# HIGH AVAILABILITY CONCEPTS

JOHAN LOECKX

Smals SECTIE ONDERZOEK 01/2012 • RESEARCH NOTE • 25

***Abstract***

Met de komst van de 24/7 interneteconomie is zogenaamde « High Availability » hoog op de agenda komen te staan. Het bouwen van zulke systemen is echter niet eenvoudig en moet vanaf de eerste fase in een ontwikkelproject (requirements analyse) tot de laatste (onderhoud) in rekening gebracht worden.

De meest bepalende fase hierin is het ontwerp van de software-architectuur. Cruciaal hierbij is het besef dat het onmogelijk is voor systemen om *op elk ogenblik* over de meest up-to-date informatie te kunnen bezitten als men hoge beschikbaarheid verkiest.

Hoewel dit onoverkomelijk en dramatisch lijkt, kan men dit euvel in de praktijk vaak verhelpen. Deze Research Note gaat dieper in op de stappen die uitgevoerd kunnen worden om de beschikbaarheid van systemen aanzienlijk te verhogen; niet alleen tijdens development, maar ook op het vlak van infrastructuur en governance.

Gezien de breedte van het onderwerp zal de bespreking beperkt blijven tot een high-level overzicht.

***Résumé***

Avec l'arrivée de l'économie internet 24/7, la « high availability » a considérablement gagné en importance. Il n'est toutefois pas évident de construire de tels systèmes, de sorte que dans un projet de développement, une attention particulière s'impose depuis la première phase (analyse des exigences) jusqu'à la dernière phase (maintenance).

La phase la plus décisive est ici la conception de l'architecture logicielle. Il faut absolument tenir compte du fait que lorsque l'on opte pour une haute disponibilité, les systèmes ne peuvent pas disposer *en permanence* des informations les plus actuelles.

Aussi insurmontable et dramatique que cela puisse paraître, il est souvent possible de s'en affranchir dans la pratique. Cette Research Note traite plus en détail des démarches à suivre pour augmenter sensiblement la disponibilité des systèmes, tant durant le développement qu'en ce qui concerne l'infrastructure et la gouvernance.

Le sujet étant vaste, sa présentation se limite à un aperçu circonstancié.

# 1. Management Summary

Centraal in het debat rond High Availability staat het zogenaamde “CAP-theorema” dat stelt dat de consistency van gegevens en beschikbaarheid niet beide kunnen gegarandeerd worden en bijgevolg strikt tegen elkaar afgewogen moeten worden. Concreet komt het erop neer dat systemen niet *op elk ogenblik* over de meest up-to-date informatie kunnen bezitten als men hoge beschikbaarheid vereist.

Hoewel dit onoverkomelijk en dramatisch lijkt, kan men dit euvel in de praktijk vaak verhelpen op voorwaarde dat men hier reeds tijdens de requirements analyse rekening mee houdt. De levensduur en vluchtigheid van de gegevens moet in kaart gebracht worden evenals de eisen op vlak van beschikbaarheid voor elke use-case, in plaats van voor het volledige systeem.

Als dit proces rigoureus uitgevoerd wordt, kan tijdens de architectuurfase de beschikbaarheid aanzienlijk verhoogd worden met een factor 40x (van 90 tot 99.75 %). De uiteindelijke beschikbaarheid wordt bepaald door de kans op falen (menselijk, configuratie, hardware, software), maar ook door de tijd die nodig is voor de detectie en het herstellen van het incident. Om deze reden is een goed geoliede organisatie essentieel.

In deze nota worden aanbevelingen geformuleerd ter verhoging van de beschikbaarheid tijdens requirements analyse, wanneer de architectuur wordt uitgetekend, en tijdens development op infrastructuurvlak. Ook wordt het belang van transparantie, governance en automatisering aangetoond. Gezien de breedte van het onderwerp, zal de bespreking beperkt blijven tot een high-level overzicht.

*“High availability cannot be achieved by merely installing failover software and walking away (… Instead,) build your systems so that they never have to failover”,*

Evan Marcus & Hal Stern, “Blueprints for High Availability”

# Inhoudstafel


**1. Management Summary .......... 2**

**2. Inleiding .......... 4**

2.1. Wat betekent Availability? .......... 4

2.1.1. Hoge beschikbaarheid .......... 4

2.1.2. Definitie ifv. up en downtime .......... 4

2.1.3. Oorzaken van downtime .......... 5

2.1.4. Business is de pasmunt (leve de roltrap) .......... 6

2.2. Verhogen van de beschikbaarheid .......... 6

2.2.1. The Availability Index .......... 6

2.2.2. Architectuur, architectuur… .......... 7

2.3. Van 99 ➔ 99.9 %: not quite the same .......... 9

**3. Oorzaken van "Low" Availability .......... 11**

3.1. Kans op falen .......... 11

3.2. Lifecycle of an outage .......... 12

**4. Het CAP-Theorema .......... 13**

4.1. Het theorema .......... 13

4.2. Grafische interpretatie .......... 14

4.2.1. Geval 1: Alles verloopt naar wens .......... 14

4.2.2. Geval 2: Het netwerk is onbeschikbaar .......... 15

4.2.3. Geval 3: Partition Intolerant Systems .......... 15

4.3. The good news: Eventual Consistency .......... 16

**5. Verhogen van de beschikbaarheid .......... 16**

5.1. Requirements analyse .......... 17

5.1.1. Levensduur en vluchtigheid van de gegevens .......... 17

5.1.2. Benodigde beschikbaarheid per use case .......... 17

5.2. Architectuur .......... 18

5.2.1. Technieken .......... 18

5.2.2. Het belang van ontkoppeling .......... 19

5.3. Organisatie .......... 21

5.3.1. Transparantie, governance, automatisering! .......... 21

5.3.2. Voorkomen is beter dan genezen .......... 21

5.3.3. Genezen is beter dan verdoven .......... 21

5.4. Development .......... 22

5.5. Infrastructuur .......... 22

**6. Conclusies .......... 23**

**Referenties .......... 24**

# 2. Inleiding

Er wordt van elektronische systemen verwacht dat ze steeds beschikbaar zijn. Het bereiken van deze zogenaamde “hoge beschikbaarheid” of “High Availability” (HA) is echter niet zo eenvoudig. In dit document worden de basisconcepten van High Availability uitgelegd, de oorzaken verduidelijkt en technieken tot het verhogen van de beschikbaarheid opgelijst.

## 2.1. Wat betekent Availability?

### 2.1.1. Hoge beschikbaarheid

Een service wordt *beschikbaar* (available) genoemd als de service **binnen een bepaalde tijdsspanne een “nuttig antwoord” geeft**. Wat dit concreet betekent, hangt af van de toepassing en de verwachtingen van de klant: een foutmelding kan al dan niet aanvaard worden als een nuttig antwoord.

We spreken van een *hoog beschikbaar* (highly available) systeem als de beschikbaarheid “hoger” is dan de andere systemen. Ook dit is nogal vaag gedefinieerd: wat “hoger” betekent hangt af van de sector (high frequency trading vs. een kantoortoepassing), hoe kritisch de toepassing is, etc.

### 2.1.2. Definitie ifv. up en downtime

Gegeven het service window T, de tijd dat een service aangeboden wordt (bv. 24x7 of 8x5, etc…), en U de tijd dat de service effectief beschikbaar was, definiëren we [1]:

$$Availability = \frac{U}{T}$$

Vaak wordt er gesproken over “het aantal negens” dat voorkomt in het Availability percentage. Elke extra negen komt overeen met 10x hogere beschikbaarheid:

| Downtime/maand (T-U) | Availability |
|---|---|
| 4 minuten 19 seconden | 99.99 % |
| **43 minuten** | **99.9 %** |
| 7u 12m minuten | 99 % |

*Tabel 1: Verband tussen downtime (T-U) en de beschikbaarheid*

Het werken met percentages voor Availability is misleidend en het is dan ook beter te spreken in termen van absolute tijd. Het opkrikken van de Availability

**van 95 % naar 99 %** komt overeen met een **vijf maal hogere beschikbaarheid, en dus niet 4 %** !!!

We geven ter illustratie een overzichtje van percentages die telkens een verdubbeling van de beschikbaarheid voorstellen:

87.2 – 93.6 – 96.8 – 98.4 – 99.2 – 99.6 – 99.8 – 99.9 %

**Dit betekent dat de stap van 99.8 % naar 99.9 % *minstens* even moeilijk is als die van 87.2 % naar 93.6 %!**

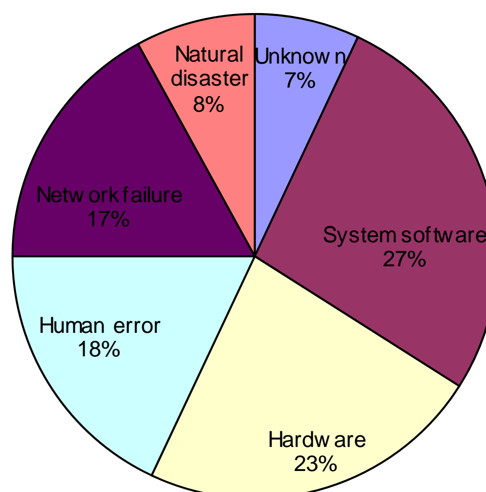
### 2.1.3. Oorzaken van downtime

Als we veronderstellen dat bij elk falen F een downtime U overeenstemt, betekent dit dat er twee manieren zijn om de beschikbaarheid te verhogen:

1. De kans op falen verminderen (minimiseren van N)
2. De service downtime gerelateerd aan een incident verminderen

Het verband tussen het falen van de service en het falen van een individuele component is erg complex. Het is bijvoorbeeld niet zo dat het falen van een component steeds het falen van de service tot gevolg heeft – bv. omdat er redundantie voorzien werd. Anderzijds kan een service falen zonder dat een component faalt, maar ten gevolge van een complexe interactie *tussen* twee componenten. Het verband tussen component- en servicefalen wordt in grote mate bepaald door de architectuur.

Hardware failure zorgt slechts in 23 % van de gevallen voor downtime. Hoewel de cijfers erg afhangen van de sector, applicatie en het instituut dat de analyse uitvoert, komt het belang van de menselijke fouten steeds sterk naar voren. Men identificeert doorgaans zes oorzaken van downtime [2]:




*Figuur 1: Oorzaken van onbeschikbaarheid*

### 2.1.4. Business is de pasmunt (leve de roltrap)

Het is van groot belang hoe men een service definieert. Stel dat we een service willen aanbieden die mensen van de ene naar de andere verdieping brengt. Op hoog niveau hebben we de keuze tussen onder andere **een lift en een roltrap**. Hoewel een lift misschien minder plaats inneemt, is **de roltrap een prachtig voorbeeld van een High Availability System**. Als de motor uitvalt, is de service weliswaar gedegradeerd, maar nog steeds beschikbaar… Als de motor van een lift uitvalt, is de service daarentegen *on*beschikbaar.

> Er blijkt op business niveau bijna steeds een **trade-off te bestaan tussen de functional requirements en de Availability requirements.** Daarom is het van groot belang om High Availability reeds op businessniveau in rekening te brengen.

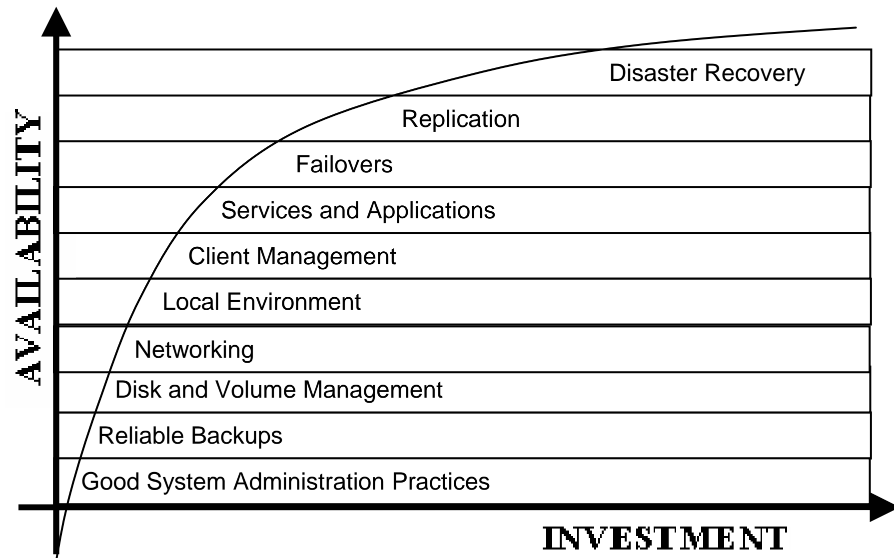
Daalt men te snel af naar het technologisch niveau, dan vermindert de waaier aan mogelijkheden drastisch. In bovenstaand voorbeeld kan men kan nog opteren om een extra motor in de lift te plaatsen, maar daar blijft het bij. *Deze vaststelling blijkt nog meer van toepassing bij complexe systemen.*

## 2.2. Verhogen van de beschikbaarheid

### 2.2.1. The Availability Index

Het verhogen van de beschikbaarheid gaat steeds gepaard met een bepaalde kost en blijft dus steeds afwegen tussen kost en nuttigheid. Evan Marcus & Hal Stern maken in hun boek “Blueprints for High Availability” [2] een zogenaamde “Availability Index” (zie onderstaande figuur) voor het bereiken van hoge beschikbaarheid waarbij vooral twee dingen belangrijk zijn:

- de **kost neemt exponentieel toe** in functie van de beschikbaarheid;
- **er bestaat een volgorde waarin de maatregelen om de hoge beschikbaarheid te bereiken het** best uitgevoerd worden.




*Figuur 2: De "Availability Index" geeft aan welke maatregelen eerst genomen moeten worden. De kost neemt exponentieel toe met de beschikbaarheid.*

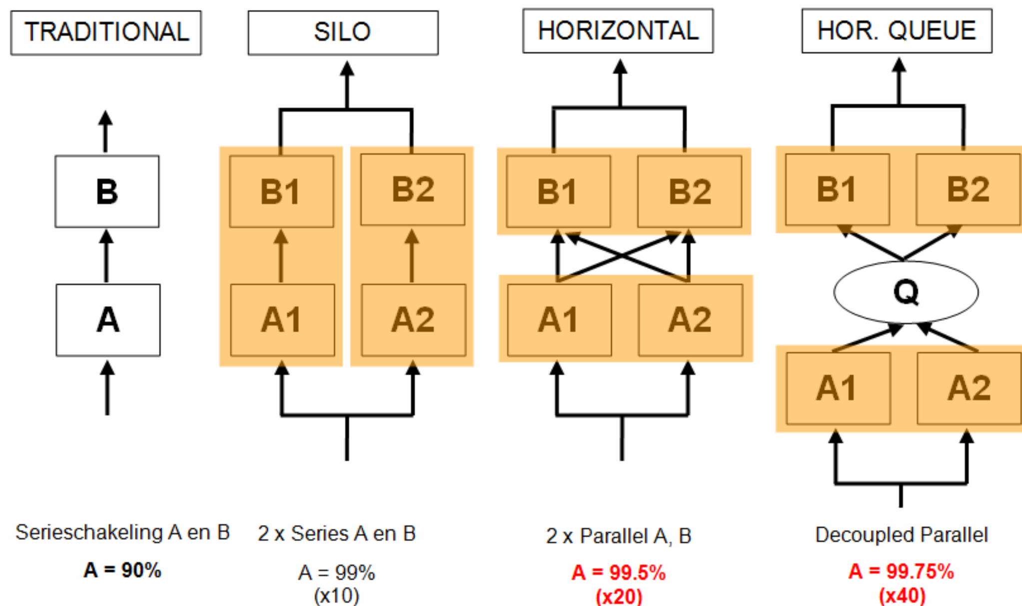
### 2.2.2. Architectuur, architectuur…

Architectuur slaat de brug tussen de business requirements en de implementatie en bepaalt in grote mate de mogelijkheden tot het behalen van High Availability. Zonder dieper in detail te gaan, zullen we drie technieken bespreken.

#### *2.2.2.1. Redundantie (resource pooling)*

Stel dat het proces in onderstaande figuur een douanecontrole op de snelweg voorstelt, bestaande uit twee sequentiële stappen A en B, elk met een Availability van 95 %. Hoewel onderstaande architecturen samengesteld zijn uit bouwblokken met eenzelfde Availability, bezitten ze duidelijk een uiteenlopende beschikbaarheid.

Een **horizontale cloud-like architectuur** ("resource pool") scoort beter dan een traditionele silo-gebaseerde redundantie, maar introduceert ook een grotere complexiteit. Belangrijk echter bij redundantie is zeker zijn dat de redundantie ook effectief *werkt*. Daarom is een architectuur waarbij voortdurend alle componenten **actief** worden gebruikt (horizontale architectuur) veiliger dan een "guard" of "passieve volg- en overname" (silo-architectuur) strategie. Het vervangen van falende onderdelen is in het eerste geval ook geen incident, maar hoort bij het dagelijkse onderhoud.



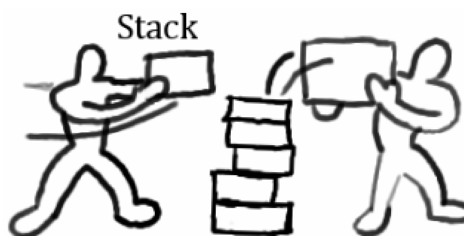

*Figuur 3: Vier mogelijke architecturen, gebaseerd op dezelfde basisblokken A en B.*

### *2.2.2.2. Ontkoppelingen aan de hand van wachtrijen*

Stel dat de architecturen voorgesteld in Figuur 3 betrekking zouden hebben op gegevens omtrent de autoverzekeringen van elke Belg. In dit geval stelt:

- Systeem A de registratie van de gegevens voor in een centrale databank;
- Systeem B de consultatie door agenten ter plekke.

*Daar de verzekerbaarheid maar tot op één dag nauwkeurig bepaald is, moet elke verandering van verzekerbaarheid in systeem A niet meteen doorgekoppeld worden naar systeem B* – zolang het de volgende dag maar in orde is. **Met andere woorden is het niet altijd nodig dat gegevens synchroon verwerkt worden.**




*Figuur 4: Een stack of queue laat toe om twee processen te ontkoppelen aan de hand van asynchrone communicatie.*

**Systeem A en B kunnen dus ontkoppeld worden aan de hand van een wachtrij,** waarbij systeem A de nodige wijzigingen wegschrijft in een wachtrij Q, en systeem B garandeert dat alle wijzigingen vóór middernacht uitgevoerd zijn.

Waar A en B vroeger op een synchrone manier gegevens uit een databank deelden, is dit nu niet langer het geval. Het grote voordeel is dat het falen van systeem A niet meer het falen van systeem B tot gevolg heeft!

> **Het is de business context die bepaalt of systemen A en B ontkoppeld kunnen worden**. **Deze beslissing kan *nooit op technisch niveau tijdens de implementatie genomen worden*.** Requirements moeten *zo vrij mogelijk* neergeschreven worden en de tijdsafhankelijke gegevensverbanden duidelijk in kaart gebracht *in functie van de business.*

De Availability kan nu nog opgevoerd worden doordat er voor de back-end, die communiceert met A, en front-end die met B communiceert, enkel het systeem waarmee ze direct communiceert, beschikbaar moet zijn. Zelfs als Q niet beschikbaar is, blijven de systemen A en B draaien. Er moet enkel gegarandeerd worden dat alle transacties uit A vóór middernacht door B zijn verwerkt.

#### *2.2.2.3. Keep it Simple Stupid (KISS)*

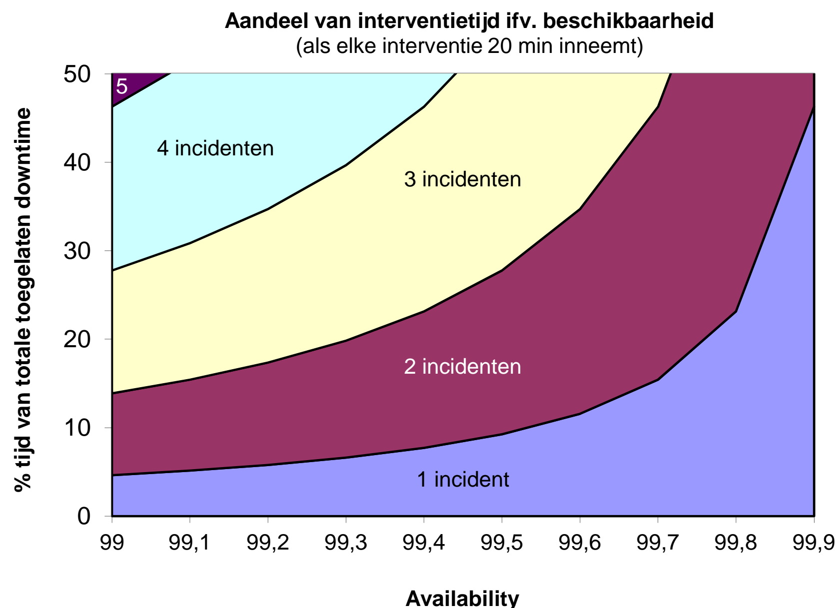
Complexe systemen zijn veel moeilijker highly available te maken omdat er meer relaties zijn tussen de verschillende componenten. Hoe simpeler de oplossing (ook al lijkt ze "stupid"), hoe beter. Niet alleen is de kans op falen groter bij complexe systemen, maar ze zijn bovendien ook moeilijker te begrijpen, te onderhouden en te verbeteren.

> *Het **ontkoppelen** van systemen (bijvoorbeeld aan de hand van wachtrijen) is buiten het **verhogen van de beschikbaarheid** ook een ideale manier om de **complexiteit te reduceren**!*

### 2.3. Van 99 ➔ 99.9 %: not quite the same

Belangrijk is inzien dat elke extra "negen" overeenkomt met tien maal minder downtime. **De overgang van 99 % naar 99.9 % is dan ook vooral erg ingrijpend op organisatorisch vlak**. Eén van de redenen is bijvoorbeeld dat er bij een traditionele organisatie (met het doel 99 % beschikbaarheid te bieden) meestal een vaste tijdskost geassocieerd is met elk incident.

> *De traditionele manier van werken (manuele analyse en resolutie van problemen) is ontoereikend om aan High Availability het hoofd te bieden.* ***Een systeem moet autonoom in staat zijn om zichzelf draaiende te houden.*** Menselijke arbeid dient enkel nog voor het fysieke onderhoud, analyse en preventie.




*Figuur 5: Het maximaal aantal "toegelaten" incidenten neemt sterk af als hogere beschikbaarheid vereist is, omdat manuele interventie veel tijd in beslag neemt*

Stel dat de detectie van een incident en de toewijzing van human resources, 20 minuten in beslag neemt. Bovenstaande grafiek plot het aandeel van deze tijd in de totale maximale toegelaten downtime per maand – de downtime is omgekeerd evenredig met de beschikbaarheid. Uitgaande van de veronderstelling dat *detectie* maximaal 50 % van de totale tijd in beslag mag nemen (er moet immers nog veel ander werk gebeuren zoals isolatie van het probleem en het zoeken van de oplossing, cf. §3.2), merken we dat **slechts één incident is toegelaten voor 99.9 %, terwijl dit nog 5 incidenten was bij 99 %.**

Bovenstaande redenering geldt dan nog enkel in het optimistische geval dat het om een klein incident gaat waarbij in 23 minuten het probleem geanalyseerd, opgelost en uitgerold kan worden!

> *Bovenstaande voorbeelden tonen duidelijk dat:*
>
> 1. *de **businesscontext** sterk meespeelt;*
> 2. *de **architectuur** cruciaal is (factor 40 winst in beschikbaarheid!);*
> 3. *de **beschikbaarheid** kan opgekrikt worden **van 90 % naar 99.75 %** indien een aangepaste architectuur gekozen wordt..*

# 3. Oorzaken van “Low” Availability

De totale beschikbaarheid wordt bepaald door de kans op falen van de componenten en de tijd die nodig is om een incident op te vangen. We zullen deze twee aspecten van Availability nu apart bekijken.

## 3.1. Kans op falen

Er bestaan verschillende manieren om systeemfalen in te delen. Er bestaat namelijk een brede waaier aan oorzaken waarom een systeem faalt. Zonder teveel in detail te treden, onderscheiden we op hoog niveau de volgende oorzaken [1, 2, 3]:

1. **Software failure**
   Software bugs kunnen software doen blokkeren, maar ook bv. memory leaks.
2. **Hardware failure**
   Hieronder verstaan we defecte hardware (kapotte disks, power supplies, fans, …) of hardware met bugs.
3. **Network failure**
   Naast het feit dat switches kunnen falen evenals kabels, kan ook een verkeerde configuratie van een firewall of switch het netwerkverkeer onderbreken.
4. **Environmental failure**
   Het uitvallen van de stroom, van de airconditioning, gebroken kabels, brand,…
5. **Timeouts / Expiration of certificates**
   Certificaten kunnen vervallen, of timeouts kunnen zich voordoen doordat een externe server overbelast is of gewacht wordt tot een “lock” vrijgegeven wordt.
6. **Menselijk falen**
   Mensen zijn vergeetachtig wat zich uit in slordige changes waarbij failovers niet volledig getest worden, afhankelijkheden over het hoofd gezien worden, etc…
7. **Denial-of-service attacks**
   Aanvallen hebben vaak tot doel het opzettelijk onbeschikbaar maken van bepaalde diensten.
8. **Configuration error**
   File permissions, geblokkeerd netwerkverkeer, broken libraries, … kunnen veel schade aanrichten. Het is belangrijk al deze verbanden goed in kaart te brengen.

## 3.2. Lifecycle of an outage

Als er zich een incident voordoet, moeten er verschillende stappen doorlopen worden om het systeem terug up & running te krijgen. Het zijn deze processen die de uiteindelijke downtime bepalen. Complexe systemen maken deze processen moeilijker en trager.

Bij een SLA van 99.9 % of 43 minuten toegelaten downtime per maand, moet bij één enkel incident binnen deze tijd volgende zeven stappen doorlopen worden [1]:

- **Detectie van het incident**
  In de eerste plaats moet een incident zo snel mogelijk gedetecteerd worden. De monitoring moet hiervoor nauwkeurig genoeg zijn (realistisch gebruik dekken) en de menselijke reactietijd gerelateerd aan de escalatie moet minimaal zijn.
- **Mensen toewijzen**
  Eens een incident bevestigd is, moet de nodige mankracht toegewezen worden. Dit hangt af van de flexibiliteit en beschikbaarheden van de technische teams.
- **Analyse & locatie van het probleem**
  Vervolgens moet het probleem geanalyseerd en gelocaliseerd worden (wat zeker niet voor de hand liggend is in een complexe ICT-omgeving. Zo kan een ventilator het tijdelijk begeven hebben en is het probleem *schijnbaar* verdwenen na heropstarten).
- **Isolatie van het probleem**
  Het is niet altijd mogelijk om een probleem op te lossen, omdat er bv. een databank "down" gebracht moet worden die op het moment van het incident noodzakelijk is voor een andere applicatie.
- **Allocatie van resources**
  Eventuele reserveonderdelen, handleidingen, software, backups, tapes… moeten beschikbaar worden gesteld.
- **Remediëring incl. documentering**
  Uiteindelijk wordt de effectieve herstelling uitgevoerd, configuratieparameters veranderd, backups teruggezet, software geïnstalleerd, het OS geüpdatet, etc. De impact van de change moet goed nagegaan worden en de change duidelijk gedocumenteerd in de *problem recovery guide*.
- **Verificatie**
  Als laatste stap moet de functionaliteit en integriteit van het gehele end-to-end systeem gecontroleerd worden. Zeker in complexe omgevingen is dit niet eenvoudig gezien de vele interagerende systemen.

Elk van deze stappen moet zo efficiënt én effectief mogelijk uitgevoerd worden en daarom best geautomatiseerd worden of tenminste ondersteund worden door software om de kans op fouten te minimaliseren.

# 4. Het CAP-Theorema

In de vorige paragraaf werd het belang van een redundante en ontkoppelde architectuur uitgelegd om hoge beschikbaarheid te bereiken. Het toevoegen van deze twee factoren zorgt ervoor dat we intrinsiek evolueren naar **gedistribueerde systemen**. Door de toegevoegde complexiteit krijgen we te maken met nieuwe fenomenen.

Beschouwen we volgende eigenschappen van gedistribueerde systemen:

- **Consistency (C)**

  De antwoorden afgeleverd door het systeem zijn steeds gelijk en up-to-date.

- **Availability (A)**

  Het systeem geeft steeds een antwoord.

- **Partition tolerance (P)**

  Het systeem is *on*gevoelig voor interne communication failures (het netwerk mag al eens een pakket laten vallen). Een partitie is een "deel van het netwerk" dat volledig afgesloten is van de rest van het netwerk. Een systeem dat niét partition tolerant (P) is (=*niet tegen communication failures kan*), komt overeen met de eis dat het netwerk volledig betrouwbaar is. Dit is in de praktijk nooit het geval!

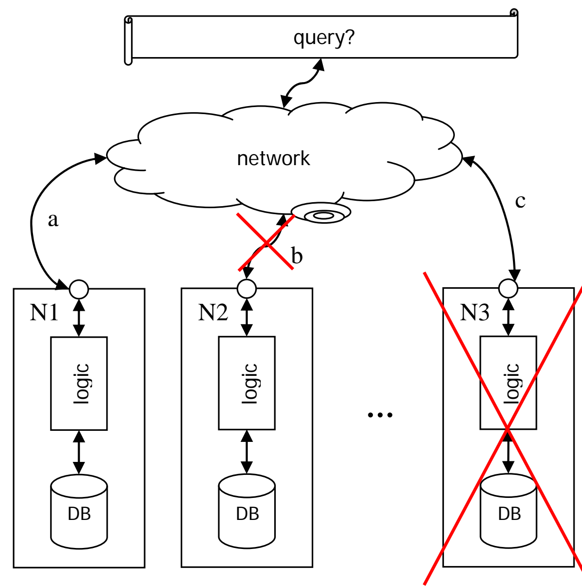
Ruwweg zegt het CAP-theorema dat er een fundamentele trade-off bestaat tussen de beschikbaarheid en consistency van gegevens. Dit wil zeggen dat als je de beschikbaarheid wil verhogen, je moet inboeten aan consistency en vice versa.

## 4.1. Het theorema

In de volgende paragrafen zullen we uitgaan van het volgend conceptueel systeem uitgebeeld in Figuur 6.

Een vraag ("query"), gesteld aan het systeem wordt willekeurig doorgespeeld aan één van de drie systemen, bereikbaar vanop het netwerk. Onderzoek aan de UC Berkeley door heeft uitgewezen dat er een fundamentele wet, het "CAP-theorema" [4] bestaat voor gedistribueerde systemen. Deze werd later bewezen door onderzoekers aan het MIT [5]:

> *Bij gedistribueerde systemen kunnen volgende drie eigenschappen*
> - ***Consistency***
> - ***Availability***
> - ***Partition Tolerance***
>
> ***niet alle drie* op hetzelfde moment *gegarandeerd worden.***



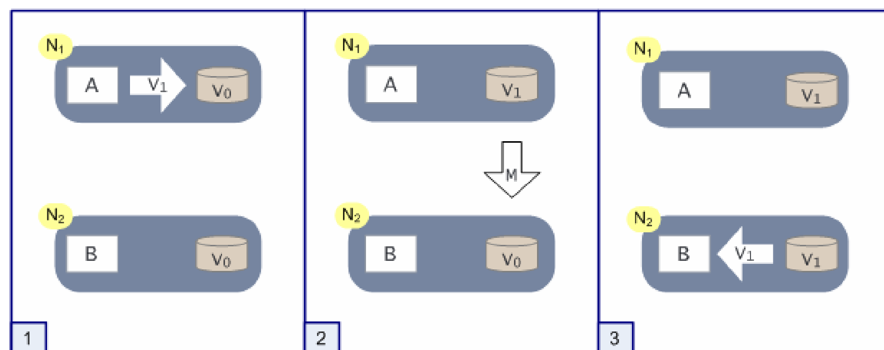

*Figuur 6: Voorbeeld van een gedistribueerd systeem. Een netwerkconnectie kan (tijdelijk) onderbroken worden (geval b), maar een node kan ook onbeschikbaar worden (zoals aangegeven door het kruis bij node N3)*

## 4.2. Grafische interpretatie

Stel dat node N1 een bepaald gegeven V0 verandert naar V1. Node N2 moet hier uiteraard van op de hoogte gesteld worden. Er kunnen zich nu verschillende situaties voordoen.

### 4.2.1. Geval 1: Alles verloopt naar wens

In het eerste geval zijn alle systemen up & running en het netwerk beschikbaar.




*Figuur 7: Een gedistribueerde transactie waarbij alles goed verloopt*

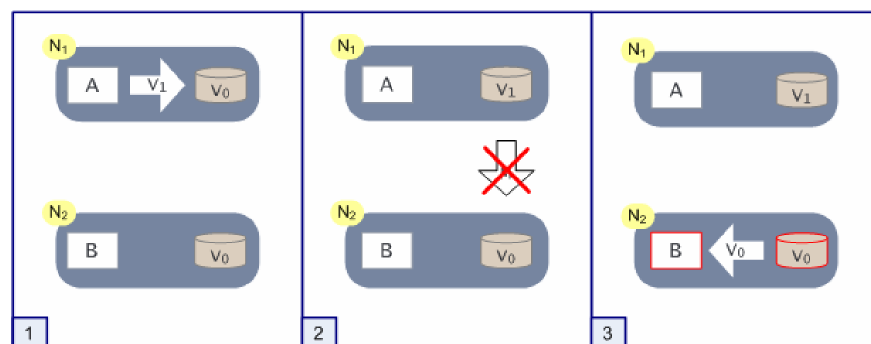
1. Proces A schrijft een nieuwe waarde V1 weg naar de databank in node N1
2. Een message M gaat over het netwerk naar N2
3. De nieuwe waarde V1 is beschikbaar in de databank in node N2
4. Proces B beschikt over de nieuwe waarde

In praktijk zullen netwerken echter altijd wel eens falen en moeten systemen bijgevolg partition-tolerant gemaakt worden. Deze situatie wordt in volgende paragraaf besproken.

### 4.2.2. Geval 2: Het netwerk is onbeschikbaar

In dit geval komt de boodschap M niet toe bij N2. De eerste stappen blijven hetzelfde:

1. Proces A in systeem N1 schrijft V1 weg naar V0
2. Node N1 stuurt een boodschap M naar node N2




*Figuur 8 : Een netwerkonderbreking zorgt voor problemen. In dit voorbeeld wordt beschikbaarheid verkozen boven consistency (B krijgt V0 ipv. V1)*

Op dit moment kan systeem N1 niet weten of het systeem N2 dan wel het netwerk down is (het systeem is partition-tolerant). Volgens het CAP-theorema zijn er nu twee keuzes:

**A. Kiezen voor consistency**

Omdat proces N1 geen ontvangstbevestiging gekregen heeft van N2, wordt het proces A afgebroken (atomicity) daar de consistency tussen de twee databanken in N1 en N2 niet gegarandeerd kan worden. **Het systeem is unavailable,** maar heeft succesvol gereageerd op de partitie**.**

**B. Kiezen voor Availability**

Hoewel systeem N1 geen ontvangstbevestiging gekregen heeft van N2, wordt de transactie toch afgehandeld (het systeem is dus available). Op dit moment bezitten N1 en N2 verschillende versies van de gegevens. **Het systeem is inconsistent.** Zodra de partitie verdwenen is, wordt het systeem weer consistent.

### 4.2.3. Geval 3: Partition Intolerant Systems

Systemen die gevoelig (intolerant) zijn aan partities, falen zodra er een partitie optreedt. Een situatie zoals figuur 8 kan dus niet opgevangen worden. In alle andere gevallen echter **is de databank consistent en available.** Namelijk, als er een deelsysteem faalt (niet het netwerk), blijft de databank consistent en

beschikbaar!

### 4.3. The good news: Eventual Consistency

In realiteit zal in het tweede geval node N1 bijhouden dat N2 nog op de hoogte gebracht moet worden (in een soort backlog), en zodra communicatie met N2 terug mogelijk is, wordt de boodschap doorgestuurd. Het systeem noemt men **eventually consistent**. Het *tijdelijk* toelaten van inconsistency in geval van falen kan op business niveau opgevangen worden, eerder dan op lager, programmatorisch niveau.

**Dit laat toe de Availability enorm te verhogen met minimale gevolgen voor de gebruiker** door hiermee rekening te houden tijdens het ontwerp van de software. De truc is dat de inconsistencies op *businessniveau* worden opgelost. Meer info vindt u in het document "CAP-theorema".

> **Gedistribueerde systemen kunnen nooit gegarandeerd tegelijk highly available én consistent zijn.** Merk op dat de zwakke, tijdelijke inconsistency enkel optreedt op momenten dat een strikter systeem volledig onbeschikbaar zou zijn!!

## 5. Verhogen van de beschikbaarheid

Het is cruciaal te beseffen dat High Availability geen infrastructuurproject is, maar reeds vanaf de requirements analyse expliciet in rekening gebracht moet worden.

Vooral **op architectuurniveau valt er veel te winnen**, maar de relevantie van het organisatorische aspect mag zeker niet over het hoofd gezien worden. Een "passieve failover"-strategie waarbij een reserve-infrastructuur enkel in actie treedt bij een incident, is in praktijk minder robuust (en minder schaalbaar) dan een horizontaal georiënteerd systeem waarbij steeds alle componenten gebruikt worden. In dat laatste geval wordt de impliciete "failover" immers voortdurend getest en maakt het vervangen van defecte componenten deel uit van het standaard onderhoud, eerder dan van incident management.

Conceptueel liggen er **vier principes** aan de basis van hoge beschikbaarheid [2,6]:

- **Transparantie**
  (Documenteer, Test Everything, Gebruik SLA's, Plan)
- **Redundantie**
  (Reuse, remove SPOF's, leer van vroegere fouten, maak geen veronderstellingen)
- **Simpliciteit**
  (Keep it Simple Stupid, "one thing at a time", gebruik mature software & hardware)

- **Diversiteit**
  (bouw HA in op alle niveaus, zowel logisch als organisatorisch: van requirements analyse tot development)

In de volgende paragrafen zullen de mogelijke manieren om te komen tot High Availability kort besproken worden van inceptiefase tot productie.

## 5.1. Requirements analyse

### 5.1.1. Levensduur en vluchtigheid van de gegevens

Reeds tijdens de requirements analyse kan er fundamenteel bijgedragen worden tot het bewerkstelligen van High Availability. Om later tijdens de architectuurfase de nodige trade-offs en beslissingen te kunnen nemen wat betreft Consistency vs. Availability, moet er tijdens de voorgaande fasen de nodige voorbereidingen getroffen worden. Zo moet er voor elk gegeven de levenscyclus bepaald worden:

- Wat **de beschikbaarheid** van elk gegeven moet zijn?
  (hoe vaak wordt het gegeven gelezen)
- Wat **de vluchtigheid** van dit gegeven is?
  (hoe vaak verandert de waarde)
- Wat **de levensduur** van een gegeven is?
  (wat is de tijdspanne tussen het aanmaken en de laatste wijziging van het gegeven?)

### 5.1.2. Benodigde beschikbaarheid per use case

Bovenstaande vragen kunnen objectief beantwoord worden of afgeleid door een zorgvuldige analyse van de huidige systemen & processen. Daarnaast moet in overleg met de klant nagegaan worden:

- **hoe belangrijk de consistency is van elk gegeven**, rekening houdend met de vluchtigheid van het gegeven (bv. verzekerbaarheid moet slechts per dag consistent zijn)
- **welke beschikbaarheid gewenst is voor elke use case**, (in plaats van het systeem in zijn geheel)

Een tabel die al deze gegevens verzamelt, zou er bijvoorbeeld als het onderstaande voorbeeld kunnen uitzien. Feitelijk verschillen deze parameters vaak ook nog per use case – voor batchverwerking gelden andere voorwaarden dan online raadpleging.

| Gegeven | Availability | Consistency | Volatility | Lifecycle |
|---|---|---|---|---|
| Verzekerbaarheid autoverzekering | < 10 min / dag | Op één dag nauwkeurig | Max. 1 maal / maand verandering | 1 maand |

## 5.2. Architectuur

De gegevens die verzameld zijn tijdens de requirements analyse worden in de architectuurfase gebruikt om te bepalen **welke systemen ontkoppeld** mogen en kunnen worden en **waar Consistency dan wel Availability de voorkeur krijgt**. Tijdens de architectuurfase worden de business requirements immers vertaald in een conceptuele structuur die zijn weerslag zal vinden in een implementatie.

### 5.2.1. Technieken

Het uittekenen en bepalen van de architectuur is een cruciale fase in elk project waarin in grote mate de beschikbaarheid, onderhoudbaarheid en schaalbaarheid wordt bepaald. Gelukkig zijn er principes bekend die de beschikbaarheid aanzienlijk kunnen verhogen:

- **BASE vs. ACID [7]**
  Traditionele relationele databanken stellen integriteit als absolute prioriteit. Ze steunen hierbij op *atomicity* (ofwel slagen alle delen van een transactie, ofwel geen enkele) en *consistency* (alle gegevens zijn steeds gelijk en up-to-date). Hoewel dit op het eerste gezicht logisch lijkt, blijken deze voorwaarden vaak veel te strikt. Fundamenteel komt dit door het feit dat **de woorden "altijd" en "nu" een totaal verschillende betekenis hebben in IT-context dan in een businesscontext**. "BASE"-systemen stellen beschikbaarheid en schaalbaarheid voor op consistency, in lijn met het CAP-theorema.

- **Ontkoppelen**
  Aan de hand van de vluchtigheid en beschikbaarheid van de gegevens moet bepaald worden welke deelsystemen data-onafhankelijk gemaakt (ontkoppeld) kunnen worden. Gegevens die met een hoge beschikbaarheid moeten kunnen geraadpleegd worden, moeten best ontkoppeld worden om de kosten beheersbaar te houden. Ontkoppeling komt in feite overeen met asynchrone communicatie, zodat systemen niet meer steunen op de beschikbaarheid van andere systemen:

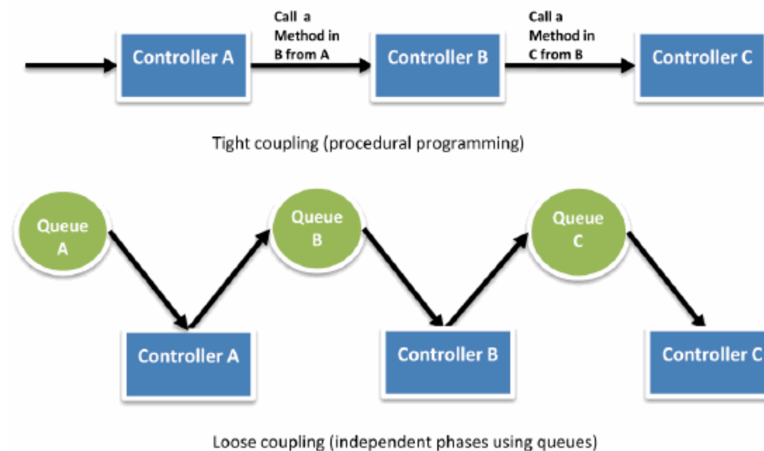



*Figure 9: Tight coupling vs. Loose coupling*

- **Keep it Simple Stupid**
  Simpele architecturen hebben meer kans op slagen en bezitten een hogere beschikbaarheid. Bovendien zijn ze beter begrijpbaar en transparanter en kosten ze doorgaans minder omdat ze gemakkelijker zijn in onderhoud. Ook worden incidenten typisch sneller opgelost.
- **Be a pessimist during design**
  Ga ervan uit dat alles zal mislopen. Door pessimistisch te zijn tijdens het uittekenen van de architectuur, zorg je ervoor dat er al nagedacht wordt over recovery strategieën op het moment dat nog veel mogelijk is voor een lage kost (in het algemeen geldt dat hoe vroeger de maatregel genomen wordt, hoe beter). Bedenk strategieën die een automatisch herstel toelaten. Ga ervan uit dat de hardware zal falen, er meer requests zullen zijn dan begroot, dat het netwerk zal falen, etc…
- **Diversity**
  Diversiteit tijdens de implementatie is een goede manier om statistisch gecorreleerde failures tegen te gaan, omdat er voor verschillende implementaties verschillende mechanismes aan de oorsprong liggen van het falen. Zo zal een hard disk een andere levensduur en andere gevoeligheden hebben dan een tape – beide oplossingen aanwenden zorgt dat de kans op gezamenlijk falen van beide oplossingen gevoelig verkleint. Gelijkaardige redeneringen kunnen ook tijdens development toegepast worden.
- **Elimineer *alle* single points of failure**
  Dit komt neer op redundancy. Hoewel het voor de hand liggend lijkt, is het een absolute noodzaak om het gehele end-to-end-systeem onder de loep te nemen. Wat gebeurt er als de SAN onbeschikbaar wordt? Of de toegang tot de ISP? Of een persoon? Het gsm-netwerk?

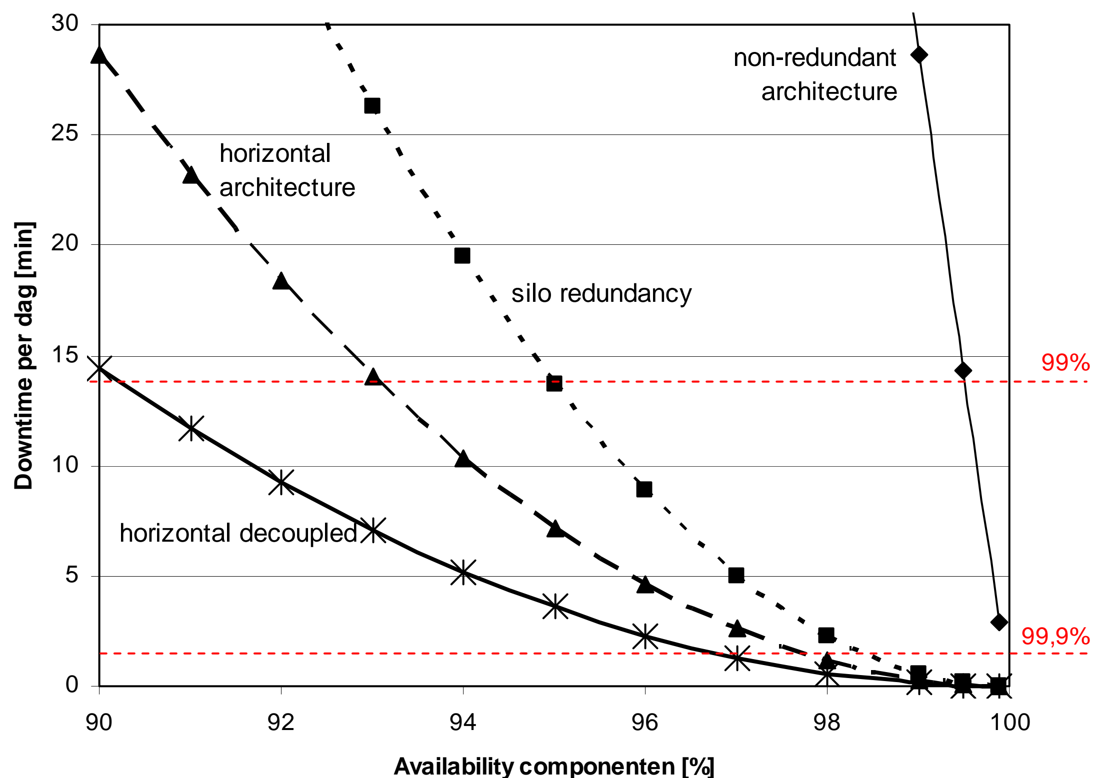
### 5.2.2. Het belang van ontkoppeling

Het voordeel van asynchrone en ontkoppelde systemen is dat men kosten bespaart doordat overbodige redundantie vermeden kan worden. Stel dat A het

front-end-systeem is en B het back-end en dat enkel het front-end highly available moet zijn. In dit geval kan gekozen worden voor een hybride architectuur waarbij A redundant wordt geïmplementeerd maar B niet. Tussen beide systemen wordt dan een queue gezet.

Volgende plot toont de verbetering van de beschikbaarheid in functie van de Availability van de basisblokken A en B, overeenstemmend met Figuur 3. Voor een systeembeschikbaarheid van 99 % (of ongeveer 15 minuten downtime per dag – cfr. rode stippellijn), is er, afhankelijk van de architectuur, een beschikbaarheid van de basisblokken van 90 % (horizontal queue) tot 99.5 % (basisarchitectuur) nodig (factor 20!!).

De systeembeschikbaarheid staat op de verticale as aangegeven met rode stippellijn. De horizontale as belichaamt de beschikbaarheid van de samenstellende componenten.




*Figuur 10: Vergelijking van de beschikbaarheid in functie van de basisbeschikbaarheid van elk blok voor 4 verschillende architecturen*

> De grafiek geeft duidelijk aan dat **de impact van architecturale wijzigingen vele malen groter is dan het verbeteren van de Availability van de componenten tijdens development!** Bovendien is de kost ook veel lager daar we ons vroeger in de applicatie-ontwikkeling bevinden.

## 5.3. Organisatie

### 5.3.1. Transparantie, governance, automatisering!

Ook de organisatorische context waarin een applicatie ontworpen en onderhouden wordt, bepaalt in grote mate de kwaliteit van de code en van de ICT-omgeving (het platform) waarop de toepassing draait. Bovendien hangt de tijd die nodig is om een incident te detecteren en op te vangen sterk af van de efficiënte werking van de organisatie.

**Een applicatie met hoge beschikbaarheid heeft nood aan een mature organisatie** waarin transparantie, automatisatie en governance een hoge prioriteit hebben.

We zullen de impact van elk van deze componenten op beschikbaarheid kort bespreken [3]:

- **Transparantie**
  Langetermijnbetrouwbaarheid is steeds gebaseerd op een transparante organisatie, begrijpelijke en up-to-date documentatie.
- **Automatisatie**
  Menselijke fouten kunnen in grote mate vermeden worden en herstelprocedures sneller uitgevoerd worden.
- **Governance**
  Om te kunnen automatiseren en transparantie te bereiken is een goede governance nodig, zoals Quality Management, Release management; om een goede root cause analyse te doen moet de CMDB up-to-date zijn (dit kan verbeterd worden door automatische discovery).

### 5.3.2. Voorkomen is beter dan genezen

Het voorkomen van incidenten is nog steeds beter dan ze op te vangen en op te lossen. Een tijdig ingrijpen is dus essentieel (bv. door het detecteren van een memory leak of een graduele stijging van het aantal requests die uiteindelijk de limieten van het systeem zullen overschrijden). Zo moeten de logs bijvoorbeeld aandachtig bestudeerd worden vóór er zich een incident voordoet om op eventuele problemen te kunnen anticiperen. Dit werk kan niet manueel gebeuren.

### 5.3.3. Genezen is beter dan verdoven

Als er zich toch een incident voordoet, kan een “problem recovery guide” de oplossing aanzienlijk versnellen. Zorg voor een beslissingsdiagram dat de

technici moeten volgen dat ook aangeeft wat er moet gebeuren als het probleem niet meteen gevonden kan worden.

**Te vaak echter worden de symptomen aangepakt met een "bypass"** (bv. rebooten van een server met een memory leak) **in plaats van het werkelijke probleem aan te pakken.**

## 5.4. Development

De intrinsieke kwaliteit van de applicaties bepaalt de "baseline Availability" die gehaald kan worden. Een paar simpele principes, consequent aangehouden tijdens de ontwikkeling, kunnen de beschikbaarheid van de basiscomponenten aanzienlijk verhogen:

- **Naadloze en "Stateful" recovery na een incident**
  Bouw een systeem dat ongevoelig is voor reboots en crashes, maar zichzelf automatisch herstelt en integreert met de rest van het systeem, inclusief de state.
- **Wees pessimistisch : ga er van uit dat alles zal falen → test!!!!**
  Veronderstel dat argumenten verkeerde waarden zullen hebben, een functie verkeerde waarden zal teruggeven, time-outs zullen voorvallen, een file niet toegankelijk is, dat er meer aanvragen zullen zijn dan begroot, een array te klein zal zijn, etc… Daarom is rigoureus testen essentieel.
- **Bouw resistentie op alle logische niveaus**
  De regel van "pessimisme" moet op de verschillende logische niveaus toegepast worden – binnen een procedure/methode (check boundary conditions), op niveaus van klasses, tussen de klasses, op architectuur, …
- **Vergeet nooit de "fallacies of distributed computing"**
  Veel fouten vloeien voort uit het feit dat men verkeerde veronderstellingen maakt, zoals "het netwerk is betrouwbaar", "de latency is nul", "de bandbreedte is oneindig", "het netwerk is veilig", "de topologie verandert niet", "er is slechts één administrator", etc…

## 5.5. Infrastructuur

Uiteraard kunnen ook infrastructuur-technologische praktijken aangewend worden om de systeembeschikbaarheid te verhogen. Zonder te diep in detail te gaan, denken we bijvoorbeeld aan:

- **Encapsulatie in Virtuele machines**
  Virtualisatie is een erg efficiënte manier om systemen van elkaar af te schermen en het beheer ervan te versimpelen, omdat er abstractie gemaakt wordt van de fysieke infrastructuur waarop het systeem draait. Instanties kunnen gemakkelijk gemigreerd en gedupliceerd worden.

- **Gedistribueerde systemen**
  Er bestaan speciale gedistribueerde file systems en databases waarbij individuele pannes van hardware geen downtime tot gevolg hebben.

- **Geautomatiseerde backup & recovery strategie**
  Stel geautomatiseerde én geverifieerde strategieën voor backups en recovery in.
- **Crash-resistente database schema's en filesystems**
  Verkies bij MySQL bv. InnoDB boven een MyISAM, omdat deze laatste corrupt is bij crash en dus meer hersteltijd nodig heeft.
- **Redundante network connecties en opslagsystemen (RAID)1**
- **Load balancing en Network Redirection**

## 6. Conclusies

Availability is het percentage dat een geleverde dienst beschikbaar is relatief ten opzichte van de totale tijd dat de dienst verwacht wordt online te zijn door de klant. Het overstappen van 99 % naar 99.9 % komt overeen met tien maal minder downtime, 43 minuten vs. ongeveer 7u per maand.

Vooral op organisatorisch vlak is dit een ingrijpende verandering! **De traditionele manier van werken (de manuele analyse en oplossing van problemen),** is in deze context dan ook **absoluut ontoereikend**. Een systeem moet autonoom in staat zijn om zichzelf draaiende te houden. Menselijke arbeid dient enkel nog voor het fysieke onderhoud, analyse en preventie.

Er zijn twee manieren om de beschikbaarheid te verhogen: de kans op falen verminderen of de service downtime gerelateerd aan een incident verminderen. Er blijkt bijna altijd een trade-off te bestaan op businessniveau tussen de functional requirements en de Availability requirements**. Daarom is het van groot belang om High Availability reeds vanaf het begin op business niveau op te nemen,** door de levensduur en vluchtigheid van de gegevens evenals de beschikbaarheid *per use case* in kaart te brengen.

Er bestaat een fundamentele trade-off tussen a) de consistency van gegevens en b) hun beschikbaarheid, verwoord in het zogenaamde **CAP-theorema:**

> *Bij gedistribueerde systemen kunnen volgende drie eigenschappen:*
> - ***C - Consistency***
> - ***A - Availability***
> - ***P - Partition Tolerance***
>
> ***niet alle drie* op hetzelfde moment *gegarandeerd worden.***

Deze analyse laat toe om tijdens de architectuurfase de Availability aanzienlijk te verhogen door systemen op gepaste wijze te ontkoppelen en een juiste trade-off tussen Consistency en Availability te kunnen maken. **Het is op *architectuurniveau* dat er het meeste te winnen valt en de nodige beslissingen hiertoe kunnen niet genomen worden zonder de business te betrekken!**

In het algemeen geldt de regel "Keep it Simple Stupid". Ook pessimisme – ervan uitgaan dat alles zal falen – vanaf de inceptiefase tot tijdens development zorgt ervoor dat er vanaf het begin rekening gehouden wordt met mogelijke failures.

Last but not least, vereist High Availability een mature organisatie met gestroomlijnde en geautomatiseerde processen, transparantie en goede governance. Zeker in de context van Availability is voorkomen beter dan genezen en genezen beter dan verdoven (bv. door een "bypass" toe te passen onder de vorm van een reboot van een server).

*Fundamentally, high availability is a business decision*[1]*.*

*Sectie Onderzoek van Smals brengt met regelmaat verschillende publicaties uit over een hele waaier aan topics in de huidige IT-markt. U kan deze publicaties opvragen via het extranet :*

*http://documentatie.smals.be*

*Of u kan rechtstreeks contact opnemen met het secretariaat van de afdeling 'Klanten & Diensten', op het nummer 02/787 58 24.*

## Referenties

[1] Floyd Piedad en Michael Hawkins, "High Availability: Design, techniques and processes", *Prentice Hal PTRl*, 2001

[2] Evan Marcus en Hal Stern, "Blueprints for High Availability", *Wiley Publishing*, 2003

[3] Yuval Lirov, "Mission-critical systems management", *Prentice Hall PTR*, 1997

[4] Eric Brewer, "Towards Robust Distributed Systems", *http://www.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf*

[5] Seth Gilbertson en Nancy Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", *Newsletter ACM SIGACT News*, Volume 33 Issue 2, June 2002

[6] http://en.wikipedia.org/wiki/Fault-tolerant_system

[7] Dan Pritchett, "BASE: An Acid Alternative", *ACM Queue*, 28 juli 2008

[1] "Blueprints for High Availability", Evan Marcus, Hal Stern