Architectures distribuées

Master 1 génie de l'informatique logicielle

Présenté par : Lydia KAHOUADJI & Luc Perin PANTA

20 mars 2024

Sommaire

1. Contexte
2. Definitions
3. Compréhension de l'Architecture Monolithique
4. Exploration des Architectures Microservices
5. Étapes de Migration
6. Cas d'usage
7. Introduction à la Communication
8. Brokers de Données

Contexte

Problème

Les exigences des applications modernes sont très dynamiques en raison de l'évolution fréquente des besoins des utilisateurs/clients, ces exigences sont nécessaires pour que les services soient fournis en continu aux utilisateurs. Il faut donc une architecture logicielle qui prenne en charge le déploiement et la livraison fréquente de ces services.

Historique

Le terme " microservice " est apparu pour la première fois en 2011 lors d'un atelier d'architecture logicielle près de Venise[1]

Définitions

Architecture Monolithique

une architecture monolithique est une architecture logicielle dans laquelle les composants sont interconnectes et interdépendants.

Architecture SOA

L'architecture orientée services (SOA) est un style architectural qui prend en charge l'orientation vers les services. L'orientation service est une façon de de penser en termes de services et d'applications basées sur les services.

Architecture Microservice

une architecture microservice est une approche visant à développer une application unique sous la forme d'une suite de petits services, chacun s'exécutant dans son propre processus et communiquant avec des mécanismes légers une API de ressources HTTP.

Illustration

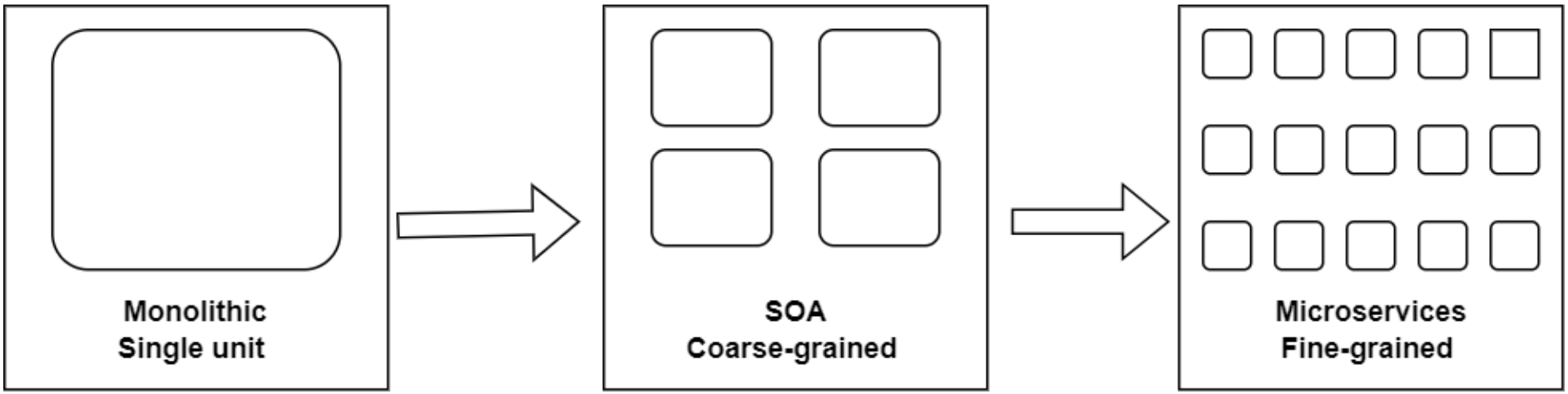


Figure – Comparatif architectural

Compréhension de l'Architecture Monolithique

- **Caractéristiques et composants d'une architecture monolithique :**
  - Une architecture monolithique regroupe tous les composants logiciels en un seul bloc.
  - Les différents modules et fonctionnalités sont interconnectés et partagent souvent une même base de code.
  - Il existe une seule base de données pour l'ensemble de l'application.
  - Les interactions entre les composants se font généralement en mémoire, facilitant la communication directe.
- **Avantages de l'approche monolithique :**
  - *Simplicité de déploiement :* En raison de sa nature intégrée, le déploiement d'une application monolithique est souvent plus simple.
  - *Facilité de développement initial :* Il est généralement plus rapide de développer une application monolithique au début d'un projet.
  - *Débogage simplifié :* La cohésion entre les composants facilite le processus de débogage.

Limitations et Défis des Architectures Monolithiques

- **Évolutivité limitée :**
  - L'ajout de nouvelles fonctionnalités peut devenir **difficile** à mesure que l'application **grossit**.
  - La croissance de l'application peut entraîner une **complexité accrue** et des défis liés à la gestion des modules existants.
- **Déploiements lourds :**
  - Les mises à jour nécessitent souvent le **déploiement de l'ensemble de l'application**.
  - Cela peut entraîner des **interruptions de service** et des périodes de **downtime** plus longues pendant les mises à jour.
- **Risques de maintenance :**
  - Les modifications peuvent avoir des **répercussions inattendues** sur d'autres parties de l'application.
  - La **maintenance** peut devenir **complexe** et nécessiter une **gestion minutieuse** pour éviter les conflits.

Exploration des Architectures Microservices

- **Principes clés des architectures microservices :**
  - Les architectures microservices adoptent une approche de développement basée sur la décomposition en services distincts.
  - Chaque service représente une fonctionnalité spécifique de l'application et peut être développé, déployé et évolué indépendamment.
  - La communication entre les services se fait généralement via des API (Interfaces de Programmation Applicative) bien définies.
  - Chaque service peut avoir sa propre base de données, favorisant l'isolation des données.
- **Avantages de l'approche microservices :**
  - *Évolutivité :* Les microservices permettent une évolutivité horizontale en ajoutant simplement des instances du service nécessaire.
  - *Déploiement indépendant :* Les mises à jour peuvent être déployées service par service, minimisant les interruptions.
  - *Technologies diverses :* Chaque service peut être développé avec la technologie la mieux adaptée à sa fonction.

Architecture monolithique vs architecture microservice

Architecture microservice

VS

Architecture monolithique

Comparaison - Flexibilité

- Les architectures microservices offrent une **flexibilité supérieure** en permettant des développements **indépendants**.
- Chaque service peut être **développé, déployé et mis à l'échelle** de manière autonome, offrant une agilité accrue.
- *Exemple :* Intégration continue et déploiement continu (CI/CD) plus efficace avec des services autonomes.

Comparaison - Complexité du Déploiement

- Bien que le déploiement des microservices soit **simplifié**, la **gestion de multiples services** nécessite une **coordination efficace**.
- Des **stratégies de déploiement progressif** et d'**orchestration** sont nécessaires pour maintenir la cohérence du système.
- *Exemple :* Utilisation d'outils d'orchestration tels que Kubernetes pour coordonner le déploiement de microservices.

Comparaison - Isolation des Erreurs

- Les architectures microservices permettent une **isolation des erreurs**.
- Les erreurs dans un service spécifique n'affectent pas nécessairement les autres services, améliorant ainsi la **robustesse globale** du système.
- *Exemple :* Un service défectueux n'impacte pas l'ensemble de l'application, facilitant la détection et la résolution des problèmes.

Comparaison - Scalabilité

- Les architectures microservices permettent une **scalabilité granulaire**.
- Chaque service peut être **mis à l'échelle indépendamment** en fonction de ses besoins, optimisant l'utilisation des ressources.
- *Exemple :* Mise à l'échelle horizontale d'un service spécifique en cas de charge accrue.

Étapes de Migration

Étapes de Migration - Évaluation de l'existant

- Analyse approfondie de l'architecture monolithique existante.
- Identification des services candidats à la migration vers une architecture microservices.
- Évaluation des dépendances et des interactions entre les composants.
- Objectif : Comprendre les défis potentiels et déterminer les premières étapes de la migration.

Étapes de Migration - Décomposition du Monolithe et Création de Services

- Utilisation de techniques de découpage pour diviser le monolithe en services plus petits.
- Gestion des dépendances entre les nouveaux services.
- Conception et implémentation des premiers microservices indépendants.
- Mise en place de la communication entre ces services.
- Objectif : Créer une base solide pour une architecture microservices évolutive.

Étapes de Migration - Gestion des Données

- Migration de la base de données monolithique vers une architecture distribuée.
- Considérations sur la cohérence des données entre les microservices.
- Objectif : Assurer une transition fluide des données tout en maintenant l'intégrité et la cohérence du système.

Étapes de Migration - Tests, Déploiement Graduel, Surveillance et Ajustements

- Mise en place de stratégies de test pour garantir la stabilité du système.
- Approches pour le déploiement progressif des microservices.
- Surveillance en temps réel des performances et des erreurs.
- Ajustements en fonction des retours d'expérience.
- Objectif : Garantir la qualité, la stabilité et l'optimisation continue du système post-migration.

Cas d'usage

- **Architecture Monolithique :**
  - Idéale pour les **petites applications** avec une **complexité limitée**.
  - Facilite le **développement initial** grâce à une structure intégrée.
  - Convient aux projets où la **simplicité** et la **maintenance** sont des priorités.
  - Avantageux lorsque la **gestion des dépendances** entre les composants est moins critique.
- **Architecture Microservices :**
  - Adaptée aux **applications complexes** avec **diverses fonctionnalités** et **besoins d'évolutivité**.
  - Permet une **évolutivité horizontale**, chaque service étant indépendant.
  - Convient lorsque la **scalabilité**, la **flexibilité** et le **déploiement indépendant** sont essentiels.
  - Avantageuse dans des environnements où la **diversité technologique** est nécessaire pour différentes parties de l'application.

Introduction à la Communication dans les Architectures Logicielles

Objectif

Explorer les mécanismes de communication au cœur des architectures logicielles.

Contexte

La manière dont les composants d'un système interagissent est cruciale pour son bon fonctionnement.

Importance

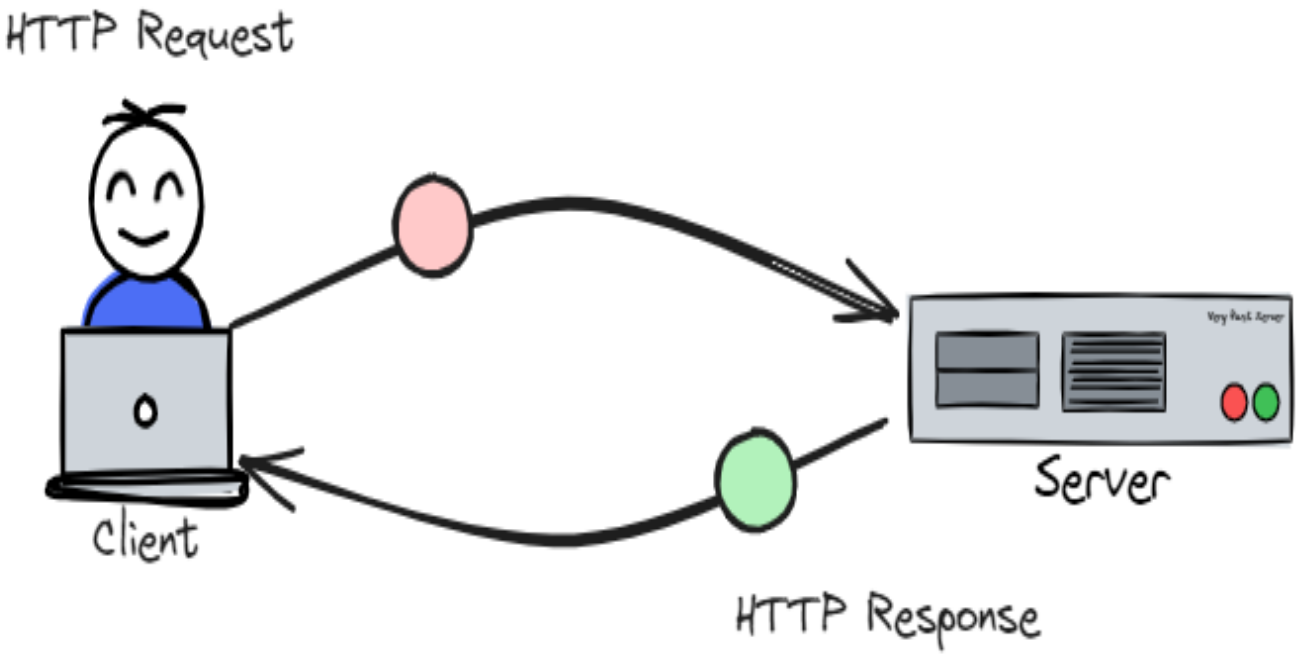
Comprendre les méthodes de communication permet de concevoir des systèmes robustes et évolutifs.

Méthodes de Communication - RESTful

- **REST (Representational State Transfer) :**

Approche Architecturale

- **REST** est une approche architecturale pour les **services web** qui met l'accent sur la simplicité et la scalabilité.
- Basé sur le principe de **Representational State Transfer**, qui implique la représentation d'états de ressources.



Méthodes de Communication - RESTful

- **REST (Representational State Transfer) :**

Communication Stateless

- La communication entre le client et le serveur est **stateless**, ce qui signifie que chaque requête du client au serveur doit contenir toutes les informations nécessaires à la compréhension et au traitement de cette requête.
- Cela simplifie le **scalabilité** du système car chaque requête peut être traitée indépendamment.

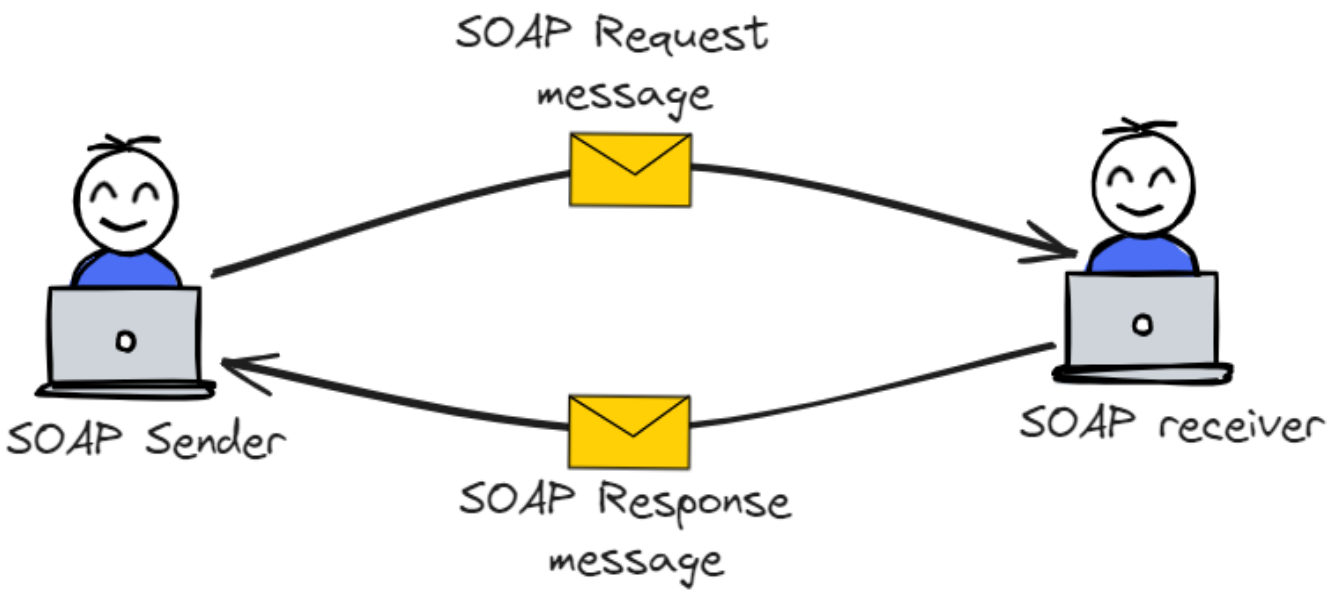
Opérations Standard

- Les opérations standard, telles que **GET, POST, PUT, DELETE**, sont utilisées pour **manipuler les ressources** sur le serveur.
- Par exemple, **GET** est utilisé pour récupérer une ressource, **POST** pour créer une nouvelle ressource, **PUT** pour mettre à jour une ressource existante, et **DELETE** pour supprimer une ressource.

- **SOAP (Simple Object Access Protocol) :**

Protocole de Communication

- **SOAP** est un protocole de communication standardisé utilisé pour échanger des informations structurées dans le développement de services web.
- Il définit une **enveloppe XML** pour structurer les messages, facilitant l'échange d'informations entre systèmes hétérogènes.



Méthodes de Communication - SOAP

- **SOAP (Simple Object Access Protocol) :**

Lourd et Rigide

- Comparé à RESTful, SOAP est souvent considéré comme **plus lourd et rigide** en raison de son utilisation d'XML et de sa structure formelle.
- Il peut offrir cependant une **sémantique plus stricte** dans la communication entre systèmes.

Opérations Définies

- Les opérations sont définies dans une **interface**, souvent décrite dans un langage de description de service tel que WSDL (Web Services Description Language).
- Les messages sont généralement envoyés via des protocoles de transport tels que HTTP ou SMTP.

Méthodes de Communication - GraphQL et gRPC

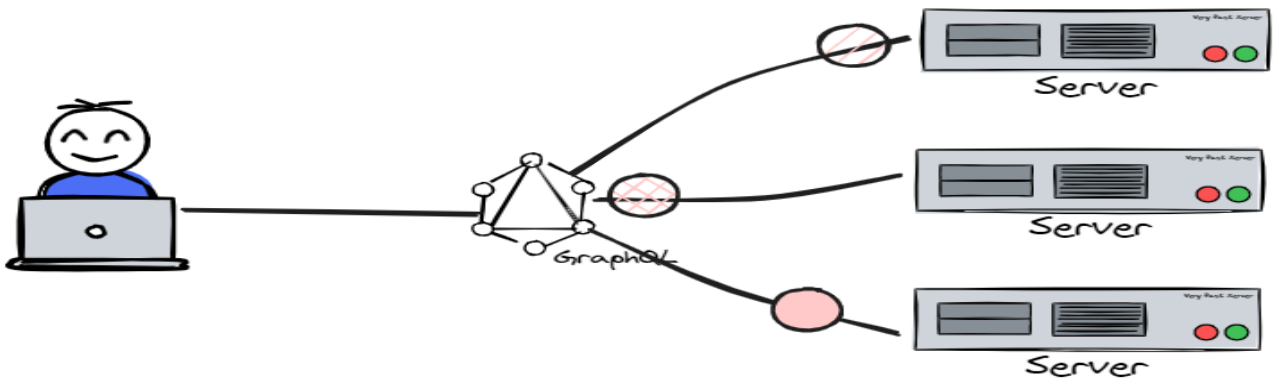
GraphQL

- Langage de requête flexible.
- Utilise un **seul endpoint** pour toutes les opérations.
- Permet aux clients de spécifier les champs exacts dont ils ont besoin.
- Souvent utilisé pour les applications web et mobiles.

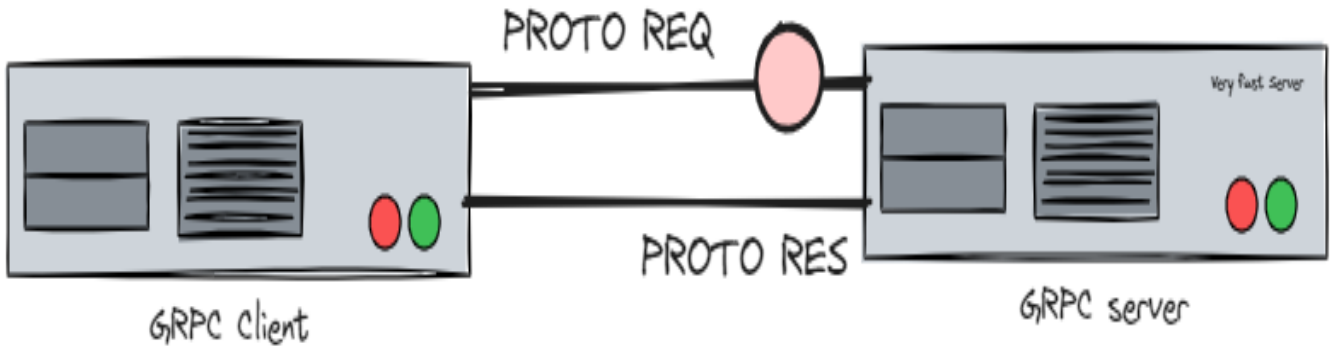
gRPC

- Framework RPC développé par Google.
- Utilise le protocole **HTTP/2** et Protocol Buffers.
- Les services sont définis à l'aide de fichiers proto.
- Performances élevées, interopérabilité entre différentes plates-formes.

GraphQL



GRPC



Communication Synchrone - Définition et Exemple

Définition

La **communication synchrone** implique qu'un client **attend une réponse immédiate** après avoir envoyé une requête au serveur.

Exemple

Requêtes HTTP traditionnelles utilisent une communication synchrone.

Communication Synchrone - Avantages et Inconvénients

Avantages

- **Simplicité d'implémentation** : Facile à comprendre et à mettre en œuvre.
- **Réponse immédiate** : Convient aux interactions nécessitant des réponses instantanées.

Inconvénients

- **Bloquant** : Le client est **bloqué en attente** de la réponse, pouvant entraîner une **latence perceptible**.
- **Sensibilité aux pannes** : Si le serveur est indisponible, le client ne peut pas progresser.

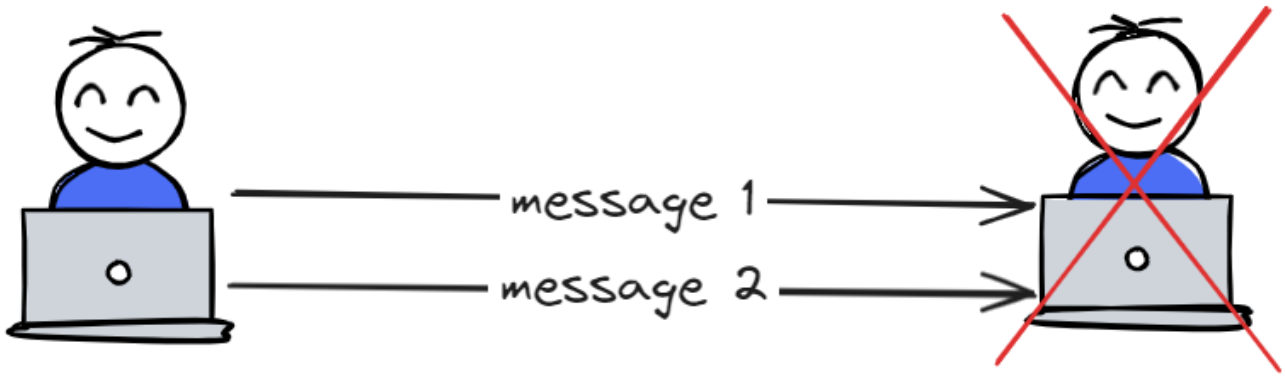
Communication Asynchrone - Définition et Exemple

Définition

La **communication asynchrone** permet au client d'**envoyer une requête** et de **continuer son travail** sans attendre une réponse immédiate.

Exemple

Utilisation de **queues de messages** pour des tâches différées.



Communication Asynchrone - Avantages et Inconvénients

Avantages

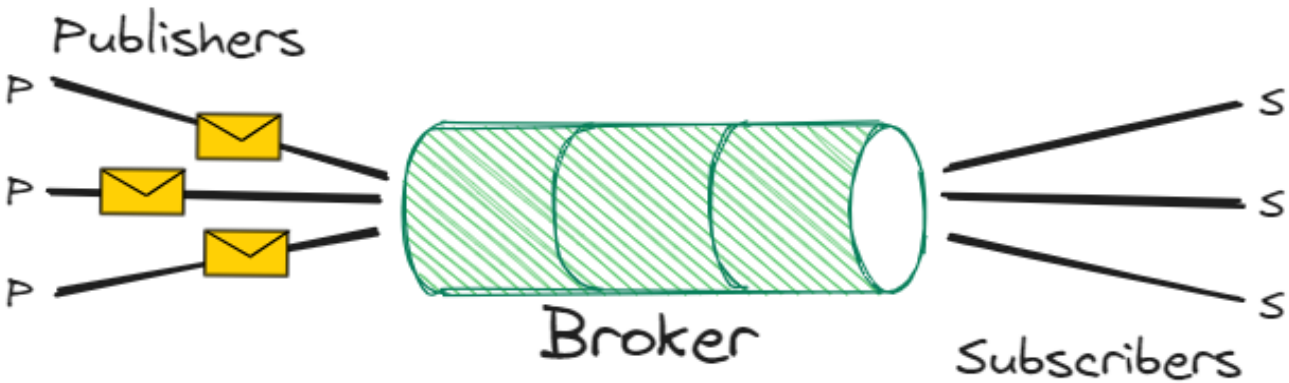
- **Non-bloquant** : Le client peut **continuer ses opérations** sans attendre la fin de la requête.
- **Évolutivité** : Convient aux **tâches asynchrones** et aux systèmes distribués.

Inconvénients

- **Complexité** : La mise en œuvre peut être plus **complexe** en raison de la gestion des réponses différées.
- **Ordonnancement** : Nécessite une gestion appropriée de l'ordonnancement des tâches.

Broker de Données - Définition et Exemples

Définition : Un **broker de données** est un composant logiciel qui facilite la communication et l'échange de données entre différentes parties d'un système distribué. Il agit comme un intermédiaire qui permet à différents services, modules ou microservices de partager des informations de manière asynchrone.



Utilisation de Brokers de Données - Monolithique

- **Peu de Besoin pour les Brokers de Données :**
  - Dans une architecture **monolithique**, l'utilisation de **brokers de données** est **souvent limitée**.
  - La communication se fait généralement **directement via la base de données partagée**.
  - Avantages :
    - **Simplicité** : La communication directe avec la base de données centralisée simplifie les échanges.
    - **Consistance des Données** : Les modifications sont immédiatement reflétées dans la base de données partagée.
  - Inconvénients :
    - **Centralisation des Échanges** : Toutes les interactions passent par la base de données, pouvant créer des points de congestion.
    - **Limitations de Scalabilité** : L'évolutivité peut être limitée en fonction de la capacité de la base de données.

Utilisation de Brokers de Données - SOA

- **Brokers de Messages plus Courants :**
  - Dans une architecture **SOA**, l'utilisation de **brokers de messages** est **plus courante**.
  - Les services communiquent via des **queues de messages** pour faciliter la communication asynchrone.
  - Avantages :
    - **Désaccouplement** : Les services peuvent échanger des informations sans être directement couplés.
    - **Scalabilité** : Les queues de messages permettent une **gestion asynchrone des tâches**, favorisant la scalabilité.
  - Inconvénients :
    - **Complexité** : L'implémentation et la gestion des brokers de messages peuvent ajouter de la complexité.
    - **Latence Potentielle** : La communication asynchrone peut introduire une certaine latence.

Utilisation de Brokers de Données - Microservices

- **Brokers de Messages Fréquemment Utilisés :**
  - Dans une architecture **microservices**, l'utilisation de **brokers de messages** est **fréquente**.
  - Les microservices interagissent de manière **asynchrone** via des **queues de messages**.
  - Avantages :
    - **Désaccouplement Extrême** : Chaque microservice fonctionne de manière indépendante.
    - **Scalabilité Maximale** : La communication asynchrone permet une **scalabilité optimale** des microservices.
  - Inconvénients :
    - **Complexité Accrue** : La gestion des interactions asynchrones entre microservices peut être complexe.
    - **Latence Potentielle** : La communication asynchrone peut introduire une certaine latence.

Impact sur la Gestion des Données - Monolithique

- **Base de Données Centralisée :**
  - Dans une architecture **monolithique**, l'approche privilégiée est d'avoir une **base de données centralisée**.
  - Tous les modules partagent la **même base de données**, simplifiant la gestion des données.
  - Avantages :
    - **Consistance des Données** : Toutes les données sont centralisées, assurant une cohérence.
    - **Facilité de Requêtes Complexes** : Les requêtes peuvent impliquer plusieurs tables sans communication inter-service.
  - Inconvénients :
    - **Dépendance Forte** : Toute modification de la structure de la base de données peut impacter l'ensemble de l'application.
    - **Évolutivité Limitée** : L'ajout de nouvelles fonctionnalités peut devenir complexe à mesure que l'application grossit.

Impact sur la Gestion des Données - SOA

- **Bases de Données Partagées entre Services :**
  - Dans une architecture **SOA**, les services peuvent partager des **bases de données**, mais chaque service peut également avoir **sa propre base de données**.
  - Favorise une **plus grande indépendance** entre les services.
  - Avantages :
    - **Isolation des Données** : Les services peuvent gérer leurs propres données sans dépendre fortement des autres services.
    - **Évolutivité Améliorée** : Chaque service peut évoluer indépendamment, facilitant l'ajout de nouvelles fonctionnalités.
  - Inconvénients :
    - **Complexité de la Communication** : Les services doivent s'accorder sur les schémas et les formats de données.
    - **Gestion des Transactions** : Assurer la cohérence des données peut nécessiter des mécanismes de gestion des transactions distribuées.

Impact sur la Gestion des Données - Microservices

- **Chaque Microservice avec sa Propre Base de Données :**
  - Dans une architecture **microservices**, chaque microservice **dispose de sa propre base de données**.
  - Favorise une **indépendance totale** entre les microservices.
  - Avantages :
    - **Indépendance Complète** : Chaque microservice est libre de choisir sa technologie de base de données et son schéma.
    - **Évolutivité Maximale** : Chaque microservice peut évoluer et être déployé indépendamment des autres.
  - Inconvénients :
    - **Consistance des Données** : Maintenir la cohérence des données entre microservices peut être complexe.
    - **Complexité de la Gestion des Données** : La gestion des relations entre les microservices peut nécessiter des stratégies spécifiques.