

UE AD FIL A1

UE Architectures et microservices FISE A3 LOGIN

Introduction

2026-2027

Hélène Coullon



IMT Atlantique

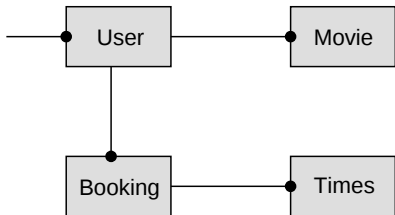
Bretagne-Pays de la Loire

École Mines-Télécom

Table of Contents

1. Architectures distribuées
2. Autres concepts associés
3. Quelques grands types d'architectures distribuées
4. Et cette UE ?
5. Déroulement de l'UE

Architectures distribuées



- Architecture **modulaire** fait de **composants** qui ont des **rôles** qui leur sont propres
- Les composants sont **distribués sur des machines différentes dans le réseau**
- Les composants **communiquent** les uns avec les autres par échange de messages sur le réseau
- Les composants peuvent être écrits dans des **langages** différents mais doivent utiliser les **même protocoles de communication**

- Séparer les fonctionnalités et les rôles n'est pas spécifique au distribué !
 - modules
 - fonctions
 - classes
 - paquets
 - etc.
- Mais pour faire du distribué il faut séparer clairement les fonctionnalités !
- Ce qui est spécifique au distribué c'est de placer les composants sur des machines distantes et de les faire communiquer via le réseau.

Architecture monolithique

Une architecture distribuée vient en opposition d'une architecture **monolithique**

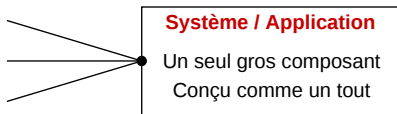
Mais une architecture monolithique peut être **modulaire** (3-tiers, hexagonale) !

Avantages

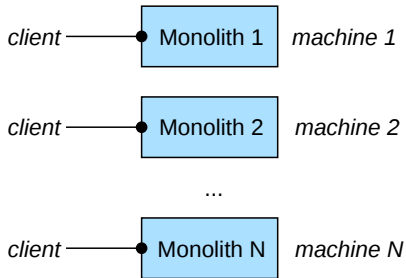
- Conception plus simple
- Pas de communications réseau

Inconvénients

- Redéploiement total si changements
- Peu tolérant aux pannes
- Peu scalable
- Pas de distribution des modules ou fonctionnalités (pas Cloud-native)



Architecture monolithique = 1 seule machine ?



Même si le système suit une architecture monolithique

- Il peut être déployé sur **plusieurs machines** !
- Chaque instance (ou déploiement) fonctionne comme un tout unique **sans communication** avec une autre.
- On peut répartir la charge avec un **load-balancer** entre chaque instance.
 - **scalabilité horizontale**

Avantages

- Modulaire
 - séparation des préoccupations, flexibilité, maintenabilité
 - réutilisation de code facilité
- Flexible et évolutif
 - mise à jour par éléments
 - déploiement continu
 - changements (reconfiguration) dynamique
- Tolérance aux pannes
- Scalabilité

Inconvénients

- Plus difficile à concevoir : API, réseau, sécurité etc.

Les architectures distribuées aujourd'hui

Désormais les architectures distribuées sont inévitables !

Infrastructures distribuées

- Clusters ou grappes de calculs
- Informatique à la demande (Cloud, Fog/Edge computing)
- Systèmes IoT et cyber-physiques
- Internet

Applications distribuées

- Applications web
- Applications mobile
- Applications IoT (villes intelligente, usines intelligentes)
- Applications liées à la 5G (voitures autonomes etc.)

Autres concepts associés

- Local : événements ou actions arrivant à un unique endroit
- Réparti (ou distribué) : événements ou actions arrivant à plusieurs endroits

*Dans une architecture distribuée, chaque composant peut être exécuté sur des ressources différentes réparties sur le réseau. Certaines de leurs actions sont **locales** mais d'autres sont **communes et réparties entre les composants**.*

Machine1 : Action1 et Action 2

Machine2 : Action2 et Action3

Machine3 : Action3 et Action4

- **Séquentiel** : des actions ordonnées totalement en suivant un temps linéaire
- **Parallèle** : des actions ordonnées partiellement en suivant des temps parallèles

*Dans une architecture distribuée, chaque composant exécute des actions qui lui sont propres, il y a donc nécessairement du **parallélisme entre les actions des différents composants**.*

Composant1 : Action1 $\rightarrow$ Action2 $\rightarrow$ Action 3 $\rightarrow \dots \rightarrow$ ActionN

Composant2 : Action1' $\rightarrow$ Action2' $\rightarrow$ Action3'

On parle de **concurrency** lorsque le parallèle doit devenir séquentiel, autrement dit lorsque l'on a besoin d'une **coordination temporelle** entre les actions.

Par exemple lorsque 2 actions parallèles accèdent à la même ressource.

→ nécessité d'accès entrelacés à la ressource, ce qui revient à du séquentiel

Mémoire partagée et distribuée

- Une mémoire est partagée lorsque le **même adressage mémoire** est utilisé entre plusieurs processus, composants, agents etc.
 - Ex : architecture multi-core et RAM de nos ordinateurs
 - **Pas de communication nécessaire** entre les éléments qui accèdent à la ressource mais une **coordination** est obligatoire pour conserver des **informations cohérentes**
 - Problème de **concurrency**
- Une mémoire est distribuée lorsque l'**adressage mémoire est différent** pour les processus, composants, agents etc.
 - Ex : plusieurs machines sur un réseau
 - Besoin de faire **communiquer** les éléments pour **accéder à une ressource distante**
 - **Echange de messages**

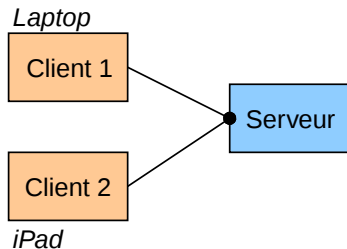
Dans cette UE nous nous intéressons à des mémoires distribuées et au passage de messages entre les composants

Communications synchrones et asynchrones

- Une communication **synchrone** nécessite que les éléments concernés par la communication soient prêts à communiquer en même temps
 - **synchronisation temporelle** des entités distribuées
- Une communication **asynchrone** permet aux éléments concernés par la communication de continuer ce qu'ils ont à faire en attendant que l'information utile leur soit transmise
 - notion d'**émetteur** et de **récepteur**
 - une absence de synchronisation temporelle ne signifie pas une absence de dépendance (attente d'une information)

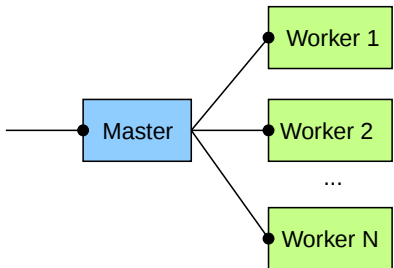
Quelques grands types d'architectures distribuées

Architecture Client/Serveur



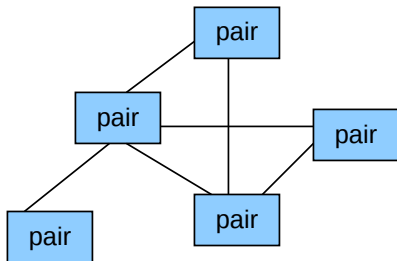
- Séparation entre les composants qui
 - **consommant** des ressources ou services
 - clients
 - **fournissent** des ressources ou services
 - serveurs
- Très classique et très utilisé
 - Ex. architectures client/serveur : Google drive, Dropbox etc.
- Frontend / backend

Architecture Master/Worker



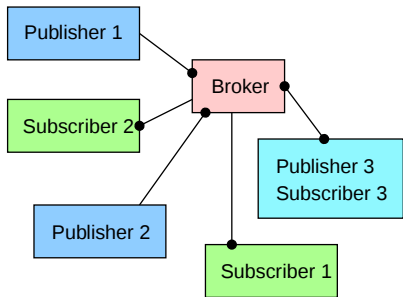
- Separation du travail entre les composants qui
 - **travaillent** : workers
 - réception de tâches
 - exécution de tâches
 - notification de fin de tâche
 - **coordonnent** le travail des workers : masters
 - réception des requêtes
 - division du travail en tâches
 - distribution des tâches aux workers
- Ici on se place plutôt côté serveur (backend)
- **Attention** diviser le travail n'est pas trivial !
- Ex. d'architecture master/worker: Apache Spark

Architecture pair-à-pair (p2p)



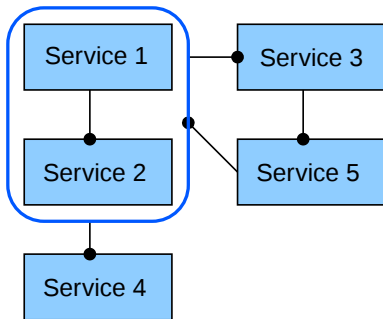
- Séparation du travail ou des ressources
- Pas de privilèges : tous les **pairs sont identiques** et à la fois client et serveur
 - fournir et consommer des ressources
 - mais il y a souvent une notion de leader pour permettre les connexions de nouveaux pairs à un réseau p2p
- On parle d'architecture complètement **décentralisée**
- Architecture popularisée dans les années 2000
 - Napster : partage décentralisé de fichiers musicaux
- Très tolérant aux pannes
 - pas de point central, le leader peut être changé

Architecture à événements



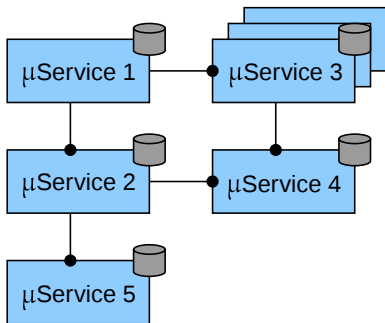
- Les composants sont tous vus comme des éléments qui
 - envoient des événements sur leurs activités locales : **publisher**
 - s'abonnent et reçoivent des événements des autres composants : **subscriber**
- Pas de communication synchrones entre les composants (asynchrone)
- Gestion tierce des événements et de leur diffusion : **broker**
- Ex. technos : systèmes de publish-subscribe, message broker
- Ex. architectures événements : IoT

Architecture orientée services (SOA)



- Les composants sont considérés comme des **services**
- Chaque service peut utiliser d'autres services
 - dépendances entre services
 - hiérarchie de services
- Les services sont vus comme des boîtes noires
 - Concepts de courtier entre services
 - Concept de découverte de services
 - Concept d'**interfaces** ou de **contrat** pour savoir ce qu'offre un service

Architecture micro-services



- Très largement **hérité du SOA**
- Architecture dite "Cloud native"
 - se prête très bien à la conteneurisation
 - se prête très bien au déploiement sur le Cloud
- "micro" ne veut pas forcément dire petit
 - un micro-service peut être conséquent
 - mais un micro-service est difficilement découposable en sous éléments
- Dépendances plates, pas de hiérarchie
- Même besoins qu'en SOA : découverte, **interfaces** etc.
- Ex. Architectures micro-services : Netflix, Twitter

Ce terme est plutôt lié en particulier à l'utilisation du Cloud

Attention ! Ca ne veut pas dire qu'il n'y a pas de serveurs !

- Serveurs gérés par une tierce partie
- Eviter à l'utilisateur de gérer le backend
- 2 services types dans le Cloud :
 - FaaS : Function-as-a-Service
 - BaaS : Backend-as-a-Service (BD, security etc.)
- Ex. technos : AWS lambda, Google Cloud functions

Certaines de ces architectures sont compatibles les unes avec les autres !

Exemples

- On peut avoir une architecture pair-à-pair qui utilise un système événementiel
- On peut avoir une architecture micro-services avec du client serveur ou du master/worker
 - Ex : dropbox

Et cette UE ?

Les architectures orientées services et micro-services !

Pourquoi ?

- Très répandu et apprécié des entreprises
 - Web
 - IoT
 - Cloud
- Flexible et évolutif
- Séparation claire entre le développeur, le DevOps et l'opérateur
- Grand nombre d'outils pour gérer des applications micro-services
 - Docker, DockerCompose
 - Kubernetes
- Compatible avec les autres architectures

Focus sur le concept d'API

- API = Application Programming Interface
- Interfaces permettant de savoir ce que fournit un micro-service
- Contrat entre le fournisseur et le consommateur de services
- C'est l'API qui permet de découpler les différents services

Ce que nous allons faire ensemble

- Concevoir et implémenter une application micro-services en Python
- Définir différents type d'interfaces pour les micro-services
- Documenter les interfaces
- Utiliser Docker et Docker-compose
- Utiliser une BD NoSQL
- Développer votre propre projet

Déroulement de l'UE

Objectifs

Comprendre les concepts de développements de micro-services et apprendre à utiliser trois types d'API concrètes.

Les TP sont fait par groupes de 2 personnes (binôme à conserver!)

- Développer une application basique de 4 micro-services pour la gestion d'une salle de cinéma
- 2 versions de l'application et donc 2 repo Git sont attendues
 - version avec uniquement des API REST (documentation OpenAPI)
 - version mixte en REST, GraphQL et gRPC
- Le rendu des codes et une présentation orale sont évalués
- Evaluation écrite individuelle à chaque début de séance (sauf première)

Objectifs

Concevoir votre propre application, appliquer ce qui a été vu en TP, développer dans plusieurs langages et utiliser une ou des APIs externes

Les projets sont faits en associant 2 binômes (4 étudiant.e.s !)

- Concevoir une application micro-services de votre choix
- Travailler en équipe et sous Git
- Le rendu des codes et une présentation orale sont évalués

- BC4 CONCEVOIR UNE APPLICATION SUIVANT UNE ARCHITECTURE DE (MICRO-)SERVICES EN COMPRENANT ET CHOISISANT LES SOLUTIONS ADAPTÉES À LA PROBLÉMATIQUE POSÉE PAR LE CLIENT OU L'UTILISATEUR
- BC5 PRODUIRE UNE APPLICATION RÉPONDANT À DES SPÉCIFICATIONS EN SUIVANT UNE ARCHITECTURE DE (MICRO-)SERVICES EN PYTHON/FLASK ET EN TYPESCRIPT EN UTILISANT LES API REST, GRAPHQL ET GRPC