TD Application Répartie

P.Morat

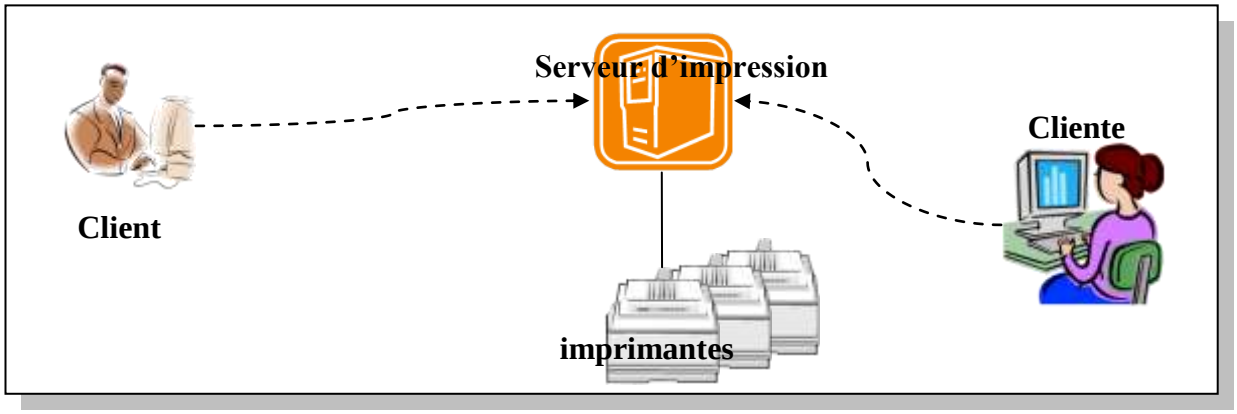
Préambule

Ce td a pour objectif de mettre en œuvre les outils de programmation du réseau en utilisant la librairie java.net et plus particulièrement les modes TCP et UDP.

1. Serveur d'impression

1.1 Présentation de l'application

On veut réaliser un simulateur de serveur d'impression distant permettant à des clients d'envoyer des requêtes d'impression à un serveur qui prendra en charge ces opérations.



L'application sera contenue dans un package de nom « jus.aor.printing ».

Dans un premier temps nous ne nous intéresserons qu'à la relation entre le Client et le Serveur d'impression. Le client voulant imprimer un fichier (nous ne considérerons que ce cas) se situant sur son espace local, enverra une requête d'impression auprès du serveur et lui fournira les informations nécessaires à l'aboutissement de celle-ci.

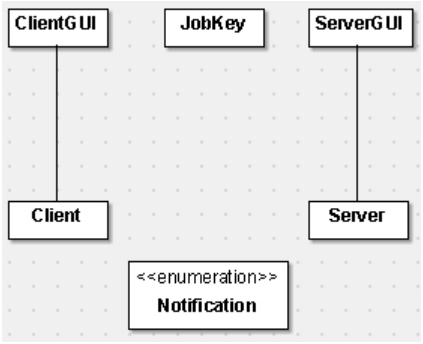
Dans le souci de pouvoir compléter, au fur et à mesure des objectifs, ce système d'impression par d'autres opérations, nous proposons de caractériser les requêtes du client et les réponses du serveur de façon claire en utilisant le type suivant :

```
public enum Notification implements Serializable {
	QUERY_PRINT;
}
```

Pour plus de clarté, les notifications du client vers le serveur seront préfixées par QUERY et par REPLY dans l'autre sens.

La forme générale d'une interaction entre le client et le serveur sera :

<NOTIFICATION>{<PARAMETRES>}{<DONNEES>} où <NOTIFICATION désigne la requête ou la réponse, <PARAMETRES> d'éventuelles informations complémentaires à la notification mais de taille réduite et <DONNEES> les éventuelles informations dont la taille conséquente n'est pas prédéterminée.



Afin de séparer clairement selon le modèle MVC la partie modèle du reste nous proposons des classes (*GUI) qui fournissent les éléments d'interaction avec l'utilisateur. Vous construirez les classes associées pour implanter les opérations à réaliser. Ces classes d'interfaçage avec les

utilisateurs sont relativement rudimentaires car elles ne mettent pas en œuvre le modèle Bean (Event et Listener). Vous pourrez les améliorer à votre convenance.

L'impression d'un fichier local sur une imprimante distante nécessite que la communication soit fiable, en effet ce qui sera imprimé devra être exactement identique à la source. Pour cela on utilisera le protocole TCP qui garantit cette fiabilité dans la communication entre le client et le serveur.

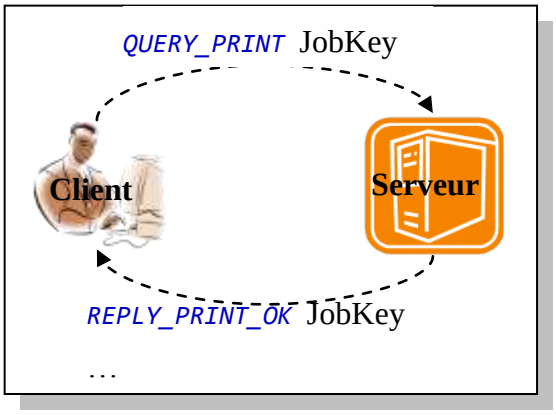
De façon à identifier les jobs (travaux d'impression) on définit une classe JobKey[1] qui identifie de manière unique un Job[2]. Pour obtenir un nom unique pour une JVM (de la forme PID@UNAME) utilisez la méthode `ManagementFactory.getRuntimeMXBean().getName()`.

Dans la suite, pour des raisons pédagogiques visant à explorer différentes possibilités nous allons mettre en œuvre plusieurs stratégies de réalisation. Afin de facilité l'évolutivité de l'application, nous définirons des méthodes pour chacune des sous-transactions possibles entre client et serveur, elles seront élaborées dans les classes TCP et UDP sous forme strictement fonctionnelle et donc static.

On met à votre disposition un mécanisme de LOGGER permettant de faire l'enregistrement de traces. Pour cela vous disposez d'une classe WriterAreaHandler ayant en charge l'affichage, une classe Level fournissant des niveaux de log, 3 logs identifiés par les noms «`Jus.Aor.Printing.Client`», «`Jus.Aor.Printing.Server`» et «`Jus.Aor.Printing.Spooler`», enfin vous compléterez 3 bundles permettant l'externalisation des messages. Vous utiliserez la méthode log du Logger concerné pour enregistrer une nouvelle trace.

1.2 Objectif n°1

Dans un premier temps, vous allez réaliser une communication réduite entre le client et le serveur d'impression qui se résume de la part du client à envoyer une notification d'impression au serveur qui lui répond par un acquittement de cette demande.



La requête est : *QUERY_PRINT* JobKey où JokKey est un paramètre de la requête. En retour le client attend la réponse : *REPLY_PRINT_OK* JobKey ou une autre notification si une erreur est survenue. Dans le cas correct on pourra vérifier que le JobKey retourné, par quelque moyen que se soit, est conforme à celui envoyé.

Dans cette première version on utilisera des **Data<X>Stream** pour réaliser les communications. Tout protocole débutera par l'identification de la requête (notification) suivie des informations spécifiques à cette requête. Dans cette première version, l'information transmise pour la notification sera l'ordinal (int) de la valeur de celui-ci. Par contre pour simplifier la transmission d'un JobKey, on utilisera une opération de marshalling comme spécifiée par la méthode marshal de la classe JobKey à laquelle on associe un constructeur de cette même classe. **Dans un premier temps ignorez le paramètre « n » de la méthode queryPrint de la classe Client**. *En première étape à cette réalisation vous pouvez restreindre le protocole aux notifications sans paramètre.*

- Pour vérifier le bon fonctionnement de votre programme, tester le interactivement en déployant le client et le serveur sur la même machine. Vous pourrez aussi le tester en mettant en œuvre plusieurs machines (virtuelles ou physiques). Attention, si vous

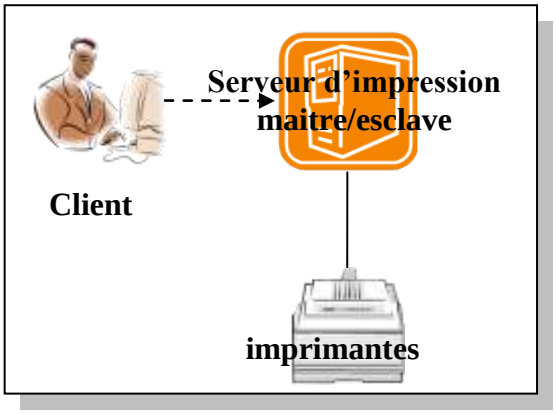
[1] Voir code fourni
[2] Voir code fourni

souhaitez faire collaborer votre réalisation (client ou serveur) avec celle d'un collègue, vous devrez vous assurer de la conformité de vos protocoles respectifs.

- Mettez en place un mécanisme permettant de simuler l'envoi quasi-simultané de plusieurs requêtes d'impression auprès du serveur (**prise en compte du paramètre n dans la méthode queryPrint du Client**). Etudiez plusieurs scénarios de configuration de celui-ci et regardez si cela a un impact sur le comportement. Entre-autre vous pourrez influer sur la temporisation du serveur qui permet de ne par rejeter une demande de connexion trop précipitamment.

1.3 Objectif n°2

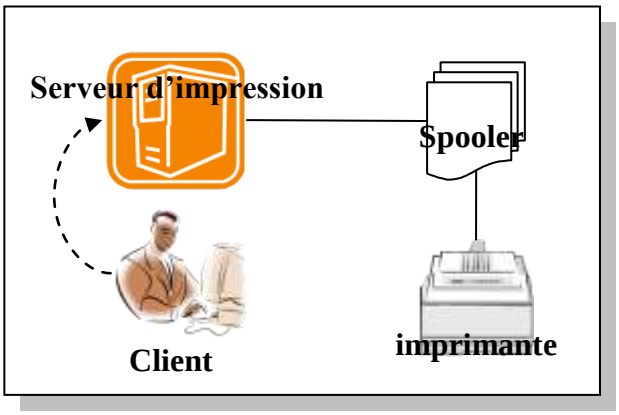
On souhaite maintenant compléter ce qui a été réalisé dans l'objectif précédent **en assurant la transmission de l'information devant être, in fine, imprimée**. Pour garantir une meilleure disponibilité du service d'impression, on décide de mettre en place un serveur selon le principe **maitre/esclave**. Le maitre assurera la prise en compte de la requête d'impression, l'esclave quant à lui sera en charge d'assurer la communication effective avec le client. Dans cette seconde version on se contentera d'écrire sur un support d'affichage le contenu du fichier devant être imprimé afin de s'assurer de la correction de l'opération. Le Client est libéré du moment où la transmission du document est achevée. Vous fixerez le protocole nécessaire pour que cette opération soit correctement réalisée et que le client soit pertinemment renseigné. **Le client ne conservera pas en interne des données de plus de 1024 octets (`MAX_LEN_BUFFER`)**. On ne se souciera pas de problèmes annexes et de gestion de timeout.



- Pour vérifier le bon fonctionnement de votre programme, tester-le interactivement.
- Adaptez le test de simultanéité précédent à cet objectif.

1.4 Objectif n°3

Pour simuler encore plus précisément cette activité, on décide de prendre en compte le temps nécessaire à l'impression du document par le spooler d'impression. On se placera dans le cas simple ou le serveur d'impression ne dispose que d'une seule imprimante. Le document à imprimer sera dans un premier temps « spoolé » (stocké sous forme d'un Job) dans le Spooler qui aura pour mission d'imprimer les documents en attente les uns après les autres. On considérera que le temps nécessaire à l'impression effective d'un document n'est fonction que de sa taille. Réalisez la classe Spooler qui assure la gestion de la file d'attente et la simulation de l'impression sur l'imprimante. On vous propose comme précédemment une classe SpoolerGUI assurant l'interfaçage avec l'utilisateur. Modifier le serveur en conséquence pour prendre en compte cet élément de simulation, désormais l'affichage du document pourra être fait par le spooler au moment de la simulation de son impression par l'imprimante.



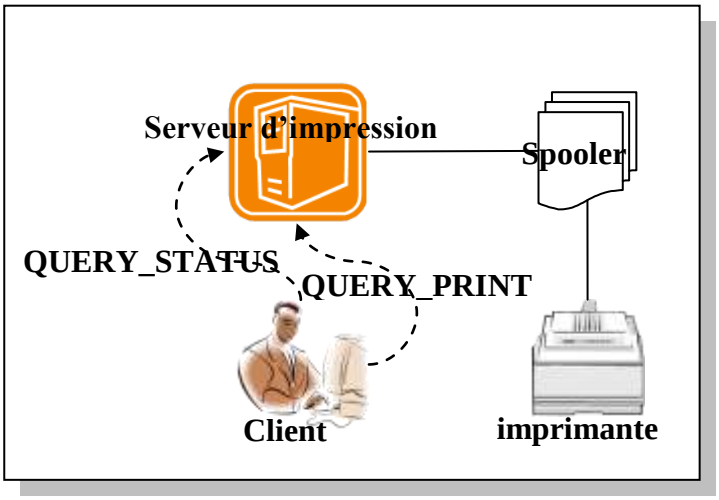
- Pour vérifier le bon fonctionnement de votre programme, tester le interactivement.

1.5 Objectif n°4

On souhaite mettre en place, dans le serveur, un pool préétabli d'esclaves pour répondre plus promptement à une demande d'impression. Le nombre d'esclaves préconstruit sera un paramètre du serveur. Lorsqu'une demande d'impression arrive, si un esclave du pool est libre on lui confie la mission. Dans le cas où plus aucun esclave du pool n'est libre alors on procède comme dans l'objectif précédent.

- Mettre en place la structure nécessaire pour ce faire.
- Objectif secondaire : Mettez en place un scénario permettant une comparaison des performances d'une solution maitre/esclave simple et d'une solution avec un pool.

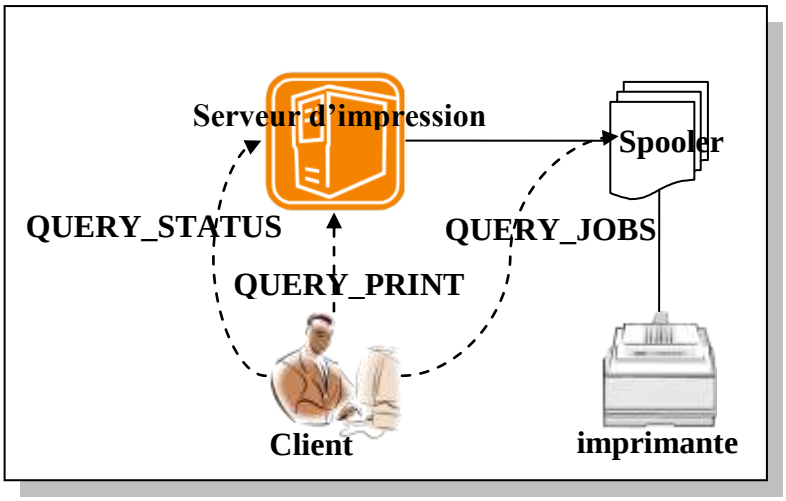
1.6 Objectif n°5



On souhaite ajouter la possibilité d'interroger le serveur d'impression sur le nombre et le délai des impressions en attente sur le spooler. Pour cela on ajoutera une nouvelle requête à laquelle le serveur fournira une réponse contenant ces 2 informations sous la forme d'un entier pour le nombre de jobs en attente et d'un long pour ce qui concerne le temps estimé de traitement de cet ensemble de jobs. Dans ce cas la fiabilité de la communication est moins cruciale, on décide donc d'utiliser le protocole UDP pour réaliser celle-ci. Le traitement de cette requête étant rapide, nous la ferons réaliser directement par le serveur maitre. On utilisera donc la même connexion que pour la requête d'impression, les 2 modes pouvant cohabités sur le même port. En retour un ServerStatus sera transmis en utilisant le même procédé que celui utilisé pour un JobKey hormis que l'on transmettra les représentations hexadécimales des attributs séparées par le caractère '¤'.

- Modifier tous les composants logiciels impactés par cette nouvelle requête.
- Objectif secondaire : Introduisez la possibilité de simuler la perte de paquet UDP. Modifier la spécification et la réalisation de votre client en conséquence.

1.7 Objectif n°6



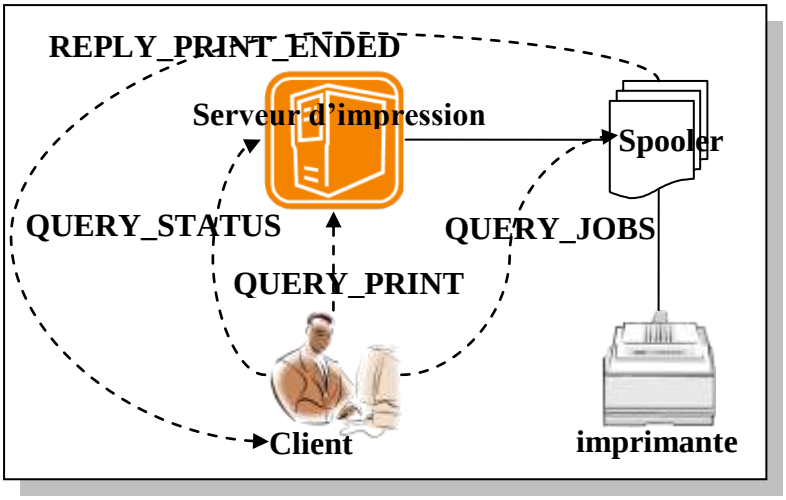
On souhaite ajouter une requête supplémentaire permettant à un client de connaître l'état d'avancement d'une(de) requête(s) d'impression qu'il a soumis au serveur. Pour cela on utilisera JobKey avec la possibilité de créer des identifications ensemblistes, au minimum « tous mes Jobs ».

Dans ce cas, nous allons considérer que cette requête se fait sur un port secondaire (par exemple le numéro du port principal+1) auquel sera attaché un service du spooler

d'impression. Là encore la fiabilité de la communication n'est pas une nécessité absolue, cependant nous utiliserons le protocole TCP. Pour la communication dans le sens du Client vers le Spooler on transmet la clé (JobKey) du(es) job(s) incriminés de la même façon que précédemment et dans l'autre sens le Spooler renverra un objet JobState contenant les informations caractéristiques demandées. Vous utiliserez les flux ObjectXStream pour transmettre cet objet JobState construit.

1.8 Objectif n°7

Pour permettre au client d'être informé de l'avancement du spooler, celui-ci émettra en mode UDP multicast à chaque fin d'impression une requête `REPLY_PRINT_ENDED` JobKey indiquant à destination d'un port où pourrons se connecter les clients souhaitant être informés de son évolution.



La diffusion multicast est possible uniquement en mode UDP. Ceci se fait au travers de la classe MulticastSocket permettant à un récepteur de rejoindre le groupe grâce à la méthode joinGroup que celle-ci fournit, il ne faut pas oublier de quitter (leaveGroup) le groupe quand on ne souhaite plus en faire partie. Dans le cas d'une connexion multicast l'adresse IP doit être de classe D, c'est à dire supérieure à 224.0.0.0 et inférieure à 240.0.0.0. Le port utilisé pour cette communication ne doit pas être déjà utilisé par une autre connexion UDP, pour cela utilisez le port secondaire (par exemple le numéro du port principal+1).

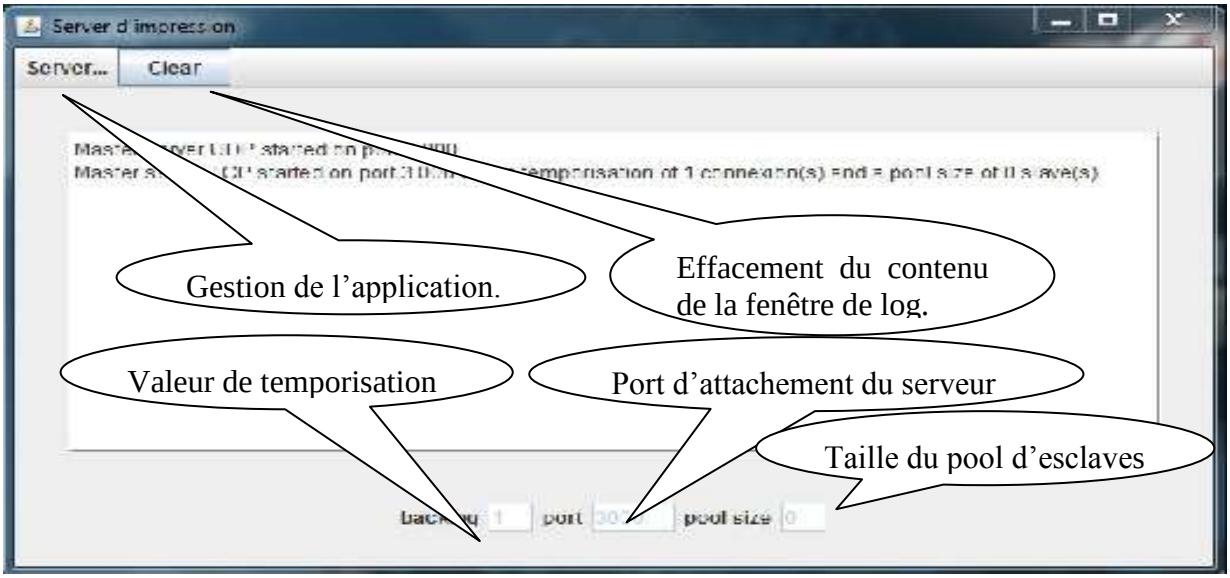
1.9 Objectif n°8

Objectif secondaire : *Reprendre le programme pour unifier la manière de transmettre les informations en utilisant systématiquement la sérialisation qui offre l'avantage d'une plus grande fiabilité sans pour autant pénaliser trop fortement le coût.*

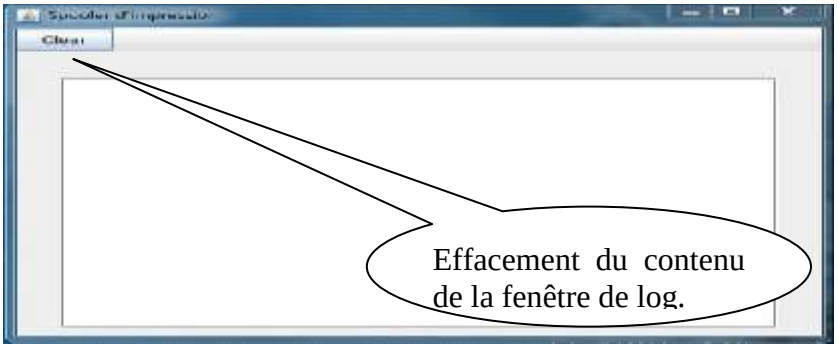
2 Annexes

2.1 Prototypes des interfaces utilisateur

2.1.1 Le serveur



2.1.2 Le spooler



2.1.3 Le client