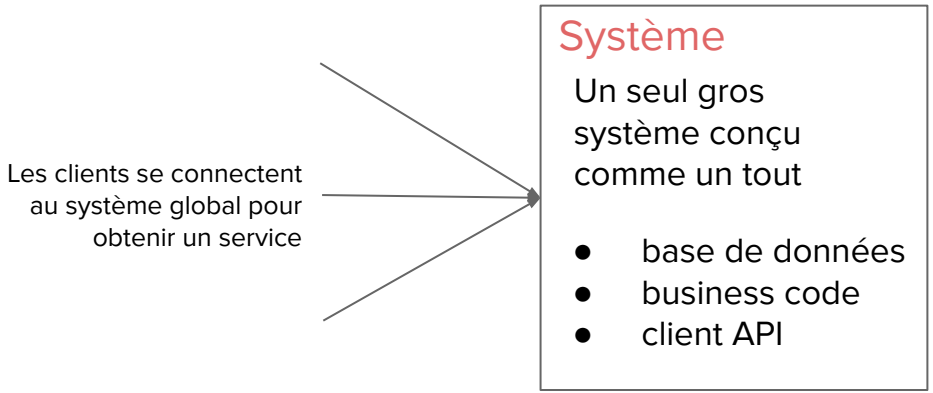
Autres connaissances architecturales

Repartons du début...

Qu’est-ce qu’une architecture monolithique ?
Quels avantages / quels inconvénients ?

Repartons du début...

Qu'est-ce qu'une architecture monolithique ?
Quels avantages / quels inconvénients ?



- c'est rapide dans une certaine mesure car pas de communications réseau !
- besoin de comprendre le code globalement pour le modifier ou y ajouter des choses (crosscutting)
- n'importe quel changement dans le code demande un redéploiement
- les tests sont compliqués à mettre en oeuvre
- peu tolérant aux pannes
- peu scalable
- peu flexible / évolutif

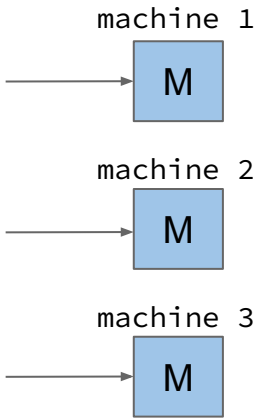
Repartons du début...

Est-ce que monolithique veut dire 1 seule machine / serveur ?

Repartons du début...

Est-ce que monolithique veut dire 1 seule machine / serveur ?
NON

Les clients se connectent à différents systèmes globaux pour obtenir un service



Mais pour autant, même si un système monolithique peut être **déployé sur N machines**, il n'est **pas décomposé en plus petits composants répartis qui collaborent** puisqu'il fonctionne comme un tout de façon centralisée !

On peut ensuite **répartir la charge** entre ces **instances du système monolithique**

Exemple d'architecture monolithique qui marche bien : Stack Overflow

Repartons du début...

Pourquoi faisons nous des architectures distribuées ?

Quels en sont les avantages / inconvénients ?

Repartons du début...

Pourquoi faisons nous des architectures distribuées ?

Quels en sont les avantages / inconvénients ?

Une architecture distribuée consiste à **dissocier** des **préoccupations,** des **fonctionnalités** ou des **ressources** d'un système en composants qui vont **communiquer et collaborer** les uns avec les autres pour accomplir le but global du système.

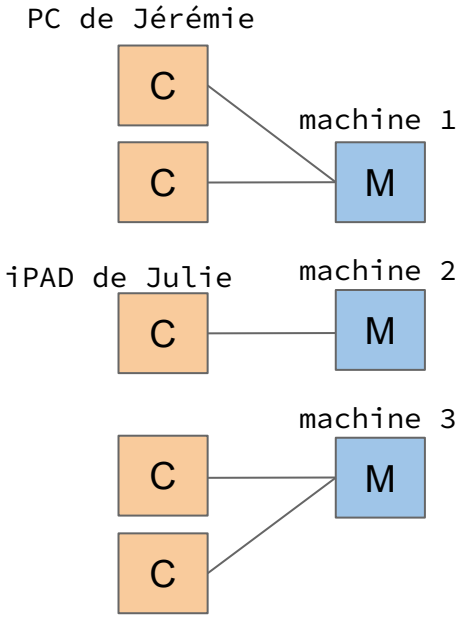
- Grands nombres de développeurs
  - séparation des préoccupations, flexibilité, maintenance
- Réutilisation de code facilitée et développement facilité
- Flexibilité / évolutivité améliorée
- Distribution et passage à l'échelle facilités par sous fonctionnalités
- Plus facilement tolérant aux pannes
- Plus difficile à concevoir (découplage, api, communications, sécurité, multi-langage etc.)

Distribué vs modulaire ?

- La **séparation des préoccupations** n'est pas quelque chose de spécifique au distribué
  - programmation modulaire
  - programmation objets
  - programmation par agents
  - etc.
- Le déploiement des "modules" ou "composants" sur des **machines distantes** et la **communication par le réseau** entre des entités est spécifique aux systèmes distribués

Monolithique vs client/serveur ?

D'une architecture **monolithique** à une architecture **client/serveur** la frontière est fine !



Si on offre une **application côté client pour interagir avec une application monolithique sur un serveur**, on a séparé les préoccupations qui sont distribuées entre les clients et le serveur !

Le code serveur reste monolithique, mais comme on considère une partie de l'application côté client et une partie côté serveur, et que les deux **communiquent**, on peut parler d'**architecture distribuée** !

Plus généralement, le plus petit élément considéré dans un système distribué est toujours monolithique à son niveau…

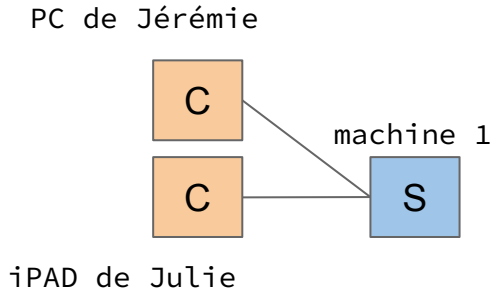
Quelques grands patrons architecturaux

Architecture client / serveur

Déjà abordée (voir slide d'avant)
C'est un peu le "niveau zéro" de l'architecture distribuée.

Très intuitive à comprendre : dissocier les éléments qui **consomment des ressources ou services (clients)** des éléments qui **offrent ces ressources ou services (serveurs)**. Les clients et les serveurs peuvent être distribués dans le réseau.

Donc pas besoin d'aller physiquement utiliser la machine 1 pour utiliser le service, on passe par le réseau !

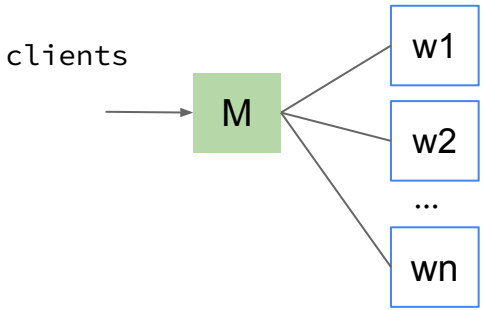


Le code client et le code serveur peuvent être écrits par des **développeurs et même des entreprises différentes** à condition que l'API du serveur réponde au besoin de l'application côté client.

Architecture master/workers

Dissocie les préoccupations dans le code côté serveur en maître / travailleur :

- **maître** : gère la réception des requêtes client, la division du travail et l'envoie de messages aux travailleurs
- **travailleur** : reçoit du travail du master, effectue le travail (et envoie une notification de travail terminé au serveur)



- s'applique plutôt aux cas ou le **serveur doit effectuer des calculs complexes** et longs.
- Permet d'**améliorer le temps de réponse** en divisant le travail.
- **Attention ! Diviser le travail côté Master peut être difficile !** (parallélisme, répartition, concurrence etc.)

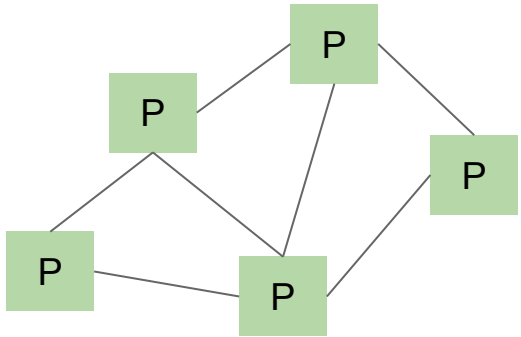
Exemple d'architecture master/worker : Apache Spark (BigData)

Architecture pair-à-pair (p2p)

Cherche également à **découper le travail**, les tâches, ou le workload entre les différents pairs, mais **tous les pairs ont les même privilèges** et il n'y a pas de maître pour la coordination centrale.

On parle aussi de **réseau pair-à-pair.** Les pairs sont à la fois créateurs et consommateurs de ressources (client et serveur).

Cette architecture est utilisée de très longue date mais a été popularisée et conceptualisée autour de 1999 avec le succès de Napster (partage de fichiers musicaux décentralisé).



Exemple d'architecture p2p : BitTorrent

- s'applique plutôt aux cas de partage de ressources entre des entités (données, CPU, disque etc.)
- très tolérant aux pannes car pas de connaissance centralisée, mais besoin d'algorithmes complexes à mettre en oeuvre pour que le système se comporte bien (découverte des pairs, établir le réseau entre les pairs, élire un leader etc.)

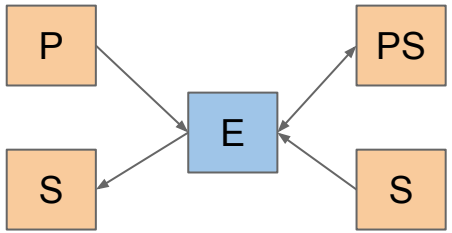
Architecture à évènements

Ici au lieu de considérer des API que l'on vient interroger on considères que des **événements sont émis lorsque des changements s'opèrent dans le système** (par exemple sur les données).

La gestion de ces événements devient alors le coeur du problème. Il peuvent être stockés de façon c**entralisée ou décentralisée, avec réplication** ou non**, avec maintien d'un ordre** ou non etc.

Globalement ce type d'architecture pose le problème de **confiance** dans la gestion des événements.

Il est reconnu plus difficile de comprendre avec finesse le comportement d'une application distribuée à événements car le **flux d'actions est plus difficile à percevoir**. Le code est plus difficile à maintenir pour cette raison aussi.

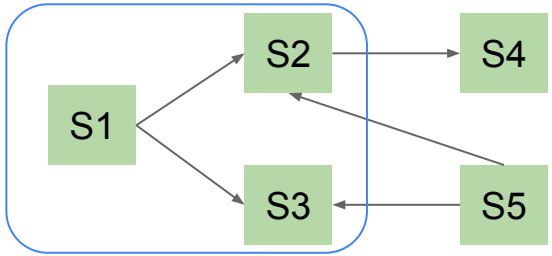


Exemple de technologies : Scala Akka, Publish/subscribe systems
Exemple d'architecture à événements : les applications IoT

Architecture orientée services (SOA)

Années 90 - pas vraiment de standard / specification

- Pousser le concept client / serveur plus loin en considérant des **services producteurs** de données et des **services consommateurs** (ou les deux à la fois).
- Découplage des fonctionnalités d'un système
- Ajoute aussi la notion de service broker (courtier), registry pour la découverte de services
- 4 propriétés pour un service en SOA :
  - il représente une **fonctionnalité répétable** avec des entrées et des sorties spécifiées
  - c'est une **boîte noire** pour les autres services qui l'utilise qui ne connaissent que son interface et non pas son fonctionnement interne
  - il peut lui même être composé de plusieurs sous-services
  - il spécifie son interface (**API**) et comment l'utiliser (**contrat**)



Architecture à microservices

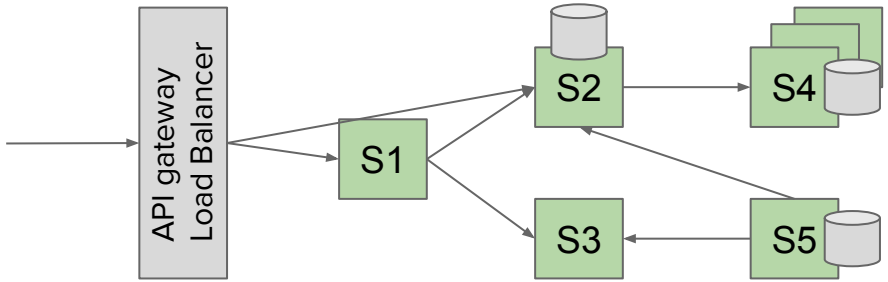
Début vers 2014 - pas vraiment de standard / spécification non plus

- Concepts très proches de l'architecture SOA !
  - Une **évolution avec les outils actuels des concepts de SOA**.
- L' architecture microservices est dite **"Cloud-native"** car elle se prête très bien à un déploiement de l'application sur le Cloud **[UE Cloud en A3]**
- Attention ! **"micro" ne veut pas forcément dire "petit"** mais avoir une granularité adaptée pour répondre à une préoccupation, et difficilement découpable elle même.
- **L'architecture microservices n'est pas incompatible avec les autres** ! (master/worker possible sous la forme de services aussi, publish/subscribe peut être utilisé comme des API REST etc.)

Architecture à microservices

- Le nombre de services fait que les clients ne se connectent en général pas directement aux services mais une **API gateway**
  - *routage des requêtes, caching, composition des requêtes, transformation des protocoles, load balancing, API spécifique pour un type de client etc.*
- Il est en général préférable d'avoir une **petite base de données pour chaque micro-services** pour réellement découpler les services
  - dans les TP nous avions un fichier json par microservice
  - attention à la duplication de données et au CAP theorem !

Très bonne suite d'articles : https://www.nginx.com/blog/introduction-to-microservices/

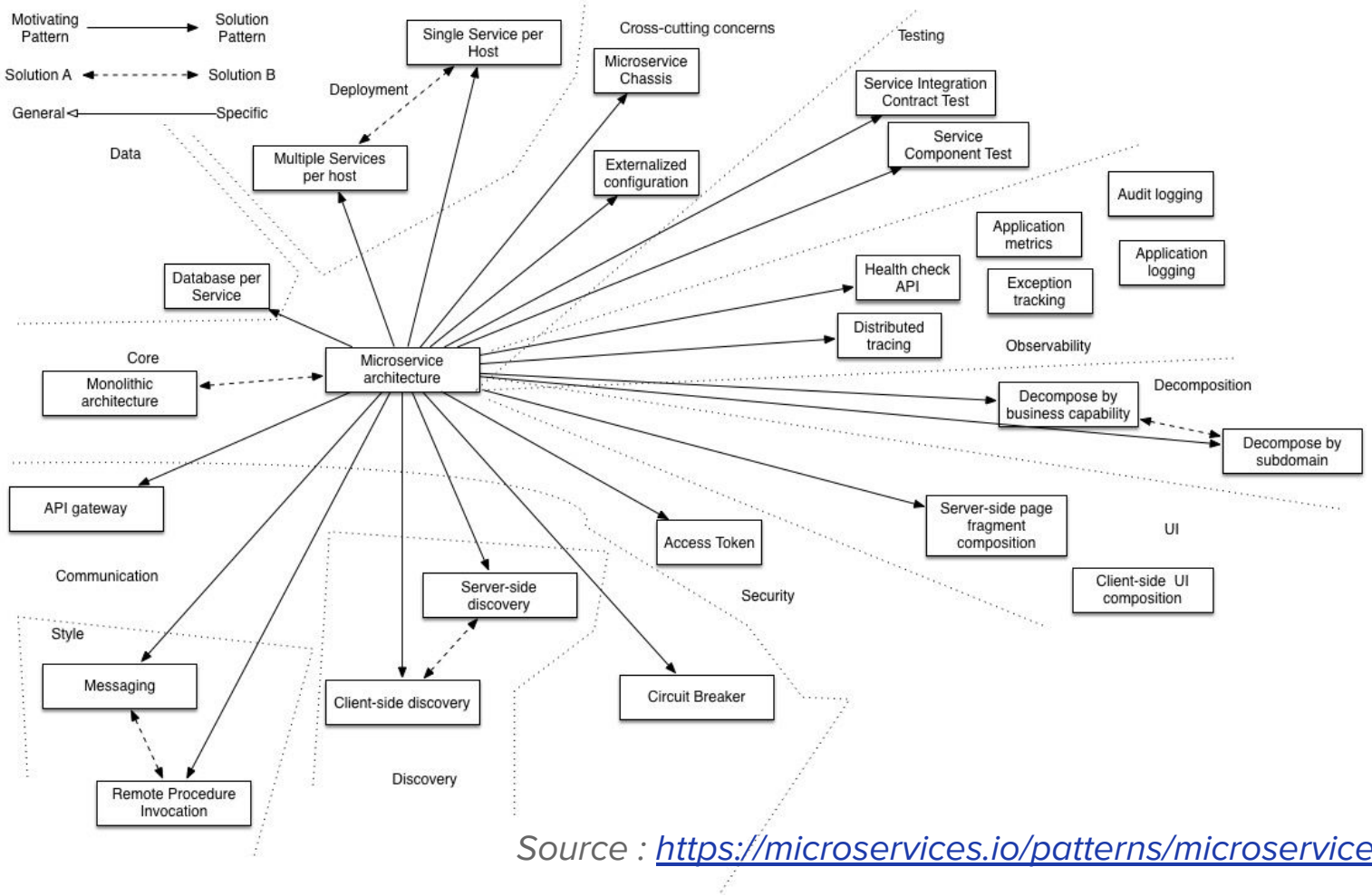


Les concepts associés aux architectures de services

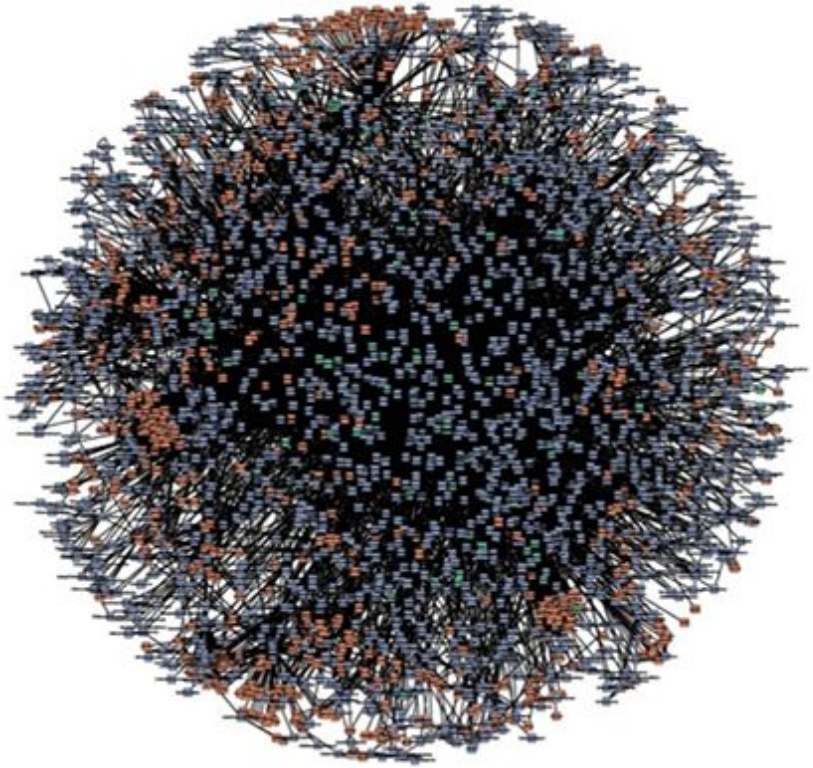
- API (REST, gRPC, GraphQL)
- Architectures pull vs push (API vs MQTT dans le projet)
- Load balancer et API gateway
- Databases (noSQL, relationnelles, sharding etc.), CAP theorem
- Caching systems
- Service discovery / registry
- Service mesh
- Circuit breaker
- etc.

A la limite du DevOps et du Cloud (A2 et A3)

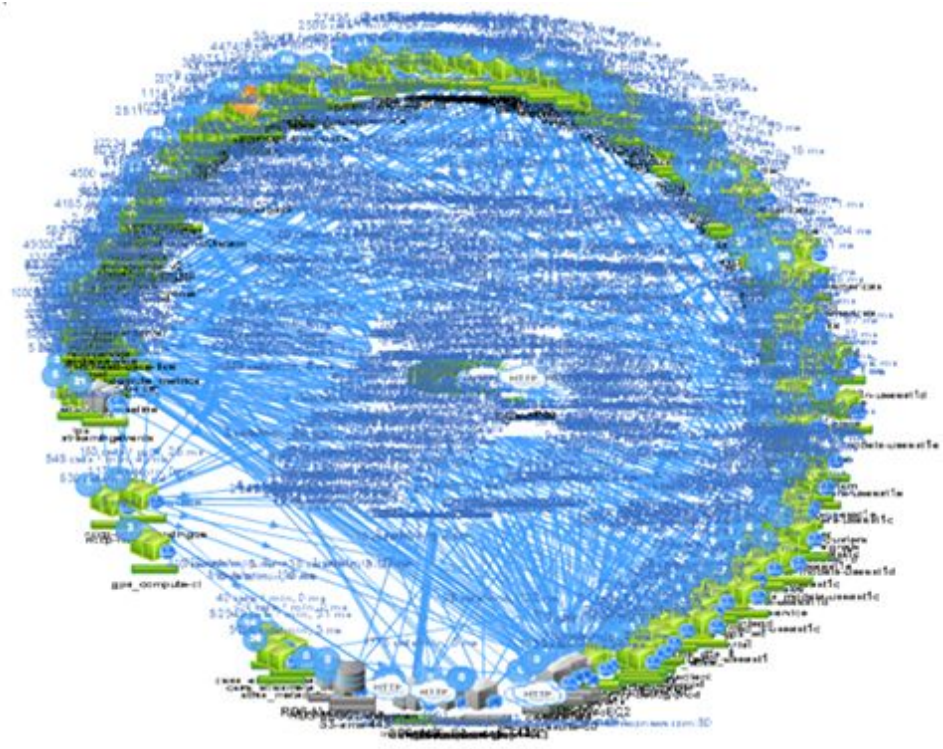
https://github.com/orgs/Netflix/repositories?type=all
https://jhohertz.github.io/netflixoss-slides/#/
https://spring.io/projects/spring-cloud



Source : https://microservices.io/patterns/microservices.html



amazon.com



NETFLIX

Architecture serverless

Attention ! Ce nom est trompeur !
"serverless" ne signifie pas "pas de serveur"

Dans cette architecture il y a en fait des serveurs mais gérés par une **tierce partie**.
L'idée est d'éviter à l'utilisateur d'avoir à gérer le backend et de se concentrer sur le frontend.

2 éléments :

- **FaaS** (Function-as-a-Service) - exécuter des petits bouts de code indépendants (fonctions) sur des ressources disponibles sur le Cloud
- **BaaS** (Backend-as-a-Service) - service du Cloud permettant de gérer les aspects backend (gestion des bases de données, de la sécurité etc.)

Inconvénients : vendor lock-in, perte de contrôle sur ce qui est fait
Avantages : coûts réduits, simplicité

Exemples : AWS Lambda, Google Cloud functions, etc.

Pour votre curiosité

UE DevOps A2

Globalement l'UE DevOps va aller plus loin dans tous les concepts utilisés en architectures microservices.

- déploiement avec des conteneurs Docker et DockerCompose
- Infrastructure-as-Code (IaC)
- Orchestration de services
- etc.

Design d'applications connues

- Tinder
- Instagram
- Whatsapp

Google design interviews

- Exemple