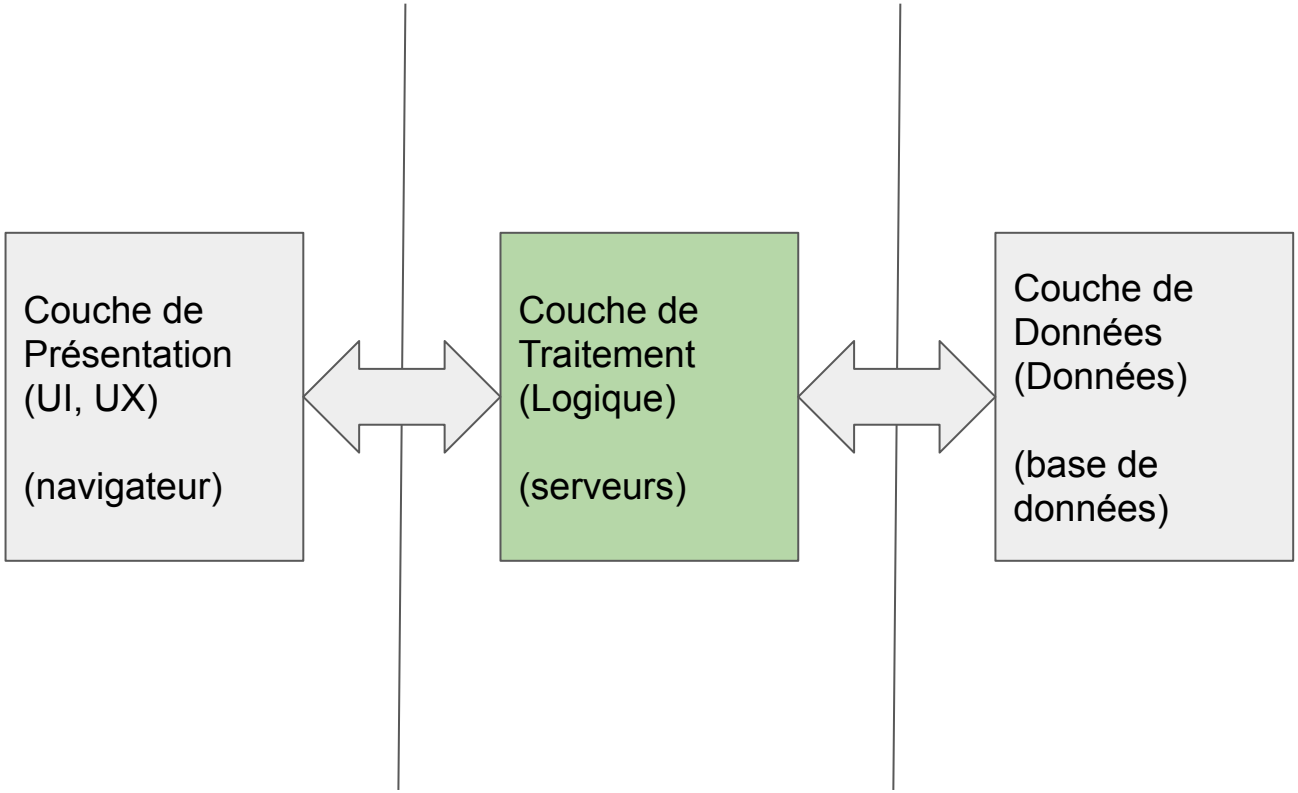
Couche de traitement

Services et Communication

Plan et objectifs

- Identifier les éléments faisant partie de la couche de traitement
- Comprendre le principe de séparation des responsabilités
- Explorer deux architectures possibles pour la couche de traitement
  - Monolithique
  - Par Services
- Comprendre les différentes formes de communication entre services
- Explorer la sérialisation des données et comprendre ses objectifs
- Les tâches asynchrones
- Exemples de service


Couche de Présentation (UI, UX)
(navigateur)
Couche de Traitement (Logique)
(serveurs)
Couche de Données (Données)
(base de données)

Couche de traitement

La couche de traitement contient la **logique d'affaire** (ou de domaine) de l'application.

Cette couche **ne doit pas se soucier de la présentation** des données (UI/UX).

Idéalement cette couche est ***sans état*** (l'état de l'interface usager est géré par la couche de présentation et la persistance des données est gérée par la couche de données), ce qui facilite les tests et la mise à l'échelle de l'application.

Préoccupations de la couche de traitement

Comme la couche de traitement est la destination des toutes les requêtes clients (usagers ou autres systèmes), elle doit être

- disponible
  - mesuré par le % de temps ou le service est accessible et utilisable
- fiable (fréquence)
  - mesuré par le nombre de pannes
- rapide
  - mesuré par le temps de réponse

Il faut aussi considérer la sécurité

- des données
- intégrité du système

Accord de niveau de service (SLA)

L’accord de niveau de service est un document qui détaille les engagements du fournisseur envers ses clients concernant la disponibilité, fiabilité et performance.

L’accord peut aussi détailler des contraintes quant au temps moyen de réponse de l’équipe en réaction à des problèmes, au temps de réparation des pannes, au niveau de sécurité du système, etc.

Un exemple de métrique typique est le temps de disponibilité.

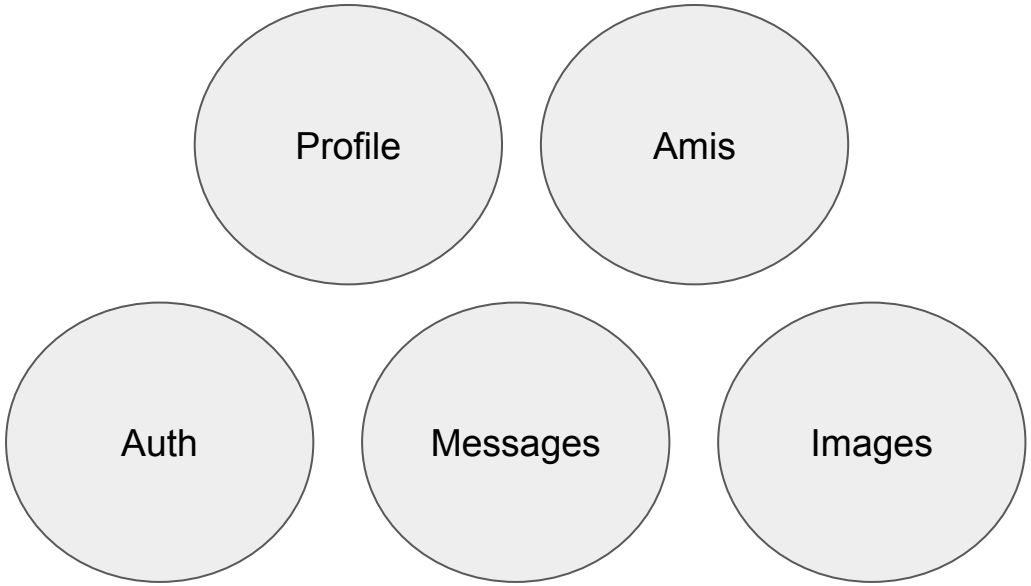
Temps de panne permis en fonction de la cible de disponibilité

disponibilité	**99%**	**99,9%**	**99,99%**
semaine	*1h 40m 48s*	*10m 4s*	*1m 0s*
mois	*7h 18m 17s*	*43m 49s*	*13m 8s*
année	*7h 18m 17s*	*8h 45m 56s*	*52m 35s*

Séparation des responsabilités

Une bonne pratique pour développer la couche de traitement consiste à identifier les différentes fonctionnalités requises pour ensuite les regrouper en composants qui ont chacun une responsabilité bien définie (SRP).

'Gather together those things that change for the same reason, and separate those things that change for different reasons.' - Robert C Martin



Séparation des responsabilités

La séparation des responsabilités:

- structure le code logiquement
- isole les changements dans une section (module, service) spécifique
- permet à plusieurs équipe de travailler en parallèle en minimisant les interférences
- facilite les tests


MAIS COMMENT SÉPARER?
GLOBETROTTER
GLOBETROTTER
VOLVO
FM
VOLVO
FM
imgflip.com

Conception pilotée par le domaine (DDD)

À la base, la **conception pilotée par le domaine** (domain driven design) est une technique de modélisation qui consiste à utiliser les connaissances (et le vocabulaire) des experts du domaine pour modéliser le logiciel.

- permet de capturer les concepts, les entités et les processus
- permet de créer un vocabulaire commun (tech vs non-tech)
- structure le logiciel de façon à pouvoir y naviguer facilement
- lie le domaine et l'implémentation

Avant tout, on recherche **une responsabilité** (un domaine) bien définie, un **couplage léger** et une **forte cohésion**.

Architecture de la couche de traitement

Il existe différentes approches pour définir l'architecture de la couche de traitement. Les deux principales sont:

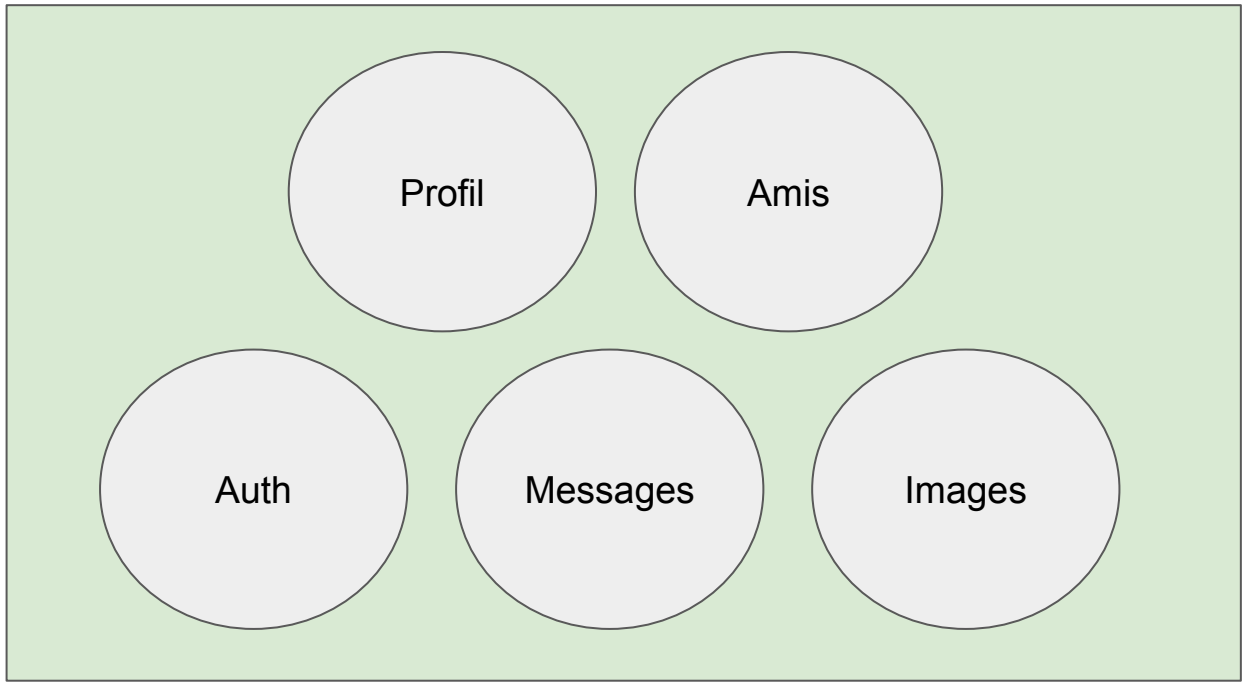
- architecture monolithique
- architecture par services

La décision du choix de l'architecture aura des répercussions à plusieurs niveaux:

- choix des technologies et des outils
- infrastructure
- et même la structure des équipes de développement (test, déploiement, etc)

Architecture monolithique

La couche de traitement consiste en un seul élément déployé. Cet élément contient l'ensemble des composants. Les composants (modules, packages) communiquent par via appels de fonction.



Architecture monolithique

Avantages

- simple à développer (initialement)
- simple à déboguer (tout se produit à l'intérieur du même processus)
- simple à tester (initialement)
- simple à déployer

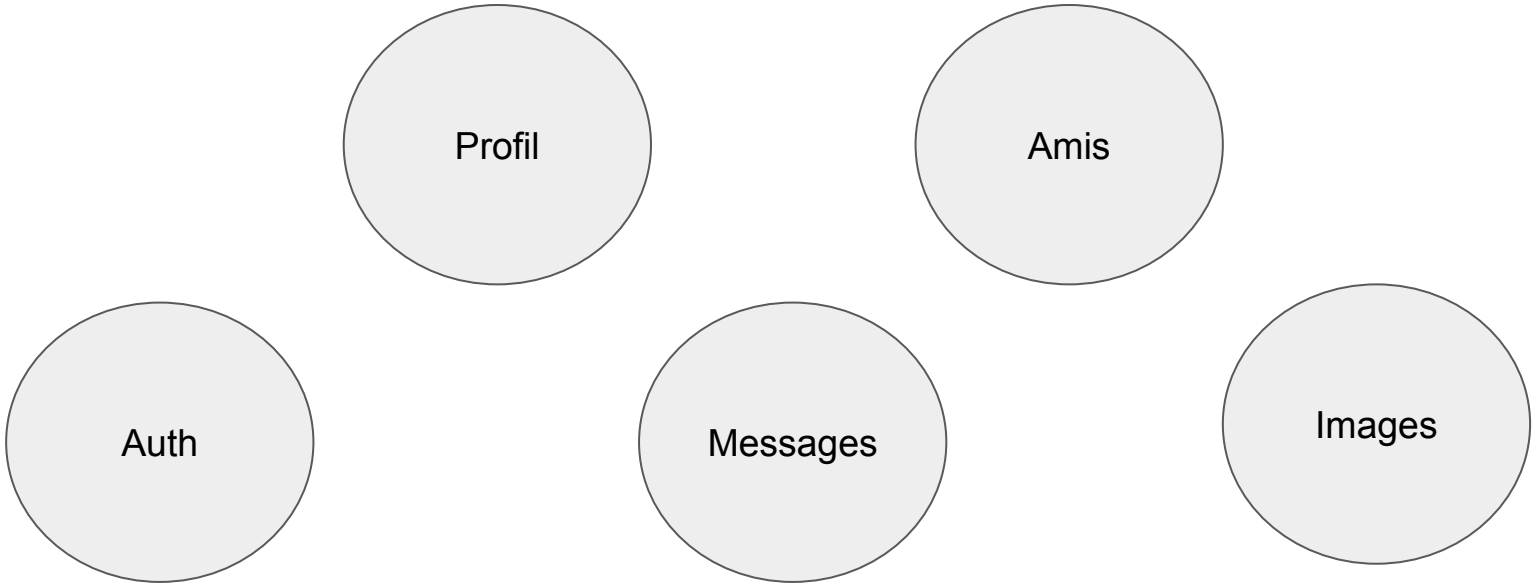
Désavantages

- le développement se complique lorsque la taille de l'équipe augmente
- même un petit changement demande de re-tester et re-déployer le système au complet (qui peut être un long processus)
- difficile d'éviter le code spaghetti (franchir les frontières entre les composants)
- difficile à faire évoluer graduellement (intégration de nouvelles technologies)

Architecture par services (microservices)

Un service est un composant déployable qui contient un ensemble de fonctionnalités qui sont exposé aux autres services via un API.

Dans une architecture par services, la couche de traitement consiste en plusieurs services interconnectés. L'exécution d'un scénario d'utilisation (use case) requiert la coordination d'un ou plusieurs services.



Architecture par services (microservices)

Avantages

- simple à mettre à l’échelle
- système plus résilient (la défaillance d’un service ne cause pas une panne complète)
- base de code plus simple (mais multiple)
- permet à plusieurs technologies de coexister
- déploiement granulaire

Désavantages

- moins rapide
- demande plus de ressources
- infrastructure plus complexe
- plus difficile à tester (de bout en bout)
- difficile à déboguer

Communication inter-services

Il existe différents patrons de communication entre les services:

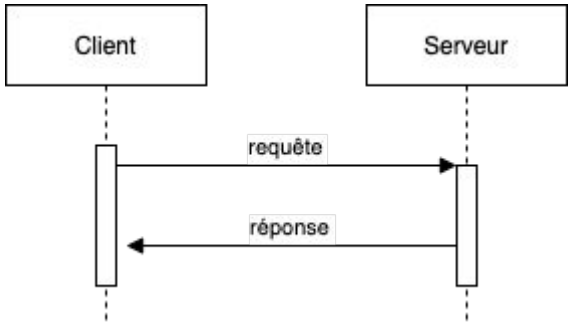
- requête-réponse (synchrone, e.g. HTTP)
- par messages (asynchrone, e.g. RabbitMQ)
- pair-à-pair (asynchrone, e.g. websocket, grpc*)

Communication Requête - Réponse

Typiquement synchrone mais peut aussi être asynchrone en utilisant des identificateurs de transaction.

Avantages:

- simple
- fiable
- gestion d’erreur simple



Désavantages:

- peu flexible
- moins performant (surtout si sync I/O)
- le serveur ne peut pas initier la communication

Attente active - *Polling*

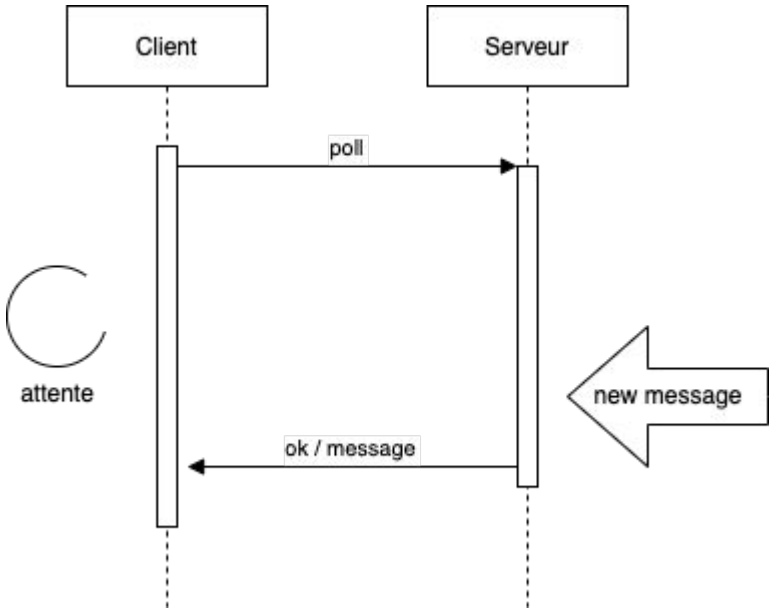
Mécanisme de communication qui permet d'utiliser un patron de communication de type requête-réponse pour vérifier si le serveur à des nouvelles informations à transmettre.

Avantages:

- simple

Désavantages:

- coûteux en terme de ressources
- délai (max = temps d'attente entre les requêtes)



Attente active longue - *Long Polling*

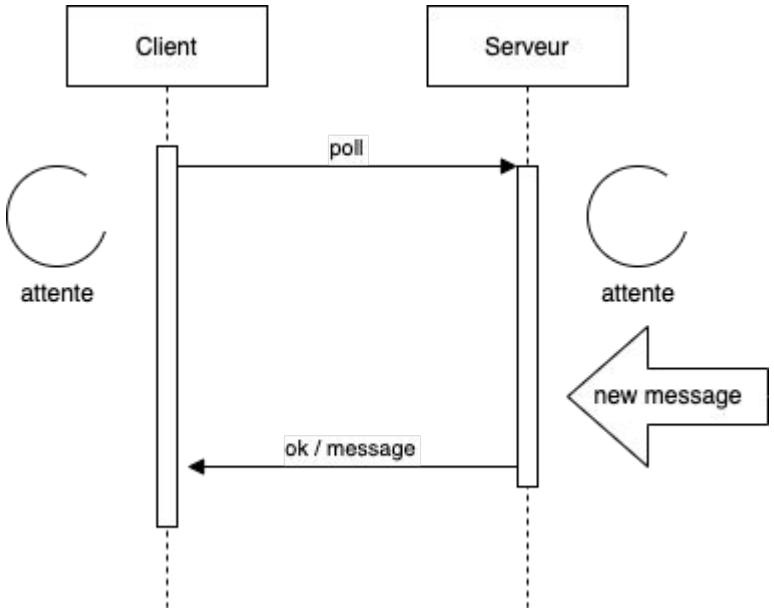
Mécanisme de communication qui permet d'utiliser un patron de communication de type requête-réponse pour vérifier si le serveur à des nouvelles informations à transmettre.

Avantages:

- simple
- minimise le délai

Désavantages:

- coûteux en terme de ressources
- gestion de déconnexion



Communication par messages / événements

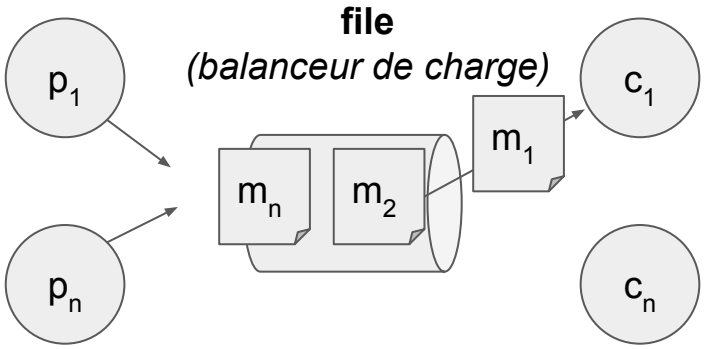
Les messages sont envoyés à un gestionnaire de messages (*message broker*).
Les receveurs écoutent sur la file pour recevoir les messages.

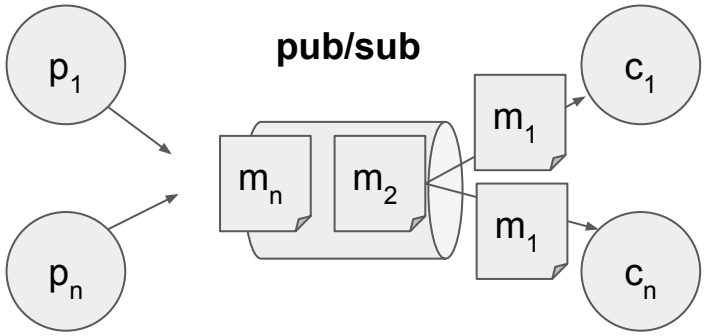
Avantages:

- fiable et persistante
- asynchrone
- flexible

Désavantages:

- complexe
- nécessite un autre éléments système (gestionnaire de messages)
- gestion d'erreur plus difficile





Communication pair-à-pair

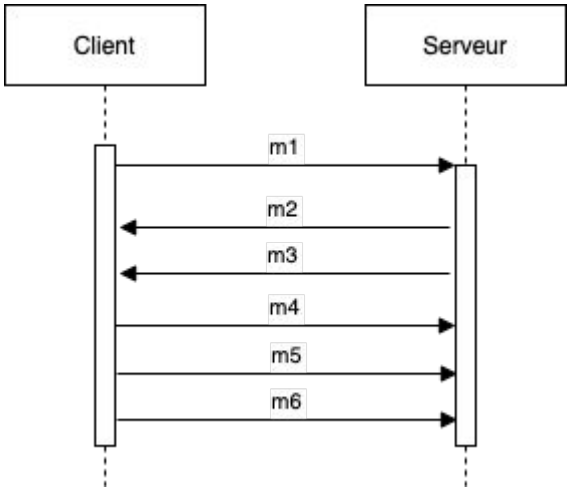
Une connexion est établie entre deux composants et ceux-ci peuvent échanger des messages.

Avantages:

- rapide
- flexible
- bidirectionnel

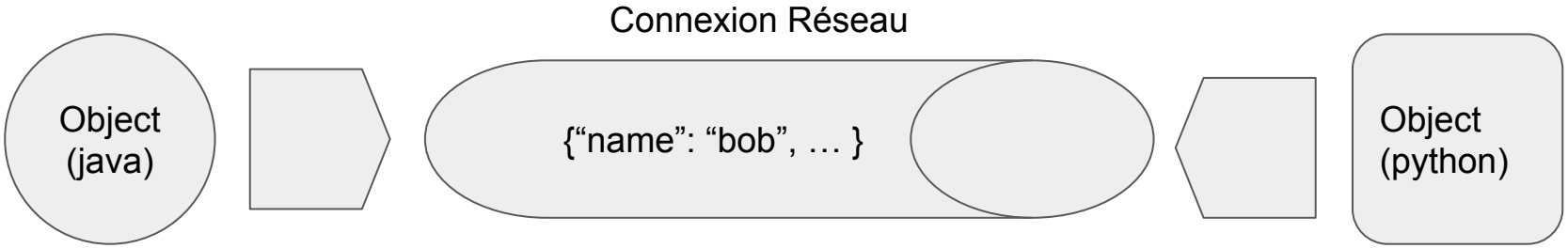
Désavantages:

- demande une connexion permanente
- mise à l’échelle plus difficile
- sémantique des messages est à définir



Sérialisation des données

Lorsque deux composants (possiblement écrits dans des langages différents) échangent des données, ils doivent s'entendre sur la représentation intermédiaire que ces données auront sur le réseau. Ainsi, lorsqu'un message est envoyé, le contenu (*payload*) doit être sérialisé par l'émetteur, puis dé-sérialisé par le receveur. Ceci assure l'interopérabilité entre les différents composants.



Sérialisation des données

Plusieurs choix de représentation sont possibles pour les données. Typiquement les options sont:

- texte - simple, lisible, non standardisé
- XML - simple, standardisé, schéma
- json - simple, standardisé
- encodage binaire (e.g. protocol buffer) - complexe, optimisé

Possibilité d’utiliser aussi un encodage (e.g. compression) pour minimiser le contenu à transmettre.

Tâches asynchrones

Dans certains cas, la couche de traitement doit exécuter des tâches qui peuvent être longues. L'exécution de ces tâches peuvent être déclenchées:

- par une requêtes utilisateur (e.g. génération de rapport)
- par un événement système (e.g. l'arrivée d'un fichier)
- ou selon un horaire précis (e.g. cron job)

Ce traitement ne s'exécute pas dans le thread principal de la requête, mais est ajouté à une liste de tâches à effectuer et est traité lorsqu'un travailleur (*worker thread*) est disponible.

Parfois, la couche de traitement contient un système d'ordonnancement de tâches (job scheduler) qui permet de configurer des flux de travaux (workflow) et de déclencher leur exécution via un API.

Introduction à gRPC

gRPC est un cadriciel de RPC (remote procedure call) haute performance.

Utilisant sur HTTP2 comme transport, gRPC encode les messages transitant entre le client et le serveur dans un format binaire très compact.

Pour utiliser gRPC, on définit d'abord le service, les méthodes et les messages en utilisant un langage de définition d'interface (IDL), *protobuf* dans ce cas.

gRPC fournit ensuite un outil pour générer le bouchon (stub) du client et la structure de base du code serveur (11 langages supportés).

gRPC permet les modes [requête - réponse], [requêtes - flot de réponses], [flot de requêtes - réponse] et bidirectionnel.

https://grpc.io/

Exemple - Service de Profil

Supposons que nous voulons définir un service qui gère le profil des utilisateurs.

Il doit permettre:

- lire un profil
- ajouter un profil
- modifier un profil
- supprimer un profil
- rechercher un profil

Service de Profil - Design

Le service permet simplement d’accéder et de modifier les données de profil des utilisateurs.

On remarque que:

- le client initie l’ensemble des scénarios d’utilisation
- le serveur effectue seulement l’opération demandée

Nous pourrions alors utiliser un patron de communication de type requête-réponse et définir l’API avec du XML.

Service de Profil - Diagramme

Service de Profil - Définition de l’API

HTTP POST, Content-Type: application/xml >>>

```
<ProfileServiceRequest>
 <GetProfile id="1234"></GetProfile>
</ProfileServiceRequest>
```

Réponse, Content-Type: application/xml <<<

```
<ProfileServiceResponse>
 <Profile>
  <Id>1234</Id>
  <Name>John</Name>
  ...
  </Profile>
</ProfileServiceResponse>
```

Exemple - Service de Notifications

Supposons que nous voulons définir un service qui envoie des notifications aux utilisateurs lorsqu'un nouveau message est reçu.

Le service doit permettre:

- au client de s'enregistrer pour recevoir des notifications
- d'écouter sur les nouveaux messages
- d'émettre une notification aux utilisateurs concernés

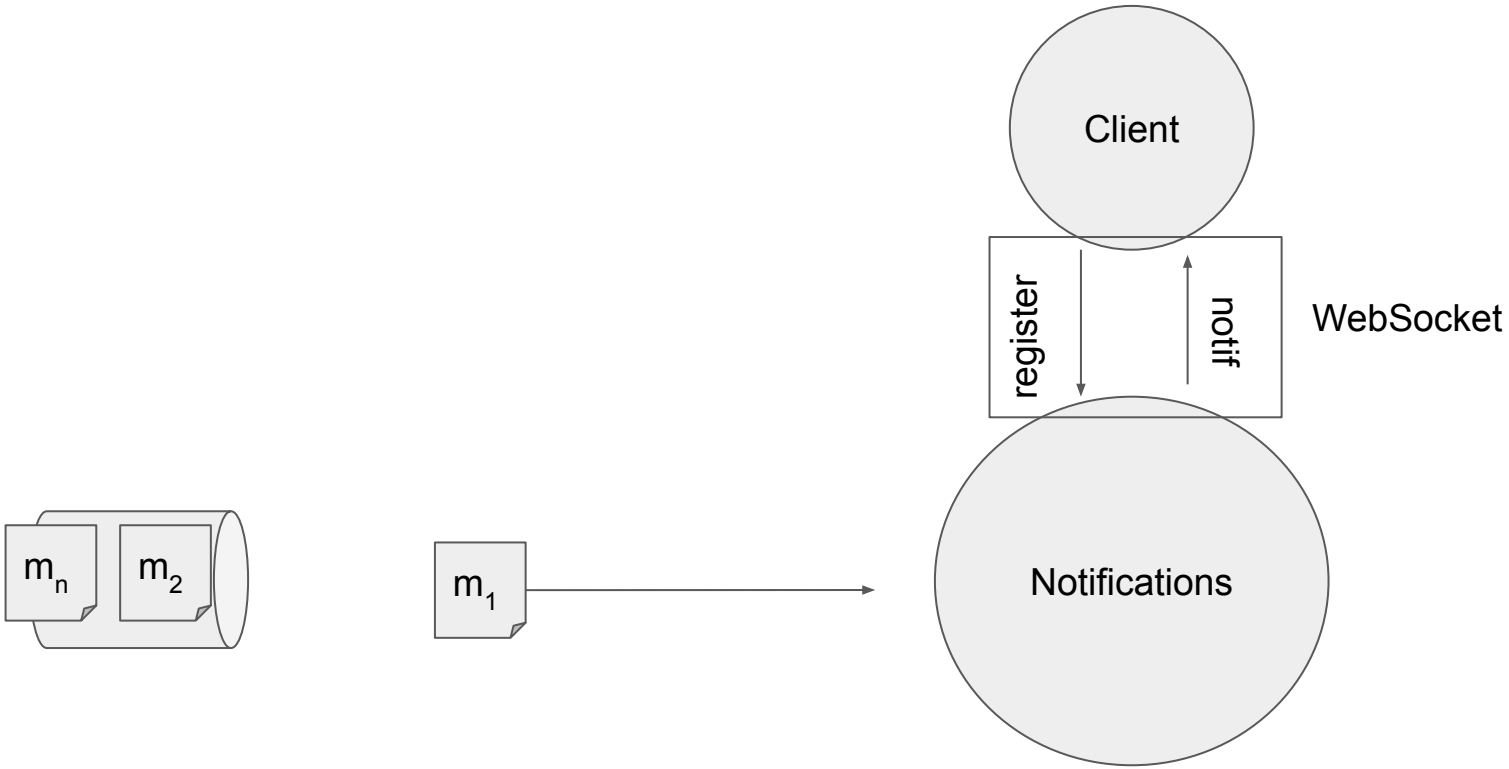
Service de Notifications - Design

On remarque que:

- le client initie son enregistrement
- le serveur doit pouvoir communiquer rapidement au client
- le serveur doit *écouter* pour savoir quand un nouveau message est disponible

Nous pourrions alors utiliser une connexion pair-à-pair entre le client et le service. Les messages échangés pourraient être en json. Le service pourrait écouter sur une file de messages pour réagir aux nouveaux messages disponibles.

Exemple - Service de Notifications - Diagramme



Service de Notifications - Définition de l'API

Enregistrement du client

```
{
  "command": "register",
  "clientId": "12345"
}
```

Notification du serveur

```
{
  "command": "notification",
  "type": "message",
  "message": {
    "id": "12233,
    "timestamp": 12345678,
    "content": "..."
  }
}
```