Cloud

TP à remettre sous GitHub

Toutes applications et tous serveurs reposent sur le cloud 🡪 Cloud omniprésent

Toutes les données collectées sont traitées dans le cloud.

Dans les apps :

* Backend :
* Front end :
  + HTML
  + API REST / Logique d’application 🡪 Permet d’interagir avec l’infrastructure (type Cloud)
  + OS

Comment fait-on une application traditionnelle ? Architecture traditionnelle ?

Client 🡪 Serveur Web (type Apache) 🡪 Serveur d’application (Tomcat) 🡪 BDD (mySQL)

Dans le serveur d’application, on a toute la logique d’application. Par exemple :

* Facturation
* Gestion de compte
* Inventaire
* UI

Avantages :

* Simple à développer
* Simple à tester
* Simple à déployer

Inconvénients :

* Ne passe pas à l’échelle 🡪 Si le serveur tombe, plus rien ne marche
* Type monolithique 🡪 Toutes les fonctions ont accès à tout donc en cas de bugs, c’est difficile à corriger mais c’est également difficile à maintenir
* Accès aux données généralisé 🡪 Pas d’accès fin

Dans une application standard, dans une BDD, il y a environ :

* 8 tables
* 8 transactions
* 30 read-only
* 10 write-only

Dans une application standard, dans le serveur d’application, il y a environ :

* 15 fonctionnalités différentes

Il y a également 2 serveurs Web.

Cela va dépendre de la charge (nombre de clients).

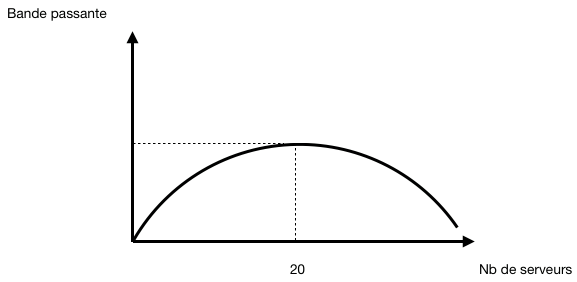
Comment passe-t-on à l’échelle ?

🡪 Quel est le point de fragilité ?

Soit on ajoute :

* Plus de serveurs Web
* Plus de serveurs d’Application
* Plus de BDD

Au total, si on a plus de 20 instances (BDD, SA et SW), les performances chutent drastiquement.



Le point de fragilité

Serveur Web

Les serveur Web est StateLess 🡪 Sans état 🡪 Toutes les requêtes sont indépendantes

Ces requêtes peuvent donc être traitées par des serveurs différents. Il n’y a pas de problèmes de passage à l’échelle car on est StateLess

Serveur d’application

StateLess également 🡪 Chaque serveur s’exécute indépendamment les uns des autres

Donc pas de problème de passage à l’échelle

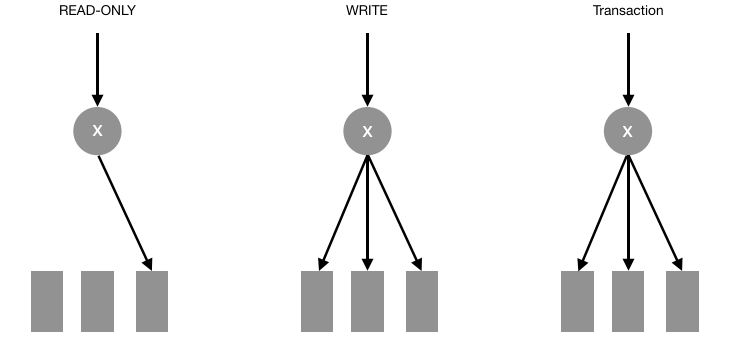
BDD

StateFull 🡪 Avec état.

Cela pose problème pour le passage à l’échelle. Donc chacune dispose d’un réplica.

READ ONLY 🡪 peut être réalisé sur chaque serveur indépendamment

WRITE ONLY 🡪 doit être réalisé par tous les serveurs



Transactions (ACID) : Plusieurs opérations d’écritures. Il faut garantir que tous les serveurs exécutent les mêmes opérations pour la même transaction. Il existe des algorithmes qui garantissent qu’une opération s’exécute sur tous les serveurs.

Si j’ai n base de données, alors la charge sera :

* Read / n
* Write
* Transaction

Charge = (Read/n) + Write + Transaction

Donc si on augmente n, cela n’a aucun impact sur les Write et les Transaction.

Rendre la maintenance et le déploiement/développement plus facile

Apache 🡪 Front end

Ce qu’on voudrait c’est que toutes les fonctionnalités du Front End soient isolées.

Exemple :

* Customer Service : 2 serveurs
* Items Service : 3 serveurs
* X Service : 1 serveur
* Recommandation Service : 5 serveurs

On essaye d’isoler toutes les fonctionnalités. Si c’est bien isolé, on va pouvoir les faire passer individuellement à l’échelle plus facilement

Identification d’une fonctionnalité 🡪 Isolation 🡪 Passage à l’échelle

Les services peuvent quand même interagir entre eux.

Ex : Pour lancer la page d’accueil d’Amazon va lancer une centaine de services.

**Une Fonctionnalité = un Microservice**

L’avantage des Microservices facilitent la maintenance. On peut changer un Microservice sans impacter les autres Microservices.

Qu’est-ce qu’un Microservice ?

Réalise seulement une tache mais extrêmement bien. Il est focalisé que sur sa tâche.

Monolithique :

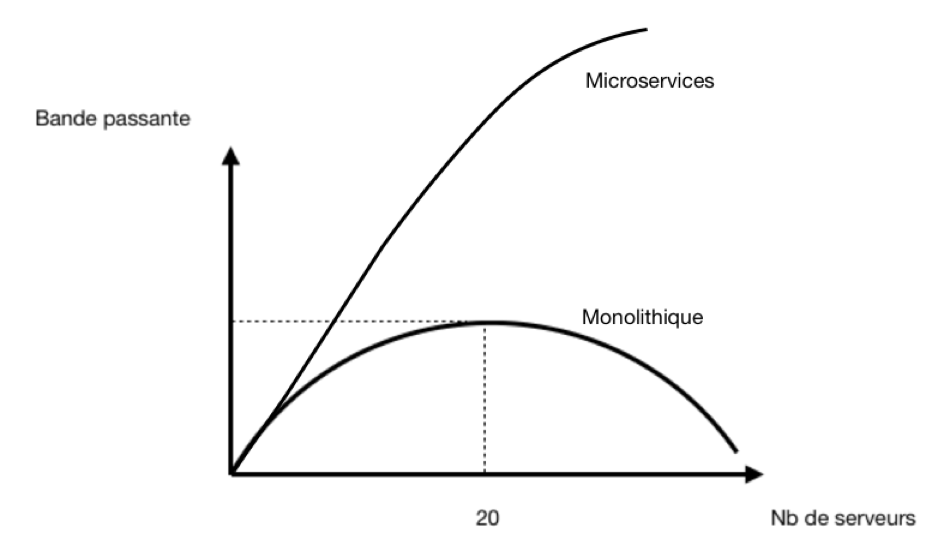
* Toutes les fonctionnalités dans le même process
* MAJ impacte la globalité
* Centralisation des données
* API globale

Microservice :

* Chaque fonctionnalité est découplée.
* Chaque MAJ impacte uniquement le Microservice.
* Décentralisation des données
* API dédiée à chaque service

Conclusion :

* Élastique : Possible de faire un scaling-up ou scaling-down d’un MS indépendamment des autres MS au sein de la même application
* Résilient : Si un MS tombe, les autres MS ne sont pas impactés.
* Composable : Un MS doit fournir une interface bien définie, uniforme qui permet de composer avec d’autres MS.
* Minimal : Une seule fonction
* Complet : Un MS doit être fonctionnellement suffisant 🡪 indépendant.

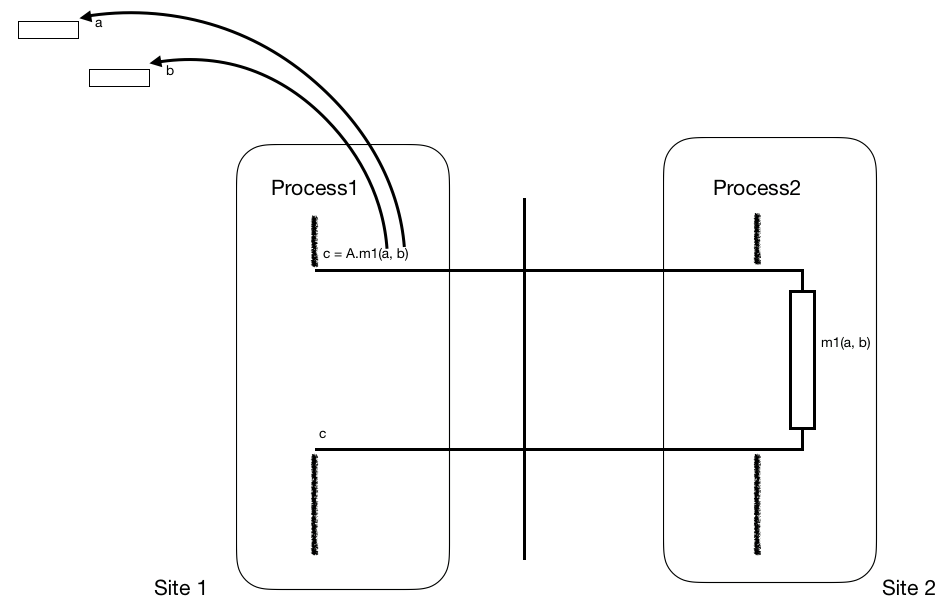


Interaction / Fonction d’un MS

Comment fait-on une interaction ?

On fait du RPC (Remote Procedure Call) 🡪 Synchrone

|  |  |
| --- | --- |
| Processus 1 | Processus 2 |
| |  |  |  c = A.m1(a,b) ----------------------------------------  Attente Active  c 🡨 ---------------------------------------------------  |  |  |  Site 1  Temps de latence plus élevé  On peut aussi avoir des pertes  Si on ne reçoit jamais c alors le processus 1 peut rester bloquer en attente active car le processus est synchrone  a et b sont des variables, des références donc des pointeurs vers une zone mémoire  Donc quand on exécute sur le processus 2, les références a et b sont nuls puisqu’ils sont stockés dans la mémoire où est exécuté P1  🡪 on fait des passage par valeur uniquement | |  |  |  -------------------------------------------------------🡪  m1(a,b)  ----------------------------------------------------------  |  |  |  Site 2 |



Format des messages entre process : JSON. Pour faire cela, on effectue de la sérialisation. C’est la transformation des données pour les envoyer dans un format compatible.

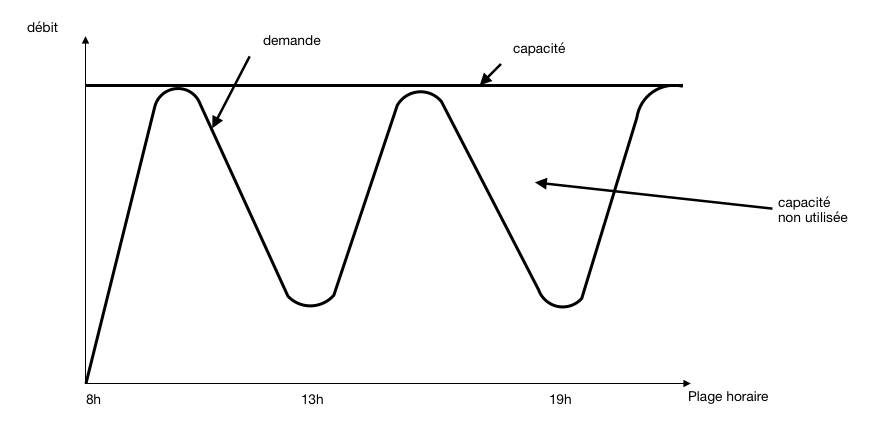
Docker

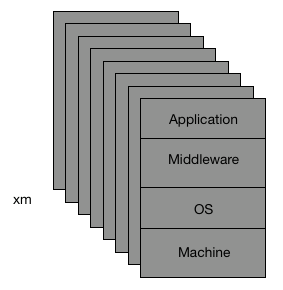
Conteneur : L’objectif c’est de répondre aux propriétés des MS. Le but est d’isoler un environnement du reste du système. On a un conteneur par MS.

Un conteneur correspond à un seul processus.

Cloud

Qu’est-ce que le cloud ?



On loue un serveur dédié. Le problème c’est que la demande n’est pas identique partout. Parfois la demande est plus faible que d’autres donc la capacité des serveurs n’est pas exploitée au maximum. Il y a donc des pertes sèche.

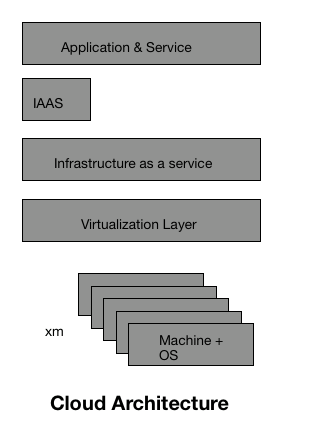
Si la demande augmente, la QoE et la QoS vont chuter. Pour lutter, il faut prévoir un dimensionnement. Cependant il faut faire attention à ne pas avoir une capacité non utilisée.

Cela engendre un cout supérieur par rapport à l’usage 🡪 sous-utilisation des ressources.

Le principal fournisseur de ressources est Amazon.

Dans l’architecture traditionnelle, la granularité n’est pas à grain fin.

Cloud Architecture

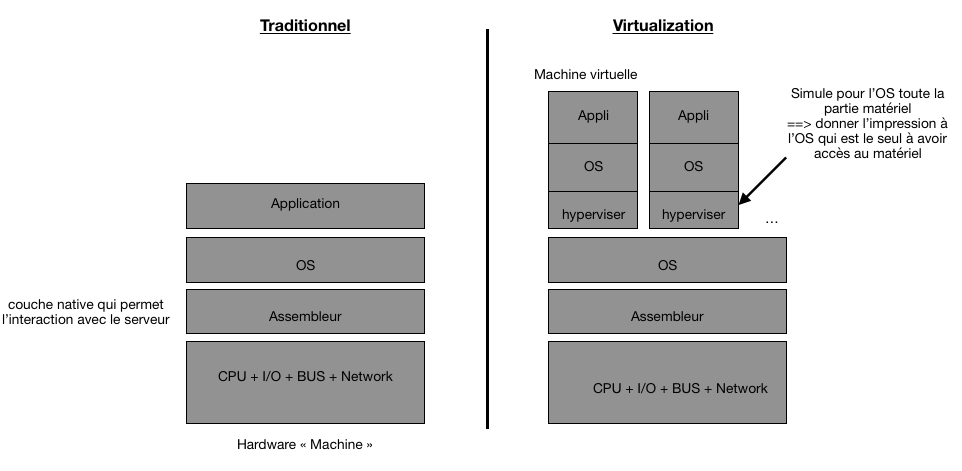
Objectif est de payer uniquement ce qui est consommé.

Le hardware est considéré comme un service 🡪 Le provider peut donc délivrer une certaine quantité de mémoire, de CPU ou de Bande Passante.

L’utilisateur n’a plus accès aux ressources matérielles et nese préoccupe plus de la localisation des services.

Le cloud abstrait complètement la complexité de l’infrastructure matérielle 🡪 utilisation uniquement d’une API pour réserver / consommer les données.

Les **services** sont disponibles à la demande, n’importe quand et n’importe où. **On ne paye que pour ce qu’on utilise.**



L’hyperviseur émule la partie machine. Il donne la main suivant un scheduler à chaque hyperviseur. Le but est de donner l’impression à l’OS qu’il est le seul sur le matériel.

Cela permet une isolation et permet d’allouer de la mémoire. Chaque VM possède un sous ensemble de ressources.

Avantages :

* **Personnalisation** : Utilisateur peut réserver une VM qui répond spécifiquement à ses besoins
* **Consolidation** : Optimisation des ressources de l’hôte suivant le besoin de l’utilisateur.
* **Gestion** : Gestion améliorer et au niveau des ressources.

Les solutions

VMWare, Microsoft, XEN, KVM, VirtualBox

Types de virtualisations :

* **Paravirtualisation** : L’OS de l’hôte est modifié pour permettre la virtualisation. Ex : XEN est le noyau Linux modifié pour utiliser la virtualisation
* **Virtualisation complète :** Rajouter directement dans le CPU des instructions matérielles qui permettent de gérer la virtualisation.

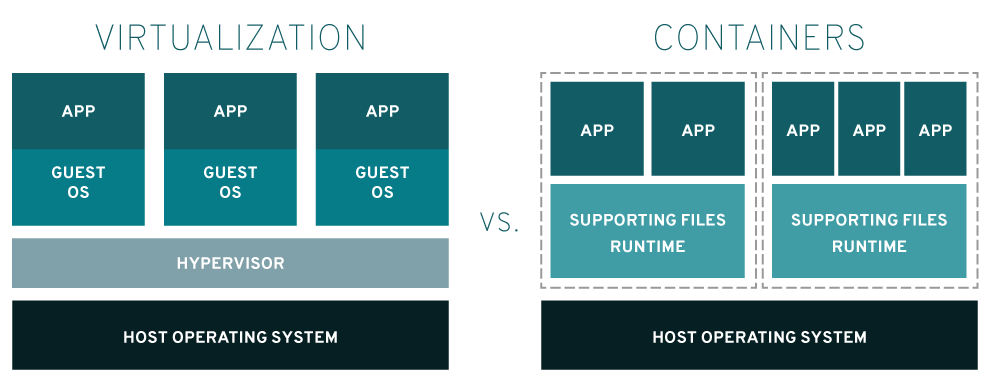
Les contraintes

Si on veut 200 VMS et qu’on veut un cœur par machine, cela signifie qu’il faut 200 cœurs. Il n’existe pas de machine qui possède 200 cœurs.

Il y a aussi un problème de démarrage. Le démarrage est long car c’est un vrai OS qui démarre à chaque fois car l’OS ne sait pas qu’il est lancé dans une VM 🡪 consomme des ressources

Si sur 50 VMs, il y en a 30 qui utilisent Linux, il faut trouver un moyen d’optimiser la solution.

Pour optimiser, on utilise les **conteneurs.**



On regroupe un ou plusieurs processus dans un environnement isolé 🡪 isolation les uns des autres

Un process appartient à un utilisateur. On est donc capable d’affecter des contraintes de capacités à chaque process ou groupe de process.

**Rôle d’une OS :**

* Abstraire le Hard
* Ordonnancer les différents processus qui consomment les ressources. Il affecte à chaque process un quantum de temps.

**Avantages des conteneurs**

* Pas besoin d’OS dédié et/ou modifications de l’OS sous-jacent
* Prise en charge nativement quelques soit l’OS (Linux, Windows, MacOS)
* On évite d’avoir à instancier les OS à peu près similaire
* Temps de démarrage plus court
* Le nombre de conteneurs peut être très élevé. On peut instancier un nombre conséquent de conteneurs sur une machine.

**Inconvénients des conteneurs**

* Beaucoup moins personnalisable qu’une VM 🡪 pas le choix de l’OS 🡪 Y a-t-il vraiment besoin de choisir l’OS ?

Public Cloud vs Private Cloud

Public Cloud

Ce sont des providers 🡪 Amazon, Microsoft, Google

Il faut payer pour l’utiliser

Private Cloud

Propre infrastructure 🡪 coût pour financer

Pour les entreprises de plus de 1000 VMs :

**2015 – 2016**

13% de Public Cloud

22% de Private Cloud

**2016 – 2017**

17% de Public Cloud

31% de Private Cloud