INF5190 - Résilience et performance

Jean-Philippe Caissy

20 novembre 2019

Résilience

Définition : capacité d'un système à récupérer d'une défaillance et rester opérationnel

- Une défaillance dans un système (application Web) va se produire éventuellement
- Plus un système est complexe, plus les risques de défaillances sont élevées
  - Une application avec 3-4 systèmes peut facilement être disponible 100% du temps vs. une application avec des centaines de composantes
- Un système est résilient s'il reste fonctionnel malgré une défaillance

Résilience

Lors d'une défaillance, un système devrait être en mesure d'opérer en mode *défaillance partiel*.

Exemples :

- Amazon : la recherche ne fonctionne plus
- Netflix : les vidéos HD ne chargent plus
- Google : impossible de se connecter

Résilience

La résilience d'une application Web se fait sur 4 niveaux différents :

- L'application elle-même
- Les données
- Le réseau
- Les gens et la culture organisationnelle

Résilience

Patrons

Il existe plusieurs patrons à utiliser pour rendre une application résiliente.

Redondance

- Architecturer une application pour pouvoir rouler de manière redondante (plus d'une instance)
- La redondance s'applique à tous les niveaux :
  - Application Web
  - Base de donnée
  - Réseau
  - Employés
- La redondance permet d'éliminer les défaillances causés par un point de défaillance unique (*SPOF, ou single point of failure*)

Résilience

Redondance

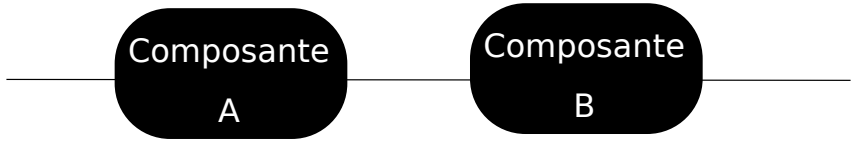

Figure 1: Composantes en série

La disponibilité d'un système en série est mesuré par la sommes de la disponibilité des deux systèmes.

Composante	Disponibilité	Temps indisponible
A	99%	3 jours, 15 heures
B	99.99%	52 minutes
A et B	98.99%	3 jours, 16 heures et 33 minutes

Résilience

Redondance

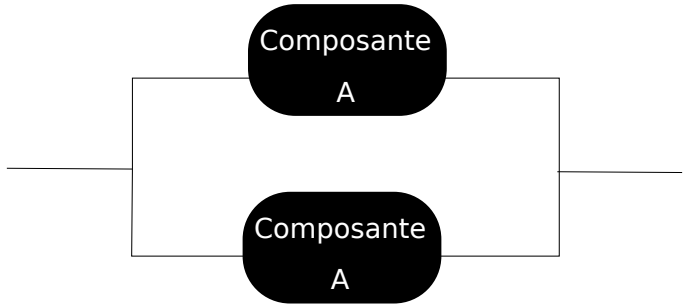

Figure 2: Composantes en parallèles

$$D = 1 - (1 - Ax)^n$$

Composante	Disponibilité	Temps indisponible
Un seul A	99%	3 jours, 15 heures
Deux A en parallèle	99.99%	52 minutes
Trois A en parallèle	99.9999%	31 secondes

Résilience

Mise à l'échelle automatique (*auto-scaling*)

- Automatiquement augmenter, puis diminuer les capacités d'un système
  - Mot clé : automatique, et non pas manuellement

Exemple : Diminuer le nombre de serveur applicatifs la nuit lorsque le trafique est très bas

Résilience

Défaillances en cascade

- Lorsqu'une défaillance survient, il est très rare que ça soit causé par un seul élément précis.
- Les défaillances en cascade sont très courantes dans les systèmes distribués (application Web)
- Effet papillon : ce qui semble être une petite coquille opérationnelle se termine par une défaillance complète du système Exemple de défaillance en cascade : surcharge applicative (trop de trafique)

Résilience

Défaillances en cascade

Algorithmes de recul exponentiels

- Dans un système distribuée, une manière simple de traiter les erreurs est de réessayer.
- Par exemple : un appel à un API distant qui ne répond pas. En cas d'échec, on refais la requête.
  - Peut facilement causé un problème de saturation sur le réseau (trop d'essais simultanés)

Pour empêcher une situation où un système surchargerait un autre en tentant de réessayer une requête, il est préférable d'utiliser un algorithme de recul exponentiel (*backoff algorithms*).

Résilience

Défaillances en cascade

Algorithmes de recul exponentiels

Objectif :

- Limiter le nombre de tentatives
- Plus le nombre de tentatives augmente, plus on attend entre chaque essais

Exemple :

1. Après une tentative infructueuse, attendre 2 seconde
2. À la 2e tentative, attendre 4 secondes
3. À la 3e tentative attendre 8 secondes
4. Retourner un erreur si la 4e tentative échoue

Résilience

Défaillances en cascade

Algorithmes de recul exponentiels

```
retries = 0
while:
    sleep( (2 ** retries) * 100 milliseconds )
    success = do_request()
    if status == True:
        break
    else:
        retries += 1

    if retries >= 10:
        raise Exception("Trop d'essais")
```

Résilience

Défaillances en cascade

Algorithmes de recul exponentiels

```
sleep((2 ** retries) * 100 milliseconds)
```

Il est important d'ajouter certain niveau d'aléatoire dans l'attente.

```
sleep((2 ** retries) * 100 milliseconds * rand())
# ----------------------------------------^
```

Résilience

Défaillances en cascade

Délais maximum

Lorsqu'une application communique avec un autre système, il est important d'avoir des délais maximum (*timeout*).
Par exemple:

- Un appel à la BD ne devrait pas prendre plus de 1 seconde
- Communiquer avec une API REST devrait se faire en moins de 4 secondes

Chaque situation est différente. Il n'existe pas de délais maximum magique.

Résilience

Défaillances en cascade

Disjoncteur

L'utilisation d'un disjoncteur (*circuit breaker*) permet de faire échouer des appels distants rapidement lors de défaillance.
Exemple :

- Un API distant ne répond plus depuis 30 secondes.
- Chaque appel d'API *timeout* après 10 secondes

Avec l'utilisation d'un disjoncteur, on peut faire échouer les appels subséquents sans attendre 10 secondes

Résilience

Graceful failing

Une fois qu'une défaillance est détecté, la dernière étape est d'adapter l'application pour avoir une défaillance partielle, ou *graceful failure* en anglais.
Lorsqu'une application est en défaillance partielle, seulement une partie des fonctionnalités est indisponible, mais le reste de l'application fonctionne.
Exemples:

- L'API distant qui récupère l'usager connecté est non disponible
  - Temporairement affiché un profil en mode invité
- La recherche d'un site ne fonctionne pas
  - Retirer la fonctionnalité de recherche
- Le système de paiement d'une boutique en ligne ne fonctionne pas
  - Désactiver l'achat, mais permettre de naviguer sur la boutique

Résilience

Graceful failing

```
class User(object):
    def get_connected_user(self):
        try:
            return User.get(id=request.user_id)
        except ConnectionError:
            return nil
```