested transaction can get a write lock when all the holders of read and write locks (on that data item) are ancestors

# Communicatie

1. Java RMI
    a. Leg algemeen uit waarom je in het algemeen een klasse Remote, Serializable of Local zou maken.

**Remote**: als een instantie van de klasse vanop afstand bereikbaar moet zijn. Als een bepaald object een methode op een object van een andere JVM wil invoken, moet dit tweede object remote zijn (dit wil zeggen dat het een instantie is van een klasse die een remote IF implementeert). Bij het invoken van de methode op zo'n remote object, worden de argumenten gemarshalled en van de (lokale) VM naar de remote VM gestuurd, die unmarshalled de data, voert de methode uit en marshalt dan het resultaat en zend het terug.
Om een remote object toegankelijk te maken op andere JVM's, moet je het met de RMI registry registreren via bind("name", het remote object zelf). Een andere JVM kan dan dit remote object verkrijgen via lookup("name"), de registry returnt een reference (stub) naar het remote object.
**Serializable**: Wanneer een instantie van de klasse over het netwerk verstuurd moet kunnen worden (=marshallen) maar het niet nodig is dat veranderingen op het object door propageren naar het originele object dan gebruiken we Serializable = pass-by-value ⇔ pass-by-reference. Alle niet-remote objecten die als argument meegegeven worden bij een RMI moeten dus serializable zijn. Een object wordt ook geserialized alvorens het wordt opgeslagen in een database.
**Local**: wanneer een klasse gewoon lokaal in de client/server bestaat en deze niet vanop afstand bereikbaar moet zijn, is deze klasse gewoon local.

b. Pas toe op je practicum (geef voor elke klasse het type). Geef je redeneringen en bespreek eventuele trade-offs.

- Car: moet door geen enkele remote methode teruggegeven worden noch als argument van een meegegeven worden ® Local
- CarRentalCompany: moet remotely accessible zijn voor alle reservation en manager sessions ® Remote
- CarType, Quote, Reservation, ReservationConstraints, ReservationException: moeten tussen de client en server gecomunniceerd worden, alleen de data is belangerijk, er is geen referentie naar het originele object ® Serializable

Alernatief: een String gebruiken om de CarType te communiceren aangezien de naam van de CarType auto de andere attributen impliceert. Dit kan geïmplementeerd worden adhv een CarTypeParser object. Deze parser zou dan een parseStringToCarType() en parseCarTypeToString() methode hebben en voor elke CarType naam alle attributen moeten weten.
Dit zou leiden tot een lagere last op de server omdat deze dan alleen de CarType naam in de vorm van een String moet sturen ipv een volledig geserialized CarType object. Als tradeoff introduceert dit echter extra complexiteit en een CarTypeParser object dat alle CarTypes moet kennen. Daarom opteerden we hier niet voor.

c. Stel dat je een klasse Serializable hebt gemaakt terwijl die eigenlijk Remote had moeten zijn, wat voor gevolgen heeft deze fout? Illustreer aan de hand van je practicum.

Wanneer we een klasse serializable maken i.p.v. remote, zal deze niet meer remotely accessible zijn door de clients. Vermits de klasse nu wel serializable is kunnen we de waarde van de instantie van de klasse steeds mee door te geven als argument bij elke operatie die anders de remote instance zou gebruiken.

Bijvoorbeeld: CarRentalCompany is een remote class in het practicum. Nu gebruikt de client deze klasse remotely om bijvoorbeeld een quote te bevestigen. Indien deze klasse niet meer remote zou zijn, krijgt de client bij het oproepen van de methode een local object terug ipv een Remote reference.
Als gevolg zal de quote toegevoegd worden aan het lokale CarRentalCompany en dus niet het gewenste effect hebben aangezien dit object immers niet op de server staat en dus niet aan de originele CarRentalCompany toegevoegd worden, e.g. andere clients kunnen dezelfde quote boeken.

# Java EE

1. Leg uit voor Java EE en maak onderscheid tussen container-managed en bean-managed transactions
   a. Wat is een transactie?

Een transactie is een set van gerelateerde operaties die samen uitgevoerd moeten worden, is een manier om consistency van data te bekomen in gedistribueerde systemen.
Een transactie bestaat uit de operaties: start(), commit(), rollback().
Een transactie wordt gekenmerkt door de ACID-eigenschappen:

- Atomicity : “all-or-nothing”: een transactie beëindigd ofwel succesvol (commit, gegarandeerd dat het persist) of heeft geen effect (abort, kan geïnitieerd worden door systeem of door user).
- Consistency : Data gaat van de ene consistent state naar de andere

- Isolation : Er is geen interferentie van andere transacties: serializability: gelijktijdige transacties hebben hetzelfde effect als een seriële executie en failure isolation: een transactie ziet geen uncommitted effects van een andere transactie, ook niet als deze gefaald is.
- Durability : Eens een transactie gecommit is, blijft ze gecommit, zelfs in geval van system failure.

### b. Hoe implementeer je dit in Java EE (bespreek granulariteit, demarcatie, rollback, ...)

In java EE bestaan er 2 soorten Transactions:
Container-managed transactions (CMTs):
De EJB-container zet de grenzen van de transacties (CMT-demarcatie). Deze zal dus een transactie starten net voor een methode van een (session of message-driven) Enterprise Bean start en deze beëindigen net voor de methode stopt. => vergemakkelijkt development want de EB-code moet de transactiegrenzen niet expliciet bepalen en geen calls naar start(), commit(), rollback(). Elke methode is dus geassocieerd met één transactie => geneste of meerdere transacties binnen een methode zijn niet mogelijk. Via transaction attributes wordt bepaald welke methodes geassocieerd worden met transacties. (niet alle methodes moeten met een transactie geassocieerd zijn)
Er zijn twee mogelijke situaties:

- De client is al aan het uitvoeren binnen een transactie en roept een methode van een enterprise bean op
- De client is nog niet geassocieerd met een transactie

Binnen CMT zijn er 2 manier voor een roll-back uit te voeren:

- Als een system exception gegooit word, zal de container automatisch een rollback doen
- De EB-methode instructs de container om een rollback van de transactie te doen door de setRollBackOnly methode van de EJBContext IF te invoken (is dus niet automatisch)

Bean-managed transactions (BMTs):
De transactie-grenzen worden expliciet door de code in de session of message driven bean bepaald. Dit kan je doen met 2 APIs:

- JDBC transactions: de transactie word gemanaged door de RDBMS, implementatie via het invoken van de commit en rollback van de java.sql.Connection IF. Het beginnen van een transactie is impliciet bij de eerste query.
- JTA transactions: de transactie word gemanaged door de JEE server gebruik makende van de Java Transaction Service (JTS) (echter roep je nooit direct de JTS methodes op) => onafhankelijk van de transaction manager implementatie

Rollbacks moeten manueel uitgevoerd worden => niet via setRollBackOnly, maar via de getStatus en rollback methodes van de UserTransaction IF.
CMTs zijn makkelijker te implementeren echter kan een methode enkel geassocieerd worden aan één of geen transactie. Indien dit het coderen van je bean moeilijk maakt kan je beter BMTs gebruiken.

### c. Geef 3 TransactionAttributes en hun betekenis

Hier alle 6 transactionattributes:
Er zijn twee mogelijke situaties:

- De client is al aan het uitvoeren binnen een transactie en roept een methode van een enterprise bean op (1)
- De client is nog niet geassocieerd met een transactie (2)

| | (1) | (2) |
|---|---|---|
| **Required** | De methode word binnen de client transactie uitgevoerd | Container start een nieuwe transactie voor methode uit te voeren |
| **RequiresNew** | stop client transaction,<br>start new transaction,<br>delegate call to method,<br>resume client transaction after method completed | container starts new transaction before running the method |
| **Mandatory** | De methode word binnen de client transactie uitgevoerd | container gooit TransactionRequiredException |
| **Not supported** | container suspends client's transaction until method had completed | container does not start a new transaction |
| **Supports** | method executes within client's transaction | container does not start a new transaction |
| **Never** | container gooit exception | Container start een nieuwe transactie voor methode uit te voeren |

d. Waar komt er een transactie voor in je practicum? Welk type transacties heb je gebruikt en waarom?

JDBC

2. Persistance uitleggen aan de hand van het practicum

Persistence wilt zeggen dat de data niet verdwijnt bij het beëindigen van het process dat het gecreëerd heeft. In het practicum worden Car, CarRentalCompany, Cartype en Reservation beschreven met de @Entity annotatie. Dit zijn dus entities. Entitiy classes zijn "Business objects" dit zijn objecten die persistent zijn, relaties kunnen hebben en een state EN behaviour hebben. In de praktijk stellen ze een tabel in een relationele database voor. Een object zelf wordt dan een rij. Elke entity class is ook voorzien van een primary key (aangeduid met @Id of @IdClass). Op entities kunnen relaties geannoteerd worden d.m.v. bijvoorbeeld @OneToMany(Cascade=ALL, mappedBy="departement").In het geval van ons practicum zijn er relaties gedefinieerd vanuit CarRentalCompany naar Car en CarType.
De CarRentalSession en ManagerSession gebruiken beide een EntityManager. Een EM definieert methodes (retrieve via queries, search, create, remove) om te interageren met de persistence context (PC) en beheert hun lifecycle. De PC is een set van gemanagede entity instances die bestaan in een bepaalde data store, alle entities in de context worden op dezelfde manier gepersist. Een EM kan zowel container-managed (container propageert automatisch de PC naar alle app components die de EM gebruiken) als application managed zijn (de PC word niet gepropageerd en de app beheert de life cycle). In ons practicum gebruiken we de EM bijvoorbeeld om specifieke instanties te vinden van een CarRentalCompany. Verder gebruiken we in het practicum ook NamedQueries om instanties op te zoeken (in Java Persistence Query Language (JPQL)).

3. Wat is een cascade type?

In de annotatie van methodes van entity klasses kan een cascade veld toegevoegd worden. Dit definieert in welke mate een bepaalde actie door gepropageerd wordt naar entity objects dat het bevat.
Wanneer bijvoorbeeld een instantie van de klasse “Order” verwijderd wordt, worden alle LineItems van deze order ook verwijderd als deze geannoteerd zijn als “CASCADE=REMOVE” of “CASCADE=ALL”. Wanneer een order gepersist wordt, zullen alle LineItems ook gepersist worden indien deze geannoteerd zijn met  CASCADE=PERSIST” of “CASCADE=ALL”.

4. Wat is een unbound entity? wat is het verband?

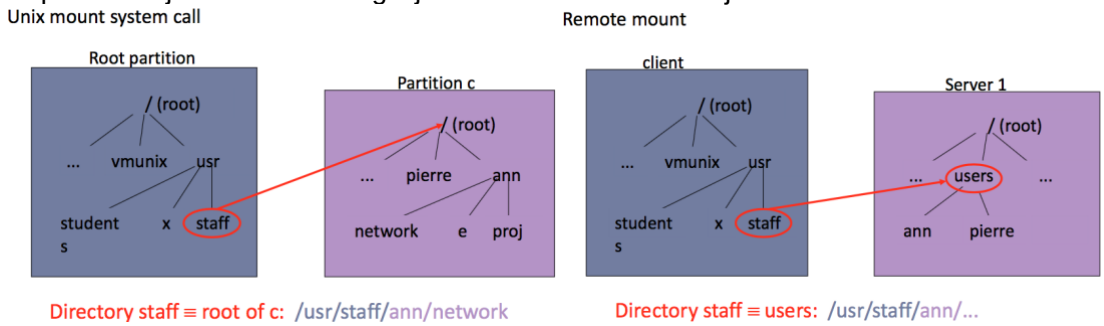
Een entity die normaal gezien geassocieerd is met een andere klasse maar dat niet meer is. Stel we hebben een klasse User en een klasse Email. Een User kan verschillende emailadressen hebben. Stel dat een User verwijderd wordt en er geen Cascade=ALL of Cascade=REMOVE gedefinieerd is. Dan hebben we geest-instanties van de klasse Email, die niet meer gelinkt zijn aan een User (want deze is verwijderd). Dit zijn unbound entities.
Het is dus belangrijk om het Cascade type te definiëren waar nodig.
4 staten life cycle entity instances:

| | new | managed | detached | removed |
|---|---|---|---|---|
| have persistent identity | | x | x | x |
| associated with PC | | x | | x & scheduled for removal from data store |

# Services

1. Bespreek het ontwerp en de belangrijkste ontwerpbeslissingen van NFS

NFS (= Network File System) komt uit de tijd dat workstations geen harde schijf hadden. Men beschikte over een centraal, gedeeld opslagmedium.
NFS wil de UNIX file system interface emuleren, is stateless en OS onafhankelijk. De client en server modules zijn kunnen in elke node zitten, NFS heeft een geïntegreerde file/directory service en integratie van een remote file system in een lokaal FS: mount -> remote mount: Gebruikt een gepartitioneerde disk, elke partitie bevat een hiërarchisch FS en heeft een naam, de partities zijn zo aan elkaar gelijmd dat ze onzichtbaar zijn voor de users.




Elke server bevat een lijst met alle file systems die op afstand gemount kunnen worden door de clients in /etc/exports + access list. De client heeft de filesystems om te mounten (@start-up) in /etc/rc:

- **Hard-mount** : Het client process wordt gesuspend en gaat wachten tot een request naar een remote file slaagt. In het geval van een server failure zal het programma niet 'gracefully' kunnen recoveren.
- **Soft-mount** : Er wordt een fout teruggegeven als de request niet slaagt na x aantal pogingen en het programma handlet deze exception => breekt Unix failure semantics/ lowers failure transparancy

Automounter (client-side): lege mount points, mount word pas gedaan bij eerste request naar remote file. Fungeert als een server voor de local client, gets references naar lege mount points, mapt de mount points naar de remote file systems en mount het FS op het mount point via symbolic link (om redundant requests naar de automounter te vermijden).
VFS module (=virtual file system) toegevoegd aan de kernel om local en remote files te onderscheiden. Zorgt voor access transparency: redirects alle file-gerelateerde system calls naar het juiste FS. NFS gebruikt file handles als identifier.
Een file handle bestaat uit:

- **Basis:** Inode: file ID in partitie op UNIX system. Het getal wordt dus gebruikt om het bestand te identificeren en lokaliseren in het file system waarin het zich bevindt.
- **Extended** met: Filesystem identifier
- **En** Inode generation number (om een inode nummer te kunnen hergebruiken nadat een bestand verwijderd is)

NFS integreert de client module in de kernel (=> client niet hercompilen/herladen) bied de standaard UNIX-interface en werkt samen met de VFS-module. Deze bevat ook de encryption key die gebruikt wordt om gebruikers te identificeren. Eén client module voor alle user-level processes.
Directory service: naam-resolutie gebeurt in de client, stap-voor-stap process voor multi-part file-names, mapping tables in de server (=> overhead gereduceerd mbv caching)
Access control en authenticatie: gebaseerd op UNIX user ID en group ID, gecheckt bij elk NFS request, secure (?) dankzij gebruik DES.
NFS heeft server caching, cached de files in zijn local VFS:

- **Write-through** : Data word in het geheugen opgeslagen en meteen geschreven naar de schijf voordat de operatie wordt bevestigd aan de client. Tradeoff: lower performance vs better failure semantics. (oude versies NFS)
- **Delayed write** : Data word in de server cache opgeslagen, waarna de client bevestiging krijgt. Deze data wordt dan naar de schijf weggeschreven wanneer een commit-operatie is ontvangen voor deze specifieke file => betere performance. (nieuwere bieden deze mogelijkheid aan)

En client caching: cached resultaten van de read(), readdir(), write(), getattr(), lookup() operaties. Gebruikt read ahead = anticipeer read access en delayed write: block van file word gefetched, geüpdate en dan als dirty gemarkeerd. Dirty pages van files worden asynchroon naar server geflusht, dirty pages van directories geflusht zonder delay. Lage consistency want meerdere kopieën van zelfde data op verschillende NFS clients kunnen tegelijk bewerkt worden.

Clients zijn zelf verantwoordelijk om regelmatig de server te pollen om hun cache te valideren via consistency checks, deze zijn gebaseerd op timestamps die de laatste aanpassing aanduiden. Gebeuren vaak (bv bij het openen van een bestand, block fetch, periodisch) en zijn zeer kostelijk.
Alle operaties zijn idempotent => ok voor als server crasht
Conclusie: redelijke performance (remote files op snelle disk > local files op trage disk), maar cache validation en performance van writes zijn nadelen.

## 2. Bespreek het ontwerp en de belangrijkste ontwerpbeslissingen van AFS

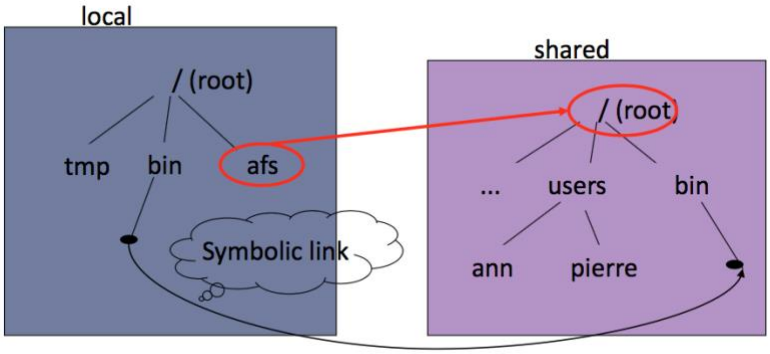
AFS (=Andrew File System) combineert het beste van personal computers en time-sharing systems. Gebaseerd op feit dat: files meestal klein zijn, vaker read dan write, sequentiële acces dan random, meeste files niet shared en als shared maar 1 user die bewerkt. Assumpties over environment: veilig FS, publieke workstations met local geheugen en geen privéfiles, principes:

- **Whole-file sharing** : Volledige mappen en bestanden worden naar client-computers overgebracht van AFS-servers (stateful servers).
- **Whole-file caching** : Na deze transfer wordt een kopie van deze files bijgehouden in de cache op de lokale schijf (persistent) zodat deze een selectie van meest gebruikte bestanden bevat. Dit zorgt voor een vermindering van de load op de servers en minimaliseert effect van network latency.
- Onderscheid tussen file en directory service

Er zijn dus ook 2 soorten bestanden:

- **Shared files** : Opgeslagen op de server, met eventueel gecachte exemplaren in de lokale cache.
- **Local files** : tijdelijke bestanden of system files nodig om op te starten.

Eén globale name space. Shared files worden geïmplementeerd als symbolic links naar de shared space. Elke server houdt een replicated location database bij.



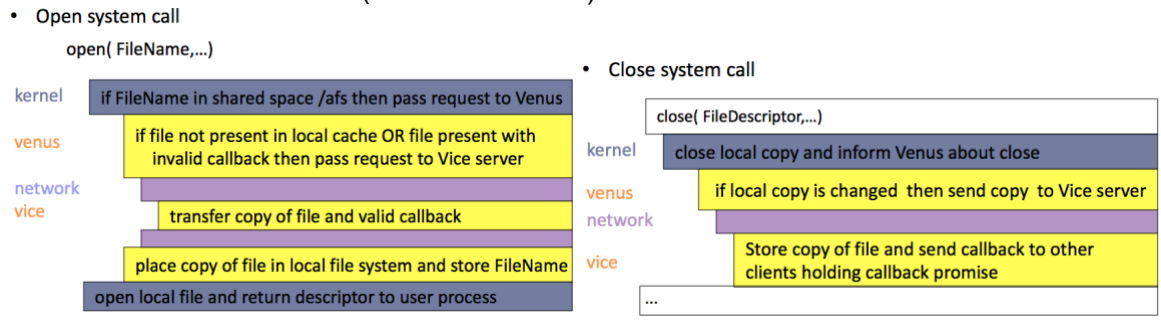

Directory afs ≡ root of shared file system

Implementatie mbv 2 softwarecomponenten:

- **Vice** = file server. Server-software die draait in user space. Begrijpt enkel file identifiers. Beveiligd systeem, beheerd door system manager.
- **Venus** : De client-software. Er is dus 1 venus process per workstation om autonomie te bewaren die de directory services implementeert.

Om dit alles te laten werken is de kernel van elk workstation aangepast om open(file), close(file) (en een aantal andere) systemcalls te onderscheppen en door te sturen naar Venus, als ze wijzen naar shared files. Read en write worden enkel door UNIX kernel van workstations gedaan.

Caching: wanneer Vice een file aanbiedt aan een Venus-process, zorgt deze ook voor een callback promise. Dit garandeert dat het Venus-process van alle clients die de files cachen verwittigd wordt als het bestand geüpdate word.
Als een Venus-process nu zo'n callback ontvangt zet het de token van het relevante bestand op 'cancelled' om aan te geven dat het gecachte bestand niet meer up to date is. Wanneer het Venus-process een open(file) request afhandelt, controleert het eerst of het bestand in de cache staat én of de token 'valid' is (en niet 'cancelled).




Bij een crash of reboot client, blijven de bestanden op de schijf staan maar er kunnen callbacks gemist worden. Venus genereert voor dit scenario een cache validation request en zet het token terug op de juiste waarde.
Callback promises worden hernieuwd bij open of wanneer er geen communicatie geweest is met de server voor T seconden. Mogelijk dat 2 kopieën op verschillende stations tegelijk bewerkt worden => enkel updates laatste close worden bewaard.
Replicatie-ondersteuning voor read-only files, 1 master voor uitzonderlijke updates, manuele procedure om veranderingen door te propageren naar andere volumes.
Conclusie: goeie performance, enkel delay eerste open van file (voor dit specifiek use pattern).

3. Bespreek de invloed van de volgende ontwerpbeslissingen bij AFS op het vlak van performantie en vergelijk met NFS:
    a. caching

NFS
**Client caching**: clients cachen de resultaten van read, write, getattr, lookup, readdir en doen aan read-ahead (nabijgelegen blokken mee sturen) en delayed-write.
**Server caching:** cached de files in zijn lokale VFS module, gebaseerd op standaard UNIX caching methodes: write-through (niet echt caching) en delayed write. Uitleg zie hierboven

AFS, enkel caching aan client-side. Doet aan whole-file caching. Hierdoor zijn er veel recent gebruikte bestanden lokaal opgeslagen op disk (persistent). Enkel vertraging bij eerste Open(). Dit zorgt er voor dat deze grote cache aangesproken kan worden bij een open() request. Dit zorgt voor een vermindering van de load op de servers en minimaliseert effect van network latency => betere performance dan NFS bij **kleine** bestanden.
AFS maakt niet expliciet gebruik van een server cache maar gaat via de callback promises een file in alle client caches markeren als niet meer valid, als ze ergens wordt gewijzigd.

    b. cache-validatie aan de client side

Bij NFS gebeuren consistency checks bij openen file, fetchen nieuwe block waarbij de timestamp van de laatste aanpassing van het lokale bestand vergeleken worden met die van de server. Dit is zeer kostelijk.
Bij AFS adhv callback promise, wanneer Vice een file aanbiedt aan een Venus-process, zorgt deze ook voor een callback promise. Dit garandeert dat het Venus-process van alle clients die de file cachen verwittigd wordt als het bestand geüpdate word.
De server notified dus de client, en de client moet dus geen "arbeid" verrichten. Enkel indien hij offline is geweest of wanneer er problemen zijn opgetreden moet hij een cache validation request sturen. AFS heeft hier dus ook het voordeel.

    c. consistency

NFS: verlaagde consistency door delayed write.

AFS: mogelijk dat 2 kopieën op verschillende stations tegelijk bewerkt worden => enkel updates laatste close worden bewaard => designed voor use case dat shared files meestal maar door 1 client bewerkt word.

d. implementatie van de directory service.

Bij NFS wordt de *name resolution* gecoördineerd door de client. Dit zorgt voor veel overhead die eventueel wel iets verminderd kan worden door caching. Het wordt gedaan door de VFS module in de kernel.
Door de whole-file caching van AFS, staan veel bestanden lokaal, waardoor voor deze meest gebruikte bestanden de *UNIX directory service* gebruikt kan worden. De aangepaste kernel zal dus enkel operaties intercepteren (en doorsturen naar venus) naar bestanden die refereren naar een shared file. Ook hier heeft AFS dus het voordeel.

4. Bespreek de ontwerpbeslissingen bij AFS en NFS om beschikbaarheid te garanderen als:

A. Client faalt

NFS
Als een client faalt is dit niet zo een groot probleem vermits de bestanden gecentraliseerd zijn. Ze zijn dus nog steeds beschikbaar op de server tenzij er lokale writes waren die nog niet door gepropageerd waren naar de server (dit gebeurt niet per se onmiddellijk na de write).
AFS
Venus zal de veranderingen propageren na het oproepen van close(). De client die gefaald heeft zal wel geen notifications ontvangen van de callback promise. Dit vormt niet echt een probleem vermits deze, wanneer hij terug online komt, een cache validation request stuurt naar de server. File bewerkingen worden lokaal gecashed op disk dus persistent, indien in tussentijd file niet bewerkt is door iemand anders.

B. Server faalt

NFS
Wanneer de server faalt zal een NFS-client enkel nog de laatste gecachete reads/writes/getattrs/lookups en readdirs hebben. De client zal geen toegang meer hebben tot zijn remote filesystems. Kan voor problemen zorgen aangezien door de location transparantie een applicatie geen onderscheid maakt tussen local en remote files => applicatie freezes of program handelt exception afhankelijk van hard of soft mount.
Server is stateless dus client blijft de operaties proberen te verzenden tot de server reboots. Is ok want alle operaties zijn idempotent (= zelfde resultaat als operatie 2* gedaan word, zoals read, write, tegenvoorbeeld: append)
AFS
AFS doet aan whole-file caching. Na de transfer van bestanden van de server, worden deze op de lokale schijf opgeslagen. De client cache bevat dus honderden meest gebruikte bestanden. Indien de server offline gaat, kan de client dus nog wel aan zijn meest gebruikte bestanden. Aanpassingen hieraan zullen dan naar de server door gepropageerd worden als deze terug online is.

C. Communicatie faalt (focus hierbij ook zeker op de cache-validatietechnieken)

NFS
Wanneer een write niet doorgegeven wordt naar de server (door bijvoorbeeld een communicatiefout), zijn er verschillende versies van hetzelfde bestand in omloop. Om dit tegen te gaan bestaan er consistency checks.
Consistency check wordt geïmplementeerd door de timestamp van de lokale versie te vergelijken met die van de versie op de server. Een gecachte file is in orde wanneer (T-Tc) < t of wanneer (T-Tc) > t en (Tm_client = Tm_server) met T= huidige tijd, Tc = tijd op moment dat de cache entry het laatst gevalideerd was, t = “freshness interval”, Tm_x = tijd op moment dat de block het laatst aangepast was op X.
AFS

Wanneer een bestand toegekend wordt aan een venus-proces, gaat Vice steeds een callback promise afleveren. Hiermee garandeert Vice dat dat Venus genotified zal worden indien een ander proces het bestand aanpast. Stel nu dat deze notification verloren gaat door een communicatiefout. In AFS is het zo dat de callback promise steeds vernie uwd moet worden voor een open() operatie als er al een tijd T verlopen is sinds de file gecached is. De client zal in dit geval dus zelf een cache validation request sturen.

# Indirecte Communicatie

5. Beschrijf bondig alle mogelijkheden van indirecte communicatie
    a. Het basisprincipe
    b. De api voor ontwikkelaars
    c. De architectuurelementen
    d. Geef voor elk van deze vormen een (één) scenario waarin dit toegepast kan worden in een systeem voor autoverhuur.

**Indirecte communicatie** = communicatie tussen entiteiten in een DS **via een tussenpersoon**, zodat zender en ontvanger(s) niet rechtstreeks aan elkaar gekoppeld zijn.
De ontkoppeling kan via:

- **space-uncoupling**: de zender moet de identiteit van de ontvanger niet kennen
- **time-uncoupling**: de zender en ontvanger hoeven niet gelijktijdig te bestaan

Indirecte communicatie kan in een gedistribueerd systeem op 5 verschillende manieren worden toegepast:

- group communication (space-uncoupled)
- publish-subscribe (time-uncoupled & space-uncoupled)
- message queues (time-uncoupled & space-uncoupled)
- distributed shared memory (time-uncoupled & space-uncoupled)
- tuple space (time-uncoupled & space-uncoupled)

**Group Communication**
Een bericht dat naar een group wordt verzonden, word aan al zijn leden bezorgt (one-to-many semantiek). Het is dus een vorm van multicast communicatie. Voorziet group membership management service, deze heeft 4 hoofdtaken (er moet niet aan alle voldaan worden): voorzien van een IF voor group management veranderingen (om groepen aan te maken en te verwijderen en leden toe te voegen en te verwijderen), failure detection (crashed of unreachable), verwittiging van de leden bij wijzigingen van leden van een group en group address expansion (het coördineren van de verspreiding van een multicast message binnen een group).
Kan reliability en ordering ondersteunen, vb implementatie: IP-multicast, is vorm van :

- **Betrouwbare multicast** garandeert het bezorgen van hoogstens één (**integriteit**), uiteindelijk (**validiteit**) en aan alle leden of geen enkele (**overeenkomst**)
- **Geordende multicast** garandeert het behouden van: de zender's volgorde (**FIFO**), "happens before relation" **causale ordening** en **totale ordening** (= aan alle leden in dezelfde volgorde geleverd)

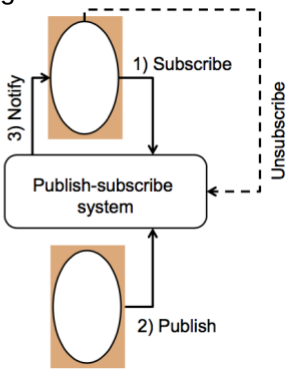
Mogelijke eigenschappen van een group:

- Closed vs open: closed = als enkel leden mogen multicasten naar de group.
- (Non-)overlapping: non-overlapping = leden hoogstens lid van 1 group
- Synchronous vs asynchronous
- Process vs object group: process group = groups met processes als leden, low-level, delivery eindigt wanneer het process het ontvangt. Object group = leden zijn objecten die berichten ontvangen = high-level.

Voorbeeld: group van alle CarRentalCompanies die berichten ontvangt ivm vakantie/sluitingsdagen.

**Publish-subscribe systems**
Zender = publisher, ontvangers = subscribers

1. Receivers subscribe using event-filters
2. Sender publishes events
3. Pub-sub system notifies interested subscribers
4. Receivers may unsubscribe for particular event-filter




Een PSS helpt om om te gaan met heterogeniteit en asynchroniteit (door time-uncoupling).
De API biedt volgende methodes: subscribe(), publish(), notify(), unsubscribe(), advertise() en deadvertise().
Filters in een PSS:

- Channel-based: Ontvang alles van een channel
- Topic-based: Ontvang alles van een bepaald topic of interest
- Content-based: Ontvang alles dat voldoet aan een query over alle attributen
- Type-based: Ontvang alles van een bepaald object-type.
- Objects of interest: Notify state changes
- Context-based: bv rekening houden met locatie ve subscriber
- Complex event processing: algoritme als filter

Potentiële vereisten: security, QoS, scalability en failure handling
Een PSS kan gecentraliseerd geïmplementeerd worden (1 server is event broker), maar dit vormt een bottleneck en een verlies aan schaalbaarheid. Een betere manier is opteren voor een gedistribueerde (p2p) aanpak. Elke node is dan zowel een publisher als subscriber als event broker. Andere alternatieve implementaties:

- Flooding: Zend bericht naar alle nodes, nodes moet beslissen of het belangrijk is.
- Rendezvous: Verdeel het broker network in groepen en laat elke group verantwoordelijk zijn voor een deel van de events, subs en pubs van dat deel worden naar die broker groep verzonden.

Voorbeeld: users die in buurt van 10km van agency wonen krijgen notificatie als nieuw type huurauto beschikbaar is.

**Message queues**

Een message queue is een simpele datastructuur die berichten kan opslaan. Een sender kan een bericht versturen naar een MQ, een receiver kan berichten ophalen van een MQ.
Het is dus point-to-point communication/one-to-one semantiek. Het verschilt met normale message passing in het feit dat MQ's werken met expliciete third-party queues, afgezonderd van de sender en de receiver. Ze zijn ook steeds persistent tot consumed (reliable delivery), waarborgen validiteit (berichten worden uiteindelijk ontvangen) en integriteit (ze worden exact 1 keer ontvangen en ze zijn identiek aan het verzonden bericht). Kan transacties ondersteunen. MQ Message bestaat uit: bestemming, meta-data en body. Vooral gebruikt in grote middleware om verschillende applicaties binnen een bedrijf te integreren = Enterprise Application Integration (EAI).

Verschillende ontvangmanieren: blocking receive: wachten op een bericht, non-blocking receive: polling (= check vd status vd queue) en receiver notification: ..when msg is available.
Er zijn verschillende queue access strategies: FIFO, priority queues, select messages: berichten kunnen ook geselecteerd worden op basis van hun attributen/eigenschappen.
De API biedt: send(), receive() (blocking), en evt poll() (bij non-blocking) en notify()
Voorbeeld: user stuurt message naar agency, agency messaged 1 voor 1 naar companies tot succesvolle reservatie, messaged confirmation naar user.

**Shared memory approaches**
**Distributed shared memory** = een abstractie die gebruikt wordt voor het delen van data tussen computer nodes die geen fysiek geheugen met elkaar delen. Processen lezen/schrijven van/naar DSM alsof het normaal geheugen in hun address space zou zijn. Het is zeer handig voor parallelle of gedistribueerde applicaties die direct de updates van andere processen moet kunnen zien (en dus niet in een client-server setting).
Elke node heeft een lokale kopie van de DSM, waarbij message passing gebruikt word voor updates van het geheugen te sturen.
Message passing vs. DSM:
DSM is vergelijkbaar met message passing aangezien de toepassing op parallel processing het gebruikt van asynchrone communcatie met zich mee brengt.

- Pro DSM
    - Makkelijke toegang (gedeelde variabelen, geen marshalling, geen extra communicatie)
    - Kan persistent zijn: time-uncoupled processen kunnen dus gebruik maken van dezelfde gegevens (bij communicatie via MP moeten gelijktijdig bestaan)
- Pro MP
    - Private address space: processen zijn “protected” van elkaar
    - Marshalling neemt het verschil in representatie voor zijn rekening indien message passing gebruikt wordt in twee heterogene systemen

Voorbeeld: alle users hebben een lokale kopie van de beschikbaarheden

**Tuple space communication**
Tuple = een opeenvolging van één of meer data fields.
Een tuple space is een gedeelde collectie van tuples die processen kunnen gebruiken om data uit te wisselen. Tuples kunnen gelezen en geschreven worden van gedeelde TS. Het zorgt voor space- en time-uncoupling.
Het lezen van tuples gebeurt door pattern matching en is steeds blocking. Tuples zijn ook steeds immutable (kunnen niet rechtstreeks aangepast worden).
De API biedt volgende operaties:

- write(): tuples in de TS plaatsen
- read(): de waarde van 1 tuple teruggeven
- take(): de waarde van 1 tuple teruggeven en de tuple verwijderen uit de TS

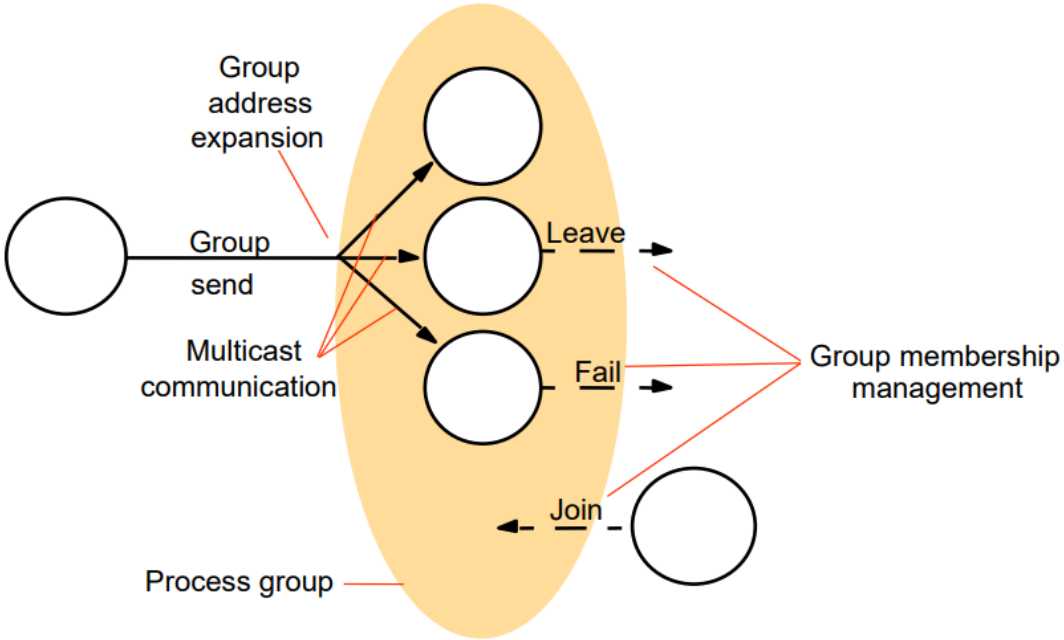
Alternatieve implementaties:

- Meer dan 1 TS gebruiken (scoping)
- Centralized vs distributed
- Tuples die andere tuples als waarde bevatten
- Type tuples: vergelijkbaar met objecten met attributen

Voorbeeld: users writen een reservering naar de TS, de eerste carrentalcompany die deze reservering kan vervullen take’d het van de TS.

## 6. Welke vorm van indirecte communicatie wordt in Coda gebruikt? Bespreek + figuur

Group communication => Message directed to a group is delivered to all its members
Provides group membership management
May support: reliability, ordering, and more




Om kopieën van aangepaste bestanden te verzenden naar de servers in de AVSG (multicast file naar AVSG).
Voor het vergelijken van de CVV van een gedownload bestand van een bepaalde server met de CVV van dat bestand op andere servers in de AVSG.
Voor de cache coherence: Venus moet binnen T seconden relevante gebeurtenissen detecteren. Venus stuurt daarom elke T seconden een probe message naar al de servers in VSGs van de files die het in zijn cache heeft. Een antwoord wordt en kel ontvangen van servers die actief zijn, als er een server antwoordt die eerder inactief was, wordt de AVSG geüpdatet met die nieuwe server. Het dropt ook alle callback promises van elke file dat het bevat van het relevant volume: het kan namelijk zo zijn dat de gecachte versie niet meer de laatste versie is in de nieuwe AVSG. Als het geen antwoord krijgt van eerder wel actieve servers, wordt de AVSG verkleind. Hier moeten de callback promises enkel gedropt worden als het over een geprefereerde server gaat.

# Cloud

1. Cloud
    a. Geef alle vormen/soorten van cloud services.

- **IaaS** (Infrastructure as a service): virtual machines on shared hardware , storage, virtual networks between machines
- **PaaS** (Platform as a service): middleware as a service without burden of the VM underneath, scalability first, development and web-hosting platform (php, .net, java)
- **SaaS** (Software as a service): Zero-install, online applications (CRM = Customer Relationship Management, Google Mail, Google Apps)

    b. Maak een bondig lijstje van andere leveranciers zoals Amazon.

- **IaaS**: Public cloud (accessible by anyone from anywhere) Amazon web services, Ms Azure. Private cloud: private data centers for companies: openstack, xen cloud platform
- **PaaS**: Windows Azure, Openshift, Google App Engine
- Serverless model PaaS: let the cloud provider handle automatic scaling of infrastructure and the management of the software stack => focus on app logic and deploy it
- **SaaS**: Google Mail, Google Apps, salesForce

    c. Geef de voordelen van werken in de cloud.

- Sneller deployment van nieuwe applicaties
- Mogelijke IT kostenbesparingen
- Higher degree of distribution

=> high availability and performance through replication,

- Multi-tenancy

Dynamically assigning the available **shared resources** to multiple customer organizations => economies of scale

- Delivery as a service

**=>** by outsourcing you don't need to make upfront commitments and investments in infrastructure and only pay-per-use

- Self-service & Elasticity

Preserves **scalability** => rapidly respond to changing service demand and customer requirements (automatically**)**

    d. Stel je wilt een social network site schrijven voor ouderen, welke vorm van cloud service zou je gebruiken en waarom?

Software as a service. Het is het eenvoudigste voor de gebruikers als ze de site en toebehoren gewoon kunnen gebruiken vanuit hun browser zonder iets te moeten installeren.

2. IOT: waarom wordt er in het algemeen geen gebruik gemaakt van TCP in IoT-applicaties?

- Meeste IoT-devices spenderen het grootste deel van hun tijd in *sleep modus* => om energie te besparen en kunnen dus geen langdurige TCP connecties staande houden => en verbindingsprotocol is duur (3-way-handshake) dus vaak connecteren zou veel energie kosten
- Veel IoT-applicatie communicatie gaat over kleine hoeveelheden data waardoor de overhead van het opstarten van een TCP connectie overdreven is.
- Veel IoT-applicaties hebben de vereiste dat communicatie een heel korte latency heeft, TCP handshaking zou toevoegen aan de latency.

3. Waar hoort Google App Engine bij: IaaS platformen, PaaS platformen of SaaS platformen en leg uit waarom ?

PaaS, targets traditional web applications, by providing the platform for development and hosting. Enforces a clean separation between a stateless computation tier and a stateful storage tier. Optimized for applications with short-lived requests. Automatic scaling and load balancing. Google App Engine also targets web applications using JSP and Servlets. It provides more than just the infrastructure (IaaS) and less than a full application:
OS = IaaS

OS + middleware = PaaS
OS + middleware + application = SaaS

4. Wat zijn de mogelijkheden met Google App Engine

Google App Engine has the advantage that it should make the applications highly scalable and available. Offers data services: DataStore, memcache, Google Cloud SQL and Blobstore. Also offers a variety of common services such as Mail, Authentication, Fetch URL, Task Queue (Pull & Push), … Supported languages: Python, Go, PHP, Java.

5. Stel je hebt een ontwerp in JEE:

   a. Wat zijn de mogelijkheden voor persistentie in JEE?

*JEE*

In JEE gebruiken we JPA (Java Persistence API): map elke business entity met een matching Java Persistence API Entity.

- Een entity is een business object in een persistent storage mechanism (definieer primary key en multiplicity)
- Persistent fields of properties
- Persistence context (alle entities in de context worden gepersist op dezelfde manier)
  a. EntityManager om te te communiceren met persistence context
- JPQL

   b. Wat zijn de mogelijkheden voor persistentie in GAE?

*GAE*

- Gebruikt het NOQSL storage systeem (<=> RDBMS)
  a. Geen vaste tabellen (schemeless)
  b. Scalet horizontaal (nodes toevoegen)
  c. Zwakke consistency garanties (**geen ACID**)
- Data voorgesteld door entities: bevat fields, primaire key, en worden gemapt op een enkele Bigtable rij. Entities van dezelfde soort kunnen een verschillend schema hebben.
- **Hierarchical Persistence Model**
  a. Entity group := 1 root entity, heeft veel child entities die root in hun primary key hebben
  b. Query-types zijn gelimiteerd (geen many-to-many, geen join, geen aggregaties, geen polymorfe queries) Dus: de meer geavanceerde JPQL-queries van JEE werken niet
  c. Geen container-managed entity manager
  d. Manueel openen en sluiten van EntityManager instantie ( **! lazy loading** )
- Transactie (optioneel)
  a. Enkel op entities in zelfde EntityGroup
  b. Optimistische concurrency: meerdere concurrent transacties op een
  c. EntityGroup, de eerste zal slagen, de rest falen
  d. CrossGroupTransactions: op meerdere entity groups (max 25)
- Geen JAVA EE/JTA support
- **Geen** container-managed (global) transactions
- **Geen** kennis over EntityGroup

   c. Wat zijn de verschillen tussen beiden?

*JEE*

Focus on multi-tier enterprise applications

- Enterprise Java Beans (EJBs)
- Persistence (JPA)
- Web technologies: Java Servlets, Server Pages, Web Services (SOAP & REST)...
- Distributed transactions

Uses SQL and scales vertical (scale by adding resources per node)

*GAE*
Focus on scalable web applications
- No dedicated business tier technology => No support for Enterprise Java Beans (EJB)
- Persistence: JPA and JDO (but limited! e.g less Key type options for entities)
- Web technologies: Java Servlets, Java Server Pages; limited support for web services -> can be used as business tier
- Limited transaction support (transactions for single data item)

Uses App Engine DataStore (NoSQL based, schemaless: no fixed tables), scales horizontally (scale by adding nodes) => offers weak consistency, no complete ACID (eventual consistency).

d. Wat moet je veranderen in je JEE practicum als je dit wilt aanpassen naar GAE?

De omzetting zal op zich niet zo moeilijk zijn maar er zou wel wat aandacht moeten besteed worden aan een aantal zaken:
- Hierarchical persistence model = 1 root entity
- Queries worden geregeld via pre-built indexes => advanced JPQL queries (geen many-to-many relations, geen join, geen aggregate of polymorphic queries)
- Restricties binnen transacties
    a. Alleen binnen zelfde entity group, hoogstens op 1 root
    b. Optimistic concurrency (geen locks, eerste transactie die commit slaagt, andere falen)
    c. Cross-Group(XG) transactions ondersteund maar maximum 25 en hogere latency
- Geen container-managed (global) transactions
- Geen support voor EJB
- Entity Group tree-structuur (geen ManyToMany) => wijziging in de klasse relaties
- Specifieke regels primary key: **Key** bevat appID, soort en entity ID, root van entity groep (Long, String, Key, Key als encoded String), child(Key (inclusief key root), Key als encoded String). Ook GenerationType.AUTO vervangen door GenerationType.IDENTITY
- Geen container-managed entity manager => manueel openen en sluiten van EntityManager instantie ( **! lazy loading** )

e. Is dit bij andere cloud platformen ook zo? Waarom?

Deze invloed komt voornamelijk als gevolg van de DataStore. De datastore heeft als doel een hoge availability en scalability the behalen en low latency. De werking van de datastore zorgt echter voor enkele restricties, zoals de restrictie op bepaalde queries. Als het cloud platform een traditioneel SQL gebaseerde opslag voorziet kan het dus zijn dat er minder aanpassingen nodig zijn.

f. Geeft twee voorbeelden

g. Leg GAE Message Queues uit

Asynchronous inter-process/-thread communication: gebruik queue om controle of inhoud door te geven (<=> request-reply). Waarom in cloud computing? verbeteren performance en scalability en om bottleneck activiteiten te ontkoppelen naar “worker processes” => asynchrone executie (geen delay voor user) en mogelijkheid om executie te parallelliseren en scalen
E.g. GAE Task Queues
2 queue configuraties: - *Push queue* (default): queue middleware die task naar worker pusht => automatic scaling van processing capaciteit, verwijdert tasks automatisch na processing, geen config nodig
- *Pull queue:* worker pulls message van de queue => meer controle over wanneer te tasks te processen, manuele scaling en deleting (door app) en ondersteunt integratie met non-app-engine code, config nodig

Task = unit werk die door app uitgevoerd moet worden, bevat eindpunt (=url) van een worker (=servlet), optioneel: (unieke) naam
2 soorten workers: default die url gebruikt (push queue), backends (geen restrictie maar kost extra) (pull queue)

## Creating and processing tasks:

- Steps:
  1. **Create** task and insert in queue
  2. Workers (i.e. task consumer) **lease** tasks:
     - When task is leased, then unavailable for other workers until lease expires
  3. Workers **process** leased tasks
     - If success => delete task from queue
     - If lease expires => stop task
     - If failure before lease expires => retry (max. # retries in config)
- Different depending on queue type:
  - Push queues:
    - automatically by App Engine (lease = 10 minutes)
  - Pull queues:
    - programmatically=> pull queue when worker is available