

# Projet Applications réseau : Microblogage fédéré

## 1 Mode opératoire

Le projet est à réaliser par *binômes* sur les prochaines 5 séances de TP. Les binômes sont constitués de membres d'un même groupe de TP.

**Livrables et organisation** Vos livrables contiendront une archive sous **forme zip** ainsi qu'un fichier **pdf** décrivant votre travail et répondant aux questions éventuelles du sujet. Les dates exactes dépendent de votre groupe

- semaine 1 : début du projet, choix des binômes sur AMETICE **avant le 25/2 :23h59**
- semaine 2&3 : livrable intermédiaire, à déposer sur AMETICE **avant le 9/3 :23h59**
- semaine 4&5 : livrable final, à déposer sur AMETICE **avant le 30/3 :23h59** (avant la soutenance)
- semaine 6 : Soutenance, voir sur ADE pour le planning
  - Luminy : 06/04 de XXh à XXh (sera précisé ultérieurement)
  - Aix : XX/04 de XXh à XXh (sera précisé ultérieurement)

**Présentation du code** Il est recommandé de coder en java, mais le choix du langage est libre (le support sera fourni pour java). Motivez votre choix si vous optez pour un langage autre que java. Il est indispensable que votre projet respecte les meilleures pratiques de génie logiciel tant dans la conception que la réalisation. Il est fortement recommandé d'utiliser *un système de gestion de version* et de *tester votre code*.

**Rendu** Le rendu s'effectuera sur AMETICE. Le rendu intermédiaire doit indiquer clairement les membres du binôme. Vous décrierez notamment dans le document **pdf** l'architecture retenue pour votre serveur. Il ne sera pas possible de changer, intégrer un binôme après le livrable intermédiaire. *L'absence de rendu intermédiaire* est pénalisé et indique en tout état de cause que vous travaillez en monôme.

Notez que vous pouvez rendre plusieurs versions de votre projet tant que vous ne validez pas définitivement.

**Aucun fichier ne sera accepté par courriel**

**Soutenance** La soutenance consiste à une **démonstration** de votre système. Vous avez *12 min* de préparation, *8 minutes de présentation + 4 minutes de questions*. La présentation pourra suivre l'ordre des parties de l'énoncé et des tests associés. Il n'est pas demandé de présenter votre code (celui-ci sera évalué ultérieurement) mais plutôt le fonctionnement de votre application. Pensez à préparer des fichiers de configuration ou des scripts permettant de montrer plusieurs scénarios en *temps limité*. Il est fortement conseillé de répéter votre démonstration avant le jour de la soutenance.

## 2 Protocole microblogamu

L'application est une application de *microblogage* (ou *gazouillage*). Les utilisateurs peuvent poster des courts messages, lire les derniers messages postés, s'abonner à d'autres utilisateurs (auquel cas ils reçoivent les

messages de ces utilisateurs) et s'en désabonner. Les messages peuvent contenir des *mots-clés* (*tags*), et l'on peut consulter les messages récents contenant un mot-clé donné.

Les utilisateurs sont identifiés par un *nom* : suite de caractères alphanumériques sans espace commençant par @ (e.g., @alice, @B42r115). Un mots-clé est une suite de caractères alphanumériques sans espace commençant par # (*hashtag*).

Les messages de l'application ont chacun un *identifiant unique* *id* sur 64 bits. Cet identifiant est attribué par le serveur. Le contenu d'un message est limité à 256 caractères. Le *port par défaut* de **microblogamu** est 12345.

## 2.1 Requêtes/réponses

L'interaction avec le serveur s'effectue soit par des échanges de type requête/réponse, soit via des flux de notifications (voir plus bas). Chaque requête donne lieu à l'ouverture d'une nouvelle connexion, qui est fermée une fois la réponse obtenue. Une *requête* ou une *réponse* se compose d'un *entête* (*header*) et éventuellement d'un *corps* (*body*). L'entête identifie la requête et contient ses paramètres tandis que le corps est formé de son contenu. L'entête et le corps sont terminés par un retour chariot (*carriage return*) suivi de fin de ligne (*linefeed*), e.g. \r\n.

Typiquement, une connexion est établie à l'initiative du client, qui ensuite envoie la requête. La connexion est fermée une fois la réponse correspondante transmise (coté serveur)/reçue (coté client).

### Requêtes

#### *Publier un message*

- entête : PUBLISH author:@user
- corps : contenu du message
- réponse :OK ou ERROR

@user est l'auteur du message. Le serveur renvoie une réponse OK ou ERROR décrite plus bas.

#### *Recevoir des identifiants de messages*

- entête : RCV\_IDS [author:@user] [tag:#tag] [since\_id:id] [limit:n]
- corps : aucun
- réponse : le serveur renvoie une réponse MSG\_IDS contenant les identifiants des *n* messages les plus récents correspondant aux critères de la requête. Les identifiants sont ordonnés par ordre antichronologiques (les plus récents en premier).
  - (optionnel) author:@user l'auteur des messages est @user
  - (optionnel) tag:#tag les messages contiennent le mot clé #tag
  - (optionnel) since\_id:id les messages ont été publiés après le message dont l'identifiant est id
  - (optionnel) limit:n aux plus *n* identifiants sont renvoyés. La valeur par défaut de *n* est *n*=5.

Ainsi, la requête RCV\_IDS déclenchera une réponse MSG\_IDS contenant les identifiants des 5 derniers messages publiés, les identifiants des messages les plus récents en premier. La requête RCV\_IDS author:@bob tag:#amu renverra les identifiants des 5 derniers messages publiés par @bob et contenant le mot-clé #amu.

#### *Recevoir un message*

- entête : RCV\_MSG msg\_id:id
- corps : aucun
- réponse : le serveur renvoie une réponse MSG contenant le message dont l'identifiant est id ou ERROR si il n'existe pas de message dont l'identifiant est id.

#### *Publier un message en réponse à un autre*

- entête : REPLY author:@user reply\_to\_id:id
- corps : contenu du message
- réponse :OK ou ERROR

Le message est publié en réponse au message dont l'identifiant est `id`. Le serveur renvoie une réponse `OK` ou `ERROR`.

#### *Re-publier un message*

- entête : `REPUBLISH author:@user msg_id:id`
- corps : aucun
- réponse : `OK` ou `ERROR`

Le message dont l'identifiant est `id` est publié de nouveau.

**Réponses** Ces réponses sont émises par le serveur en retour d'une requête

#### *Confirmation de publication*

- entête : `OK`
- corps : aucun renvoyée suite à une requête de publication d'un message

#### *Signalement d'une erreur*

- entête : `ERROR`
- corps : message d'erreur

Exemples de messages d'erreur : `Bad request format`, `Unknown message id`, etc .

#### *Liste d'identifiants de messages*

- entête : `MSG_IDS`
- corps : identifiants de messages, un par ligne, ordonnées par les plus récents en premier.

#### *Message*

- entête : `MSG`
- corps : la première ligne contient les méta-données du message, i.e. `author:@user msg_id:id [reply_to_id:id] [republished:true/false]`. Les couples `key:value` sont séparés par un ou plusieurs espaces. Les couples entre crochet [ ] sont optionnels. La deuxième ligne et les suivantes contiennent le contenu du message.

## 2.2 Flux

Le mode requête/réponse requiert l'établissement d'une nouvelle connexion pour la réception de chaque nouveau message. De plus le client à la charge de périodiquement demander au serveur les messages qui l'intéressent. Dans le mode flux, le serveur notifie le client à chaque fois qu'un nouveau message pour lequel le client a manifesté de l'intérêt est publié.

#### *Se connecter*

- entête : `CONNECT user:@user`
- corps : aucun
- réponse : `OK` ou `ERROR`.

Cette requête déclenche l'ouverture d'une connexion avec le serveur sur laquelle sera transmis le flux de messages auquel s'abonne le client. La gestion des abonnements se fait au moyen des requêtes `SUBSCRIBE/UNSUBSCRIBE`.

#### *S'abonner*

- entête : `SUBSCRIBE author:@user` et `SUBSCRIBE tag:#tag`
- corps : aucun
- réponse : `OK` ou `ERROR`. Le serveur envoie `ERROR` si `@user` n'est pas géré par l'application. Si par contre le mot-clé `#tag` n'est pas encore géré, il est ajouté et `OK` est renvoyé.

Suite à cette requête, le serveur enverra une réponse `MSG` sur la connexion précédemment ouverte par `CONNECT` à chaque fois qu'un message dont l'auteur est `@user` ou qui contient le mot-clé `#tag` est publié.

#### *Se désabonner*

- entête : `UNSUBSCRIBE author:@user` et `UNSUBSCRIBE tag:#tag`
- corps : aucun

- réponse : OK ou **ERROR**. Le serveur renvoie **ERROR** si le client n'est pas abonné à **@user** ou au mot-clé **#tag**.

Suite à cette requête, le serveur cesse d'envoyer des messages dont l'auteur est **@user** ou qui contiennent le mot-clé **#tag**.

## 3 Service centralisé

Nous commençons par un service centralisé fourni par un seul serveur.

### 3.1 Serveur rudimentaire

1. Écrire un serveur rudimentaire qui répond aux requêtes **PUBLISH**. Les messages reçus via les requêtes **PUBLISH** seront affichés avec leur identifiant sur la sortie standard.
2. Écrire un premier client **publisher** qui :
  - Demande un pseudo (**@user**) à l'utilisateur
  - Attend les messages de l'utilisateur (sur l'entrée standard)
  - Transmet les messages de l'utilisateur au serveur

Valider votre client et votre serveur avec **netcat**.

### 3.2 Gestion des requêtes/réponses

3. Compléter votre serveur pour répondre aux requêtes **RCV\_IDS** et **RCV**
4. Écrire un deuxième client **follower** qui
  - Récupère les identifiants des messages d'un ou plusieurs utilisateurs (spécifiés au lancement du client)
  - Affiche le contenu des messages correspondant à ces identifiants

Vous avez le choix de l'architecture du serveur : utilisation de **select**, d'un pool de *threads* ou combinaison des deux. On fera attention à l'unicité des identifiants des messages, ainsi qu'à l'ordre dans lequel les messages sont affichés.

5. Finaliser votre serveur en ajoutant la gestion des requêtes **REPLY** et **REUBLISH**.
6. Écrire un client **repost** qui re-publie chaque message émis par un ensemble d'utilisateurs déterminé au lancement du client.
7. Valider votre serveur avec plusieurs clients.

### 3.3 Gestion des flux

Nous allons maintenant écrire le serveur **microblogamu\_central**, qui en plus en plus du mode requête/réponse supportera la gestion des flux d'abonnement.

8. Proposer une architecture fondée sur des files d'attente associées à chaque client abonné à des mots-clés et/ou utilisateurs.

On pourra utiliser ces files en mode producteur/consommateur :

- Les producteurs placent dans les files concernées les nouveaux messages publiés.
- Il y a un seul consommateur par file, qui renvoie dans la socket du client les messages présents dans la file d'attente associée à ce client.

On pourra utiliser les implémentations `ArrayBlockingQueue<T>` ou `ConcurrentLinkedQueue<T>`. Voir aussi les implémentations de tables de hachage `java.util.concurrent.ConcurrentHashMap<K,V>` ou `java.util.concurrent.ConcurrentSkipListMap<K,V>` pour la gestion des abonnements.

9. Comment s'assurer que la communication est asynchrone ?
10. Proposer une validation fonctionnelle basée sur des tests automatisés avec plusieurs clients.
11. Écrire le serveur `microblogamu_central`.
12. Écrire un client complet `microblogamu`. Il lira sur l'entrée standard les commandes de l'utilisateur (PUBLISH, REPLY, REPUBLISH, (UN)SUBSCRIBE) avec leurs paramètres. Il affichera les messages reçus précédés de leur auteur et de leur identifiant.

## 4 Fédération de serveurs

Afin de mieux supporter la charge, on décide de *fédérer* les serveurs, c'est-à-dire de coordonner plusieurs serveurs sur lesquels les clients peuvent se connecter indifféremment.

13. Vis-à-vis des autres serveurs, un serveur peut-il se comporter comme un client classique ?  
Ajouter une opération `SERVERCONNECT`.

### 4.1 Fédération simple

14. Faire une fédération simple à deux serveurs, puis tester avec deux puis trois clients.
15. Que constatez-vous en terme d'ordre d'affichage des messages par des clients connectés à des serveurs différents de la fédération ?
16. Ajouter le principe de *serveur maître* : l'un des serveurs détermine l'ordre d'affichage des messages. Pour cela, tous les messages lui sont d'abord remontés. Le serveur maître utilise une file `master` dont
  - les producteurs sont les autres serveurs et les clients directement connectés
  - les consommateurs sont les autres serveurs et lui-même
17. connecter les serveurs entre eux : on écrira un fichier de configuration donnant les caractéristiques des autres serveurs sous la forme d'un fichier `pairs.cfg` :

```
master = adresseIP0 port0
peer = adresseIP1 port1
peer = adresseIP2 port2
...
peer = adresseIP3 port3
```

Si la ligne `master =` n'apparaît pas, alors le serveur est *maître*.

### 4.2 Montée en charge

18. Connecter deux serveurs entre eux. Puis trois, puis 10.
19. Proposer une validation fonctionnelle basée sur des tests avec plusieurs clients et au moins trois serveurs.

### 4.3 Fédération décentralisée

La solution précédente, avec un serveur maître, présente un point de défaillance individuel. En effet, si le serveur maître tombe en panne, l'ensemble de la fédération cesse de fonctionner.

20. Pour vérifier cela, relancer votre système en modifiant votre serveur maître afin de simuler des défaillances : toutes les 15s, le serveur maître se met en pause pour 5s puis s'arrête définitivement au bout de 3 minutes. Que constatez-vous ?

Nous cherchons donc une solution fédérée véritablement décentralisée. Pour cela, nous n'allons plus utiliser de serveur maître, mais un *algorithme distribué* identique sur tous les pairs de la fédération.

**Algorithme distribué tolérant aux défaillances** Nous nous plaçons dans la situation où tous les pairs communiquent directement les uns avec les autres mais où le temps d’envoi de certains messages est imprévisible (asynchronicité). Par conséquent, il faudra mettre en place des mécanismes supplémentaires pour, éventuellement, distinguer les messages lents des messages qui pourraient ne plus arriver.

On supposera ici que le nombre de défaillances possibles est strictement inférieure à la moitié du nombre total de pairs.

21. Quel type d’ordre faut-il sur les messages pour qu’à l’affichage des messages on soit certain qu’une réponse est toujours affichée après la question correspondante ?

**Implémentation** Implémenter l’algorithme décrit ici pour 3 serveurs fédérés.

**Simulation des défaillances** Reprendre le code des serveurs en ajoutant des pauses de temps aléatoires pour chaque thread.

## 5 (Optionnel) Améliorations

Mettre en œuvre une ou plusieurs des propositions ci-dessous :

- *Persistence* Inclure le stockage sur disque des publications et des abonnements utilisateurs, par exemple via une base de données ou directement dans des fichiers.
- *GUI* Modifier le code pour utiliser une fenêtre graphique avec
  - Une fenêtre principale pour afficher le flot des publications
  - Une fenêtre pour entrer les publications
  - Un menu pour la gestion des abonnements
- *Sécurité et contrôle d’accès* Quels sont les manques en terme de sécurité de votre système ? Proposer une amélioration permettant de garantir qu’une personne ne puisse usurper le pseudo d’un autre utilisateur. Proposer des améliorations pour contrôler les abonnements aux publications des utilisateurs.
- *Analyse de performance* Évaluer les performances de votre système selon différents scénarios de défaillances de votre choix ainsi que différentes tailles du système.
- *Statistiques* Ajouter de nouvelles requêtes grâce auxquelles on pourrait obtenir :
  - Les utilisateurs/mots-clés populaires
  - Les messages les plus re-publiés
  - Les mots-clés (ou plus généralement mots) qui sont actuellement mentionnés avec les plus grandes fréquences (*trending topics*)
  - Les utilisateurs les plus centraux. Voir la notion de *centralité* dans un graphe ici
  - etc.