Modèles d'interaction

Introduction : intergiciel
Mémoire virtuelle répartie
Interaction par messages
- Modèle élémentaire
  - *Interactions asynchrones*
  - *Intergiciels à messages (MOM)*
  - *Schéma publier/s'abonner (communication par événements)*
  - *Exemple : JMS*
  - *Extension : communication de groupe*
- Interactions synchrones : schéma client-serveur
  - *Présentation*
  - *Mise en œuvre : services de base, architecture*
  - *Extensions : intégration au langage, organisations réparties*
- Etude de cas : supervision d'un réseau
- Synthèse : mise en œuvre de l'interaction C/S avec l'API sockets de Java

***Crédits** : ce cours a été construit à partir des sources suivantes :*
- G Coulouris et coll : Distributed Systems, Addison-Wesley
- S Krakowiak : Introduction aux applications réparties, École d'été INRIA "Construction d'Applications Réparties", en ligne
- R. Balter : modes de structuration d'applications réparties, Ecole INRIA,
Certains dessins et schémas de ce cours sont directement issus du cours de S. Krakowiak

1 – Introduction : intergiciel

But : faciliter l'écriture d'applications réparties

Point de vue abstrait (le plus simple pour le programmeur)

Application répartie = application concurrente
= ensemble d'activités coordonnées

⇒ services de :

- gestion des interactions/échange d'informations
- synchronisation
- gestion de l'hétérogénéité
- désignation (localisation)
- gestion des pannes
- gestion des ressources (sécurité, placement, équilibrage, capacité de croissance)

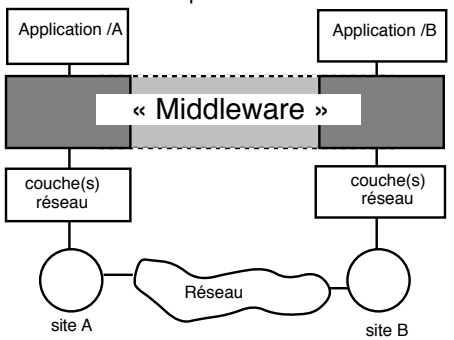
Alléger la tâche du programmeur

→ réaliser (automatiquement) des services induits par la gestion de l'exécution répartie
→ offrir un environnement d'exécution aussi proche que possible
d'un environnement centralisé, sans (les difficultés de la) répartition.

Schéma utilisé :

interception des interactions entre une activité et son environnement, afin de réaliser les services liés à la répartition de manière **transparente**, sans avoir à modifier l'activité.

⇒ **intergiciel** (middleware), couche logicielle
- de médiation entre activités à coordonner,
- simulant un environnement centralisé à partir d'un environnement réparti.



Mise en œuvre de l'environnement centralisé virtuel

- Activités
  → processus exécutés sur un site
- Interactions
  Selon le modèle de programmation, les interactions peuvent
  - être plus ou moins abstraites
  - correspondre plus ou moins directement à la réalité physique

  → être plus ou moins faciles à implémenter

  → compromis à trouver entre facilité d'écriture et efficacité de l'implémentation

Deux modèles de programmation principaux

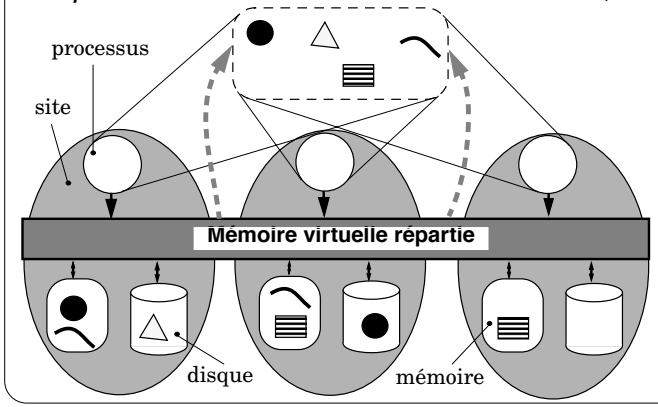
- **mémoire virtuelle répartie** (interaction implicite : modèle de la mémoire partagée)
  ◊ Linda → JavaSpaces, JSDT
  ◊ modèle simple pour le programmeur, mais sans correspondance directe avec un environnement réparti (pas de mémoire partagée en réparti)
- **activités communiquantes** (interaction explicite) :
  ◊ messages (MOM, JMS) → C/S → RMI (CORBA)
  ◊ correspondance directe entre interactions et messages
  ◊ prise en charge plus ou moins poussée des services liés à la répartition, selon l'intergiciel

2 – Mémoire virtuelle répartie

Motivations

- (re) placer le développeur d'applications dans les conditions d'un environnement centralisé :
  → communication et synchronisation par variables/objets partagés
- avantages attendus pour le programmeur :
  simplicité (interactions implicites), efficacité (algorithmes de gestion généraux et transparents)

Principe de réalisation :* mémoire virtuelle *: indirection/adresses « interprétées »



Mise en œuvre efficace de la mémoire virtuelle partagée répartie

Points critiques

- gestion de l'espace libre (glanage de cellules)
- localisation/liaison des données
- stratégies de répartition/(dé)placement des données :
  ◊ recours au principe de localité
  ◊ statique ou dynamique
  ◊ partition ou duplication → gestion des conflits/de la cohérence

Modèles

- Espace de tuples (base de données partagée d'enregistrements (tuples))
  ◊ Opérations sur la base : déposer, retirer, consulter un tuple (correspondant à un motif)
  ◊ Exemples : Linda, JavaSpaces, JSDT (Sun/Oracle)
- Objets (persistants) répartis partagés : Guide, SOR, WebObjects, Globe, Legion…

3 – Interaction par messages

1) Modèle élémentaire

Primitives

Emettre(message, destination)

Recevoir(message, source)

Modalités de base

Synchronisation : schéma producteur/consommateur

- communication synchrone : émission et réception bloquantes (Rendez-vous)
  → suppose un médium de communication fiable
- communication asynchrone :
  ◊ émission non bloquante
  ◊ réception bloquante (attente de message)
  → conditions habituelles des environnements répartis
  → complique la tâche du concepteur d'application
- mode intermédiaire : schéma producteurs-consommateurs (tampon borné) : pipes UNIX

Remarque : il existe généralement une version non bloquante des opérations bloquantes :
un blocage est alors remplacé par le retour d'un diagnostic

Désignation

- communication « point à point » entre activités (canaux)
- communication indirecte : passage par une « boite aux lettres »
  ◊ attachée à un processus (n → 1 : port, porte)
  ◊ partagée par tous les processus (n → m : file de messages)
- statique (ex : IPv4) ou dynamique (ex : redirections UNIX)
- déterminisme en réception ou non (ex : select Ada/UNIX)

Propriétés du service de communication

- Fiabilité (sites)
- Intégrité (messages)
- Qualité de service (QoS) : débit, délai, gigue
- Ordonnancement relatif des émissions et réceptions : local, causal, total

Mise en œuvre

Micro-noyaux (Chorus, Mach)

mécanisme fondamental de l'environnement

Systèmes conventionnels (Unix…)

API (bibliothèque) spécifique : sockets, files de messages SystemV…

Environnement réparti, calcul parallèle

MPI, PVM

Environnement réparti, intégration/répartition d'applications « classiques »

Middleware à messages (MOM) : JMS (Java Message Service), IBM MQSeries

2)Interactions asynchrones : intergiciels à messages

(MOM, Message Oriented Middleware)

- Intergiciel (middleware, surcouche logicielle aux systèmes hôtes)
- Interaction asynchrone, par messages ou événements

2.1) Echange de messages

Principes

- Communication asynchrone, par échange de messages
- Désignation : files de messages partagées

  → communication *anonyme*

  le processus récepteur est déterminé par le *contenu* du message, pas par un identifiant
- Services
  - ◊ Filtrage des messages : types, propriétés
  - ◊ Tolérance aux pannes :
    atomicité des échanges, files de messages rémanentes, accusés de réception
  - ◊ Sécurité : cryptage, authentification, contrôle d'accès

Structure des messages

- En-tête
  identifiant (unique) du message, informations de gestion et d'acheminement
- Paramètres
  couples <nom,valeur> utilisables par le système ou l'application pour filtrer les messages
- Données
  définies par l'application

Opérations de base

- (dé)lier un processus et une file de messages
- déposer/retirer (avec délai de garde) un message dans une file
- confirmer (transaction) la réception d'un message

2.2) Schéma publier/s'abonner (communication par événements)

Principe

≈ signaux UNIX :

- possibilité de définir
  - ◊ des **événements**
  - ◊ des **réactions** (traitements déclenchés par l'occurrence d'événements)
- possibilité de lier dynamiquement événements et réactions

  → forme de **communication anonyme** :

  a priori, émetteurs et récepteurs d'un événement ne se connaissent et ne se désignent pas.

Schéma d'interaction publier/s'abonner (Push, publish/subscribe) : prise en compte (réception) des événements à l'initiative de l'émetteur

- Un événement « publié » (émis) par un processus est diffusé auprès de tous les processus qui se sont « abonnés » à cet événement en lui associant une réaction.
- La réception de l'événement diffusé, déclenche l'exécution de la réaction de chaque abonné.

Remarque : possibilité d'un schéma d'interaction alternatif : scrutation (pull) : prise en compte (réception) des événements à l'initiative du récepteur

- Les événements émis sont conservés, et le récepteur doit consulter périodiquement la file des événements émis, pour déclencher éventuellement la réaction appropriée.

  → schéma analogue à la communication par émission/réception de messages asynchrones

Interface

- abonnement (sujet1, sujet2, ...)
- désabonnement (sujet1, sujet2, ...)

- associer (sujet, réaction [paramètres])
- dissocier (sujet1)

- déclaration de types d'événements : définition des champs (sujet, attributs, ...)

- créer (événement [paramètres])
- publier (événement)

Principes de mise en œuvre

- un serveur enregistre les abonnements et transmet les événements émis
- le service peut être
  - ◊ centralisé (« Hub »)
  - ◊ réparti
    - « Snowflake »: serveurs interconnectés. Chaque serveur gère une fraction des abonnés.
    - Utilisation d'un bus logiciel. Diffusion des publications, filtrage pour les abonnés locaux.

2.3) Exemples

Exemple de système à événements : TIB/Rendez-vous

Source : Tanenbaum & van Steen, Distributed Systems, Prentice Hall

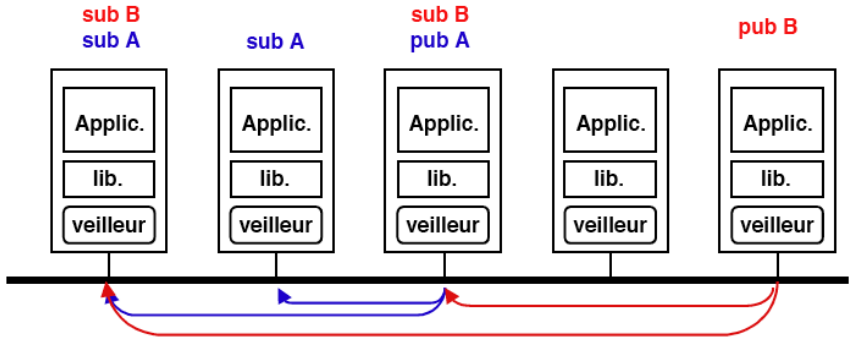
Schéma d'interaction

Publier/s'abonner avec sujet

Réalisation

- Connexion des sites par réseau à diffusion (point à point possible)
- Sur un réseau local :
  chaque site a un veilleur (démon) qui sert d'interface pour l'échange des messages
- Sur une interconnexion de réseaux locaux :
  chaque réseau a un routeur qui lui permet de communiquer avec les autres réseaux.
  Le veilleur implanté sur le routeur sert d'interface de communication.
  On construit ainsi un réseau virtuel (overlay network) au niveau de l'application

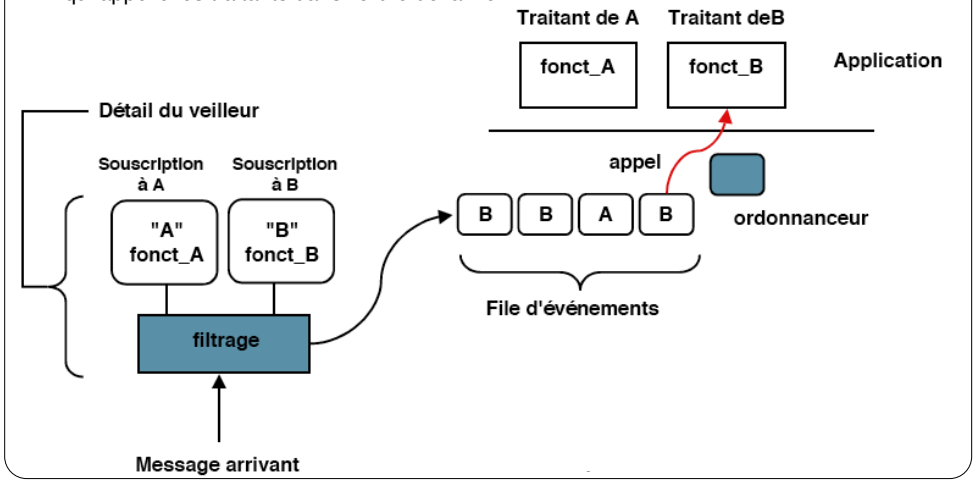
Réseau local



- Un veilleur sur un site S gère une table donnant la liste des sujets auxquels a souscrit l'application sur S
- Un site qui publie sur un sujet X passe le message à son veilleur qui le diffuse à tous les autres veilleurs
- Chaque veilleur filtre les messages arrivants en fonction des sujets souscrits localement

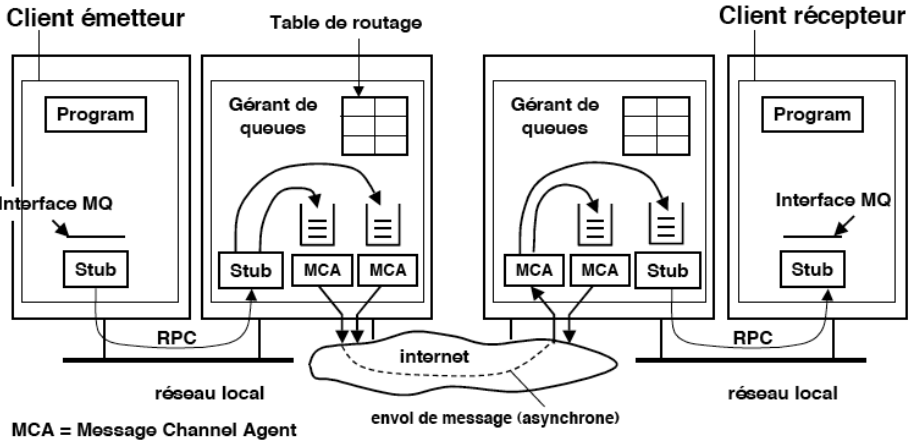
Traitement des réceptions

- Chaque réception de message correspondant à la souscription à un sujet S déclenche l'appel (asynchrone) d'une fonction de l'application (callback) chargée de traiter S.
- L'asynchronisme est traité au moyen d'une file d'événements et d'un ordonnanceur, qui appelle les traitants dans l'ordre de la file.



Exemple de bus à messages : IBM MQSeries

Source : Tanenbaum & van Steen, Distributed Systems, Prentice Hall



Remarque

Pour des raisons de place, on n'a représenté que des files en émission sur l'émetteur, des files en réception sur le récepteur. En fait les 2 sortes de files sont sur tous les gérants.

2.4) Conclusion

Domaines d'application des MOM : systèmes faiblement couplés

- Couplage temporel : interactions asynchrones/systèmes autonomes communicants
  - ◊ Communication en mode « push » → découverte des évolutions de l'environnement
  - ◊ Fonctionnement en mode déconnecté : site absent ou utilisateur mobile
- Couplage spatial : systèmes à grande échelle
  - ◊ Fonctionnement en mode partitionné/déconnecté
  - ◊ Communication « anonyme »
- Couplage sémantique : systèmes hétérogènes
  Modèle d'interaction minimal, s'appuyant sur une base universelle (TCP-UDP/IP)
  → possibilité d'intégrer des environnements(systèmes, réseaux)/applications hétérogènes

Exemples

- Supervision d'équipements distribués
- Echange, partage et intégration de données hétérogènes
  - ◊ génie logiciel (coopération entre outils de développement): SoftBench, ToolTalk, DecFuse…
  - ◊ diffusion d'informations/de logiciels sur WWW: iBus, Castanet, TIB/Rendezvous, SmartSockets…
- Intégration d'applications
  - ◊ Intra-entreprise: EAI (communication, routage, workflow) : KoalaBus…
  - ◊ Inter-entreprises: B2B et Web Services (communication, orchestration)

Etat des lieux

Domaine en évolution constante, largement utilisé

- large champ d'application
- besoins d'intégration grande échelle

Infrastructures propriétaires, mais début de normalisation

- autour de Java, interface JMS
- autour de CORBA : Corba Event Service

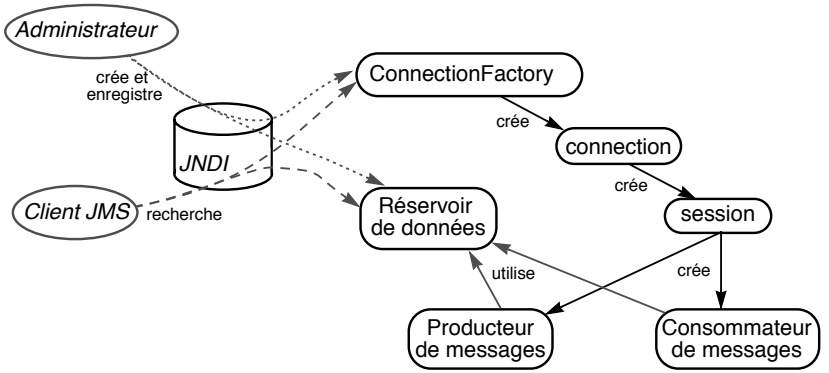
Référence

http://www.middleware.org (en particulier pages sur Message Oriented Middleware)

2.5) Exemple : JMS (Java Message Service)

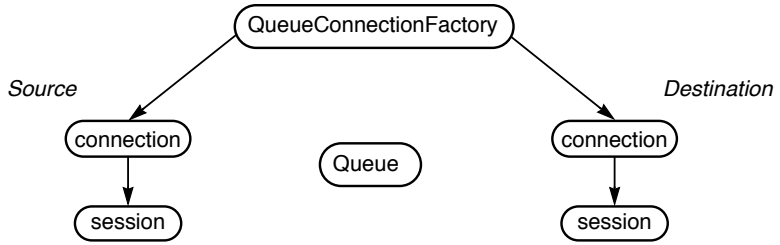
- Interface Java standard pour la communication par message
- Ne définit pas d'implémentation
  - ◊ Peut servir à interfacer un système existant : IBM MQ Series, Novell, Oracle, Sybase, Tibco
  - ◊ Des implémentations spécifiques peuvent être réalisées
  - ◊ Supporte point à point et publier/s'abonner

Architecture générale de JMS



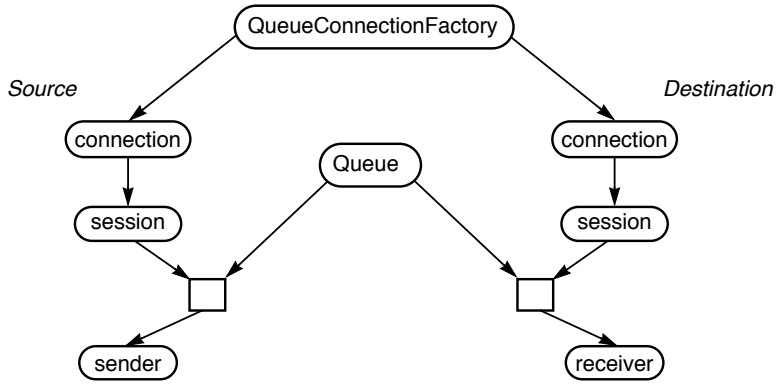
JMS en mode point à point

- Opérations préalables :
  - ◊ création de `Queue` et `QueueConnectionFactory` par l'administrateur (via un outil d'administration proposé par l'environnement de développement)
    Exemple : `j2eeadmin` pour le J2EE SDK
  - ◊ le client obtient la connexion avec un annuaire JNDI via un objet de la classe `Context`, obtenu par appel de la méthode `javax.naming.InitialContext()`



Sites source et destination

```
Context messaging = new InitialContext();
QueueConnectionFactory coFact =
         (QueueConnectionFactory) messaging.lookup("nom_fabrique") ;
Queue queue = (Queue) messaging.lookup("nom_queue.") ;
QueueConnection connection = coFact.createQueueConnection () ;
QueueSession session = connection.createQueueSession () ;
```

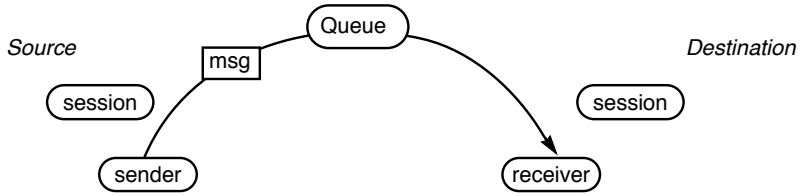



Site source

```
QueueSender sender = session.CreateSender (queue) ;
```

Site destination

```
String selector = new String ("(name = 'Bull') or (name = 'IBM')");
QueueReceiver receiver = session.CreateReceiver (queue, selector) ;
// -> Réception sélective des messages dont l'attribut name contient la chaîne donnée
```

Site source

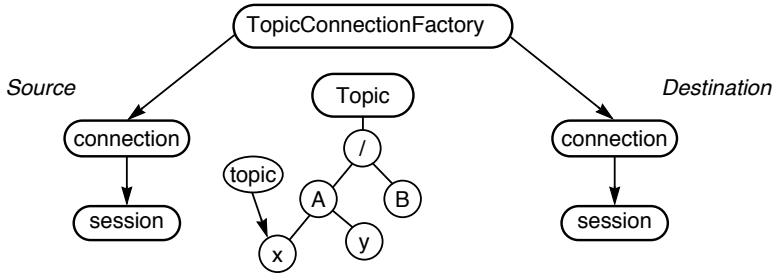
```
TextMessage msg = session.createTextMessage() ;
msg.setText ("le texte du message") ;
Sender.send (msg) ;
```

Site destination

```
TextMessage msg = (TextMessage) receiver.receive () ;
```

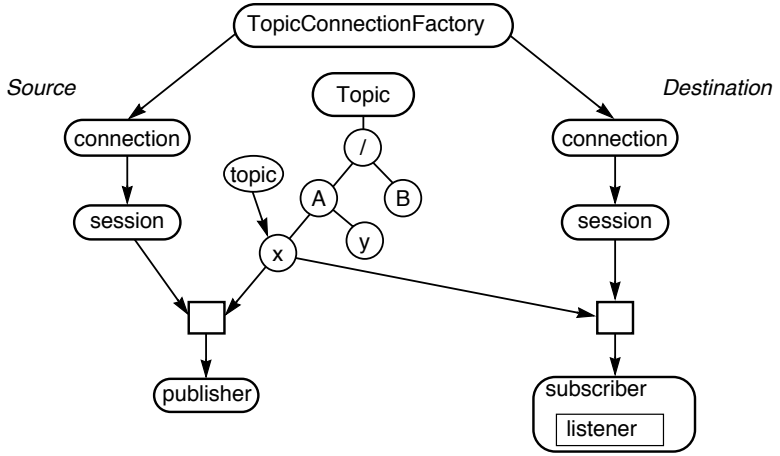

JMS en mode publier-s'abonner

- Opérations préalables :
  - ◊ création de `Topic` et `TopicConnectionFactory` par l'administrateur (via un outil d'administration proposé par l'environnement de développement)
  - ◊ le client obtient la connexion avec un annuaire JNDI via un objet de la classe `Context`, obtenu par appel de la méthode `javax.naming.InitialContext()`



Sites source et destination

```
Context ctx = new InitialContext();
TopicConnectionFactory coFact=(TopicConnectionFactory) ctx.lookup ("nom_fab") ;
Topic topic =(Topic) ctx.lookup ("/A/x") ;
TopicConnection connection = coFact.createTopicConnection () ;
TopicSession session=connection.createTopicSession(false,Session.CLIENT_ACKNOWLEDGE);
                                  // pas de transactions, acquittement à la réception
```

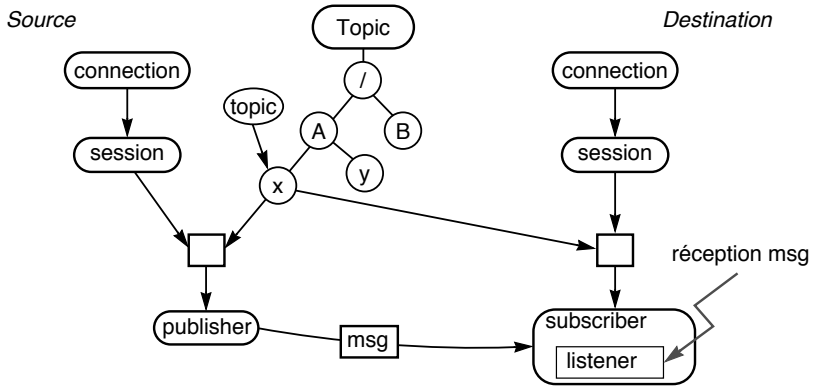


Site source

```
TopicPublisher publisher = session.CreatePublisher (topic) ;
```

Site destination

```
TopicSubscriber subscriber = session.CreateSubscriber (topic) ;
subscriber.setMessageListener(listener) ;
```

Site source

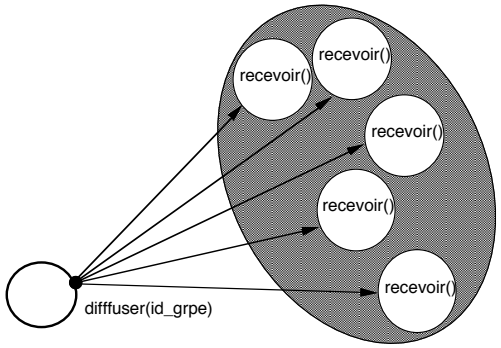
```
publisher.publish(msg) ;
```

Site destination

```
public class listener implements javax.jms.MessageListener {
  void onMessage (Message msg) throws JMSException {
     … // déballer et traiter le message
  }
}
```


2.6) Extension : diffusion/communication de groupe (Isis, Horus, Ensemble (Cornell))

groupe = ensemble (dynamique) de récepteurs



Applications :

- travail coopératif,
- tolérance aux pannes (gestion de la duplication, de l'atomicité)

Mise en œuvre :

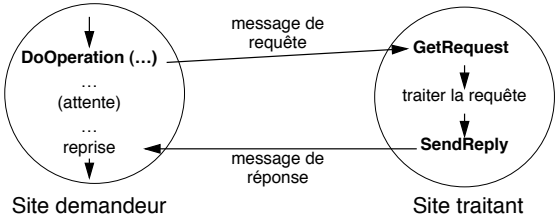
- utilisation possible du multicast IP
  difficultés : atomicité de la diffusion, ordre global des diffusions (si groupes non disjoints)

3) Interactions synchrones : schéma client-serveur

3.1) Schéma C/S : présentation/motivation

But : faciliter l'expression de certains motifs (formes) « récurrents » d'échanges employés pour la programmation d'applications réparties

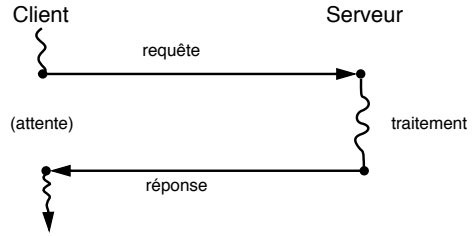
Motif de base : requête-réponse (Amoeba, Chorus, Mach…**)**



- réduction du nombre d'appels systèmes
- interaction **synchrone, pour le demandeur**
- la réponse peut tenir lieu d'acquittement

Client/Serveur (C/S)

Utilisation (systématique) du protocole requête-réponse
pour réaliser l'**encapsulation** de ressources



Espaces d'adressage séparés

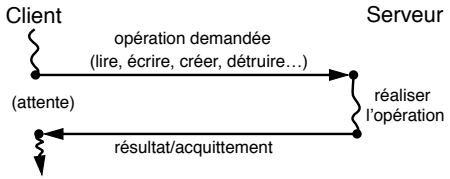
→ encapsulation { du service / de la ressource } → { gestion / mises a jour / protection / adaptation a la charge } facilites

Rôles différenciés (asymétrie)

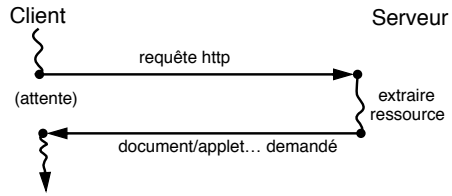
- vis-à-vis du *service*
  - ◊ le serveur offre un service (traitement, accès ressource)
  - ◊ le client consomme/demande le service
- dans la *désignation* : relation 1-n (serveur notoire/clients semblables, anonymes)
- dans l'*initiative* : client actif (entame l'échange)/serveur passif (attend les requêtes)

Exemples

Serveurs de fichiers (NFS)



Serveurs WWW



3.2) Mise en œuvre de l'interaction C/S

Sites séparés → nécessité de traiter

- les pannes de sites
- les différences { de representation / d'architecture } entre sites

Traitement des pannes

Prise en compte des pannes « franches »

- Panne totale d'un site
  - ◊ avec amnésie (pas de mémoire stable)
  - ◊ avec reprise depuis l'état initial
- Pertes de messages

Outils de base

Détection

- horloges de garde (compromis, vu l'impossibilité de détection en asynchrone pur)
- acquittements { positifs / negatifs }

Guérison

- identifier les messages (estampilles)
- répéter les émissions

Interaction client-serveur en présence de pannes

Sémantiques pour l'exécution d'opérations à distance

Au moins une fois

Client :

```
répéter
  armer une horloge de garde
  envoyer la requête
  attendre réponse ou délai
jusqu'à réponse
```

Serveur :

```
répéter
  recevoir requête
  traiter reuête
  envoyer réponse
sans fin
```

- OK si opérations idempotentes (calcul, lecture, écriture, suppression) → serveur *sans état*
- Nombre d'opérations ne sont *pas* idempotentes (données rémanentes…)
  Exemple : débit/crédit d'un compte bancaire

Au plus une fois

Client : ~ même protocole

Serveur : ne pas *traiter* deux fois une même requête (réémise plusieurs fois par un client)

- identifier les requêtes (*estampilles*)
- conserver un *historique* <id-requête, réponse> pour les requêtes déjà traitées
  → à la réception d'une requête réémise, la réponse est fournie (et réémise) grâce à l'historique
- le client ignorera les réponses aux requêtes déjà satisfaites
- serveur *avec état* : plus lourd, plus général
- réduire l'historique → utilisation d'acquittements (protocole RRA)

Remarque

La sémantique « exactement une fois » ne peut être réalisée avec une communication purement asynchrone (problème des deux généraux)

Traitement de l'hétérogénéité

Difficultés

- différences d'architecture (de représentation) entre site serveur et site client ;
- transmisssion de structures de données (SDD) complexes

Remède : convertir (processus d'emballage/déballage)

- entre représentations d'architectures différentes
- entre SDD complexes et flots d'octets *[(dé)linéarisation ou (dé)sérialisation]*

Choix

Pour les conversions de représentation

- passage par une représentation pivot (Exemples : XDR (Sun), XML (Web Services))
  + : nombre de traducteurs
  - : deux traductions par transfert
- conversion directe
  ◊ chaque architecture est identifiée
  ◊ les données transmises sont accompagnées de l'identifiant d'architecture
  ◊ la conversion est faite à l'arrivée, si les architectures diffèrent (Exemple : GIOP)
  - : nombre de traducteurs
  + : une traduction au plus

Pour la linéarisation

- type transmis avec les données (ASN1, Mach, Java, XML)
- ou supposé connu des correspondants (XDR, CDR (CORBA))

Exemples

Structure d'un message XDR (Sun)

2,18562,4,Gudule, Algo-Prog, 17

- Les données sont alignées sur des multiples de 4 octets
- Les entiers positifs sont codés sur 4 octets et transmis directement
- Les autres données sont précédées d'un entier positif (leur taille réelle) et complétées pour que la taille transmise soit un multiple de 4
- Les caractères sont codés en ASCII

2
18562
4
6
Gudu
le~~
9
Algo
-Pro
g~~~
17

Mécanisme de sérialisation en Java

- la conversion des données élémentaires est sans objet (échange entre JVM)
- une interface (`Serializable`) est proposée,
  qui définit deux méthodes (`readObjet` et `writeObject`) autorisant
  la réalisation et le contrôle fin de la (dé)linéarisation de graphes d'objets (voirTD)

3.3) Mise en œuvre du serveur

Architecture de principe

But : accroître le débit du serveur (nombre de requêtes traitées par unité de temps)

→ **paralléliser** le serveur

Schéma de base

- un processus **veilleur** (aiguilleur) réceptionne les requêtes sur le port associé au service
- chaque requête est transmise à un processus **exécutant**, qui la traite, puis transmet la réponse

Remarques

- couplage faible entre veilleur et exécutants → communication par boîte aux lettres
- ce schéma n'est pas avantageux pour des traitements courts
  (coût du traitement vs coût activation des exécutants
- activation des exécutants
  ◊ choix dans une poule fixe (couches basses)
  ◊ création dynamique (exemple : fork())

Serveur multiactivités

Activités vs processus

2 niveaux de coopération :

- activités (processus poids-plume, threads, LWPs)
  - ◊ correspondent à une décomposition d'un traitement en fonctions logiques
  - ◊ partagent mémoire et ressources (appartiennent à un même traitement/programme)
    → commutation légère
  - ◊ se coordonnent par verrous/conditions (~ sémaphores)
  - ◊ sont regroupés au sein d'un
- processus (lourd) (tâche, heavyweight process) : unité d'allocation de ressources
  Espaces d'adressages, ressources distincts → commutation coûteuse

Serveur réalisé par un processus multiactivités

→ gestion moins coûteuse des exécutants

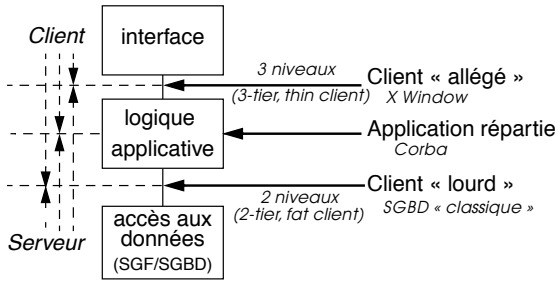
Rôle (affectation) des exécutants

- une activité (exécutant) par requête : convient au traitement de **requêtes** longues (SGBD…)
- une activité par client (connexion) : convient au traitement de **sessions** longues
- une (ou plusieurs) activité(s) par service
  - ◊ reprise de code existant
  - ◊ problème : équilibrage de charge

Pool (ferme) d'exécutants polyvalents (+ régulation par vol de tâches) ou création dynamique d'exécutants

Architectures C/S

Composants d'une application « générique »



Evaluation

2 niveaux (2-tier)

- souple
- utilisateur ≈ développeur
- l'utilisateur doit connaître le schéma de la base de données

3 niveaux (3-tier)

- gestion simple
- trafic réduit
- utilisateur ≈ consommateur
- nécessite un serveur puissant

3.4) Exemple : la bibliothèque des sockets

Note : certains paramètres d'appel des opérations présentées ici ont été omis, pour alléger la présentation.

Situation

Bibliothèque de communication assurant un service de niveau session ou transport.

- Les protocoles sous-jacents déterminent le *domaine* des sockets
  - ◊ UNIX (communications locales à une machine)
  - ◊ Internet (communication entre machines)
    - réseau/**liaison de donnée**s : Ethernet
    - **réseau** : IP (datagrammes, sans garantie d'arrivée)
    - **transport** : notion de service logique ("ports") :
      UDP (IP+ports), TCP (flots d'octets+fiabilité+ports)
    - les premiers ports (<1024) sont réservés aux services standards : echo (7), ftp (21)…
  - ◊ Autres domaines : XeroxNS, IBM SNA, CCITT (X25…), DECNet, AppleTalk…
  - ◊ Seules peuvent être connectées des sockets d'un même domaine (UNIX, Internet,…)
    → les sockets servent à réaliser les protocoles de niveau supérieur à celui du domaine
- Un socket peut être vu comme une porte/prise de communication :
  une BAL, vue par les processus comme (associée à) un (descripteur de) fichier
- Pour être désigné depuis un site distant, un socket doit être associé (implicitement ou explicitement) à une adresse réseau. Exemple (Internet) : numéro de port + adresse IP)

Principe

- la communication s'effectue entre **couples** de sockets (émetteur-récepteur).
  Image : une communication est identifiée par les 2 « prises de communication » qu'elle relie.
  - ◊ les messages émis sur le socket d'émission sont conservés jusqu'à être acheminés (et éventuellement acquittés)
  - ◊ les messages reçus sont conservés jusqu'à être consommés par le processus récepteur
- les sockets sont créés par appel à la procédure `socket`

```
ds = socket (domaine, type, protocole)
/* ds : descripteur de socket, c-à-d de fichier */
```

  - ◊ **type** définit les propriétés de la communication :
    - Datagramme (SOCK_DGRAM)
    - Connectée (SOCK_STREAM)
    - Brute (SOCK_RAW) : communication par les protocoles de bas niveau
  - ◊ **protocole** permet de préciser un protocole spécifique (ex : UDP, TCP…) ;
    la valeur 0 indique habituellement le protocole par défaut associé au domaine et au type.
- Un socket peut être associé à une adresse (publique) de socket (opération **bind**).
  Cette liaison est nécessaire au niveau du récepteur. Elle est dynamique, mais définitive.

Communication (Internet) en mode datagramme

- Une association (couple de sockets) est définie à chaque transmission
- Opérations :
  - ◊ **sendto** (socket source, message, socket destinataire,…)
  - ◊ **recvfrom** (socket récepteur, message, socket source,…)

```
s = socket(AF_INET, SOCK_DGRAM, 0)
…
bind(s, AdresseSource)
…
sendto(s, "hello", AdresseDestinataire)
```

domaine Internet

Emetteur

```
s = socket(AF_INET, SOCK_DGRAM, 0)
…
bind(s, AdresseDestinataire)
…
lgRecue = recvfrom(s, tampon, source)
```

retourne `AdresseSource`. Le récepteur (serveur) n'a donc pas à connaître l'adresse de l'émetteur.

Récepteur

- La taille des datagrammes est bornée (ex : UDP : 8K)
- Si le protocole sous-jacent le permet (ex : Ethernet), une émission de message peut être diffusée (partie hôte du destinataire = adresse de diffusion conventionnelle)

Communication (Internet) en mode connecté (exemple : interaction C/S

Demande l'établissement d'une liaison virtuelle.
→ le client *doit* connaître l'adresse du serveur.

```
s = socket(AF_INET, SOCK_STREAM, 0)
…
connect(s, AdresseServeur)
…
write(s, "hello", taille)
```

Client

Indique que des demandes de connexion peuvent être adressées au serveur sur *secoute*

5 : nombre maximum de demandes en attente.

attend 1 demande de connexion et établit une liaison virtuelle entre le socket du client et un socket de service créé pour l'occasion.

```
secoute = socket(AF_INET, SOCK_STREAM, 0)
…
bind(secoute, AdresseServeur)
listen (secoute,5)
…
while (1) {
…
sservice = accept(secoute,AdresseClient)
if (fork()==0) {
…
lgRecue = read(sservice, tampon, taille)
…} /* end if */
} /* end while */
```

Serveur

- *fork* lance un exécutant pour traiter la requête du client
- Les échanges s'effectuent par *read* et *write*

Remarque

L'établissement de la connexion est dissymétrique, mais la communication est symétrique.

Compléments

- *gethostbyname* permet d'obtenir l'adresse (IP) d'une machine, à partir de son nom.
- La prise en compte des demandes de service peut être soit séquentielle, soit réalisée par un appel à *select*
- Messages urgents : caractères OOB (Out Of Band) :
  - ◊ un canal logique double les liaisons entre sockets `SOCK_STREAM`
  - ◊ un caractère peut être émis sur ce canal, indépendamment du flot de données
  - ◊ le signal `SIGURG` accompagne la réception de ce caractère.
- *connect* peut aussi être utilisé en mode datagramme, fixant alors l'émetteur ou le récepteur des messages. L'émission et la réception se font alors respectivement par *write* et *read* .

Evaluation/comparaison

contrôle de l'intégrité des messages, dans tous les cas

mode connecté

+ : routage simplifié, traitement des pannes, de l'ordonnancement des messages

- : gestion de la connexion (établir/rompre)

→ adapté à la transmission de *flots* (continus) de données (voix, image…)

mode datagramme

+ : cf - supra ;

- : cf + supra

→ adapté aux interactions client/serveur simples (requête/réponse)

3.5) Variantes et extensions de l'interaction C/S

Appels de procédures à distance (RPC : Remote Procedure Call)

But : intégrer l'interaction C/S aux langages de programmation

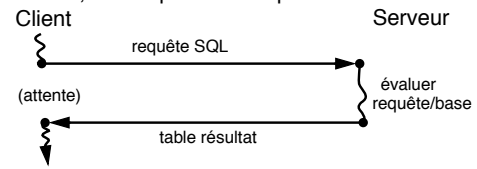
Idée : analogie entre l'interaction C/S et l'appel de procédure

C/S de données

Contexte : bases de données

Principe de fonctionnement :

- le client réalise les traitements non liés aux données, gère le dialogue avec l'utilisateur, et émet des requêtes (SQL)
- le serveur gère les données, et interprète les requêtes d'accès aux données



Remarque : requêtes ~ code, réponses ~ données

Code mobile

Programmes pouvant être transmis et exécutés d'un site à l'autre

- mobilité faible : pas de transfert du contexte d'exécution
  - ◊ serveurs de code : transfert du code seul (applets Java)
  - ◊ transfert du code, et des données modifiées (Aglets IBM)
- mobilité forte : transfert du contexte d'exécution (AgentTcl)

Motivations :

- permettre au programmeur d'intervenir sur la gestion des ressources (placement du code vis à vis des ressources)
- répondre à certains besoins applicatifs : informatique nomade, administration système ou réseau…

Service critique : sécurité

Architectures pair à pair (peer to peer, P2P)

- interactions synchrones (requête/réponse) entre sites
- répartition des ressources/services entre sites, pour équilibrer la charge et éviter les points de contention.

4)Etude de cas : supervision d'un réseau

Sources :

M. Riveil, A. Freyssinet : Modèles à bus de messages, école "Construction d'Applications Réparties", 1999
A. Freyssinet : Le système Joram, école "Intergiciel et Construction d'Applications Réparties", 2006

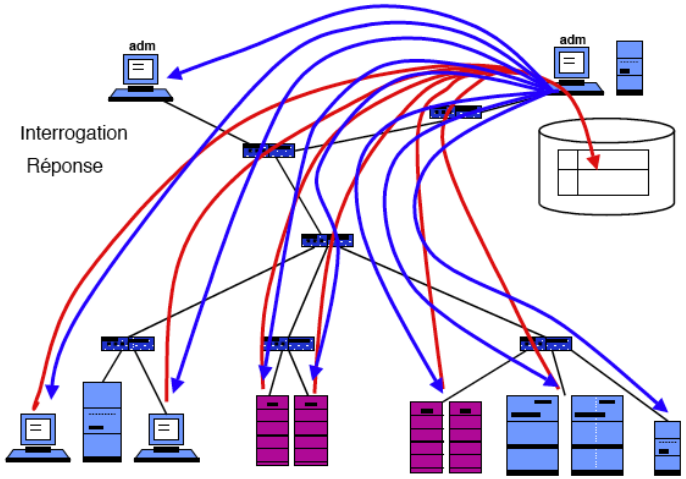
Contexte

- Surveillance de l'état de machines, systèmes et applications dans un environnement distribué
- Flot permanent de données en provenance de sources diverses sur le réseau
- Possibilité permanente d'évolution (ajout, suppression, déplacement des équipements)
- Possibilité d'accès des administrateurs depuis n'importe quel poste de travail

Objectif

- suivi des changements de configuration dynamiques
- émission de messages signalant les changements d'état et les mises à jour
- statistiques, journal de fonctionnement

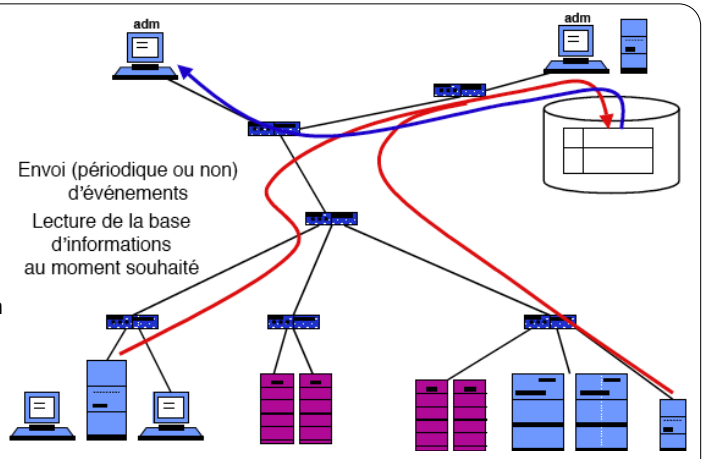
Approche client-serveur



- Interactions synchrones
- Communication essentiellement 1 vers 1 (ou n vers 1)
- Entités (clients, serveurs) désignées explicitement
- Organisation de l'application plutôt statique

Approche MOM
Schéma d'interaction

- Les composants administrés envoient des messages déclenchés par des événements
  - ◊ horloge (relevé périodique)
  - ◊ alertes, changement d'état ou de configuration
- Des processus cycliques (démons) mettent à jour l'état du système à partir des notifications reçues



Les MOM (événements, messages) fournissent dans ce cas un modèle mieux adapté que le C/S

- Communication asynchrone
- Communication possible n vers m
- Possibilité de désignation non explicite des entités
- Organisation dynamique des applications (facilité d'évolution, ajout/retrait de composants)

5) Synthèse : mise en œuvre de l'interaction C/S avec l'API sockets Java

5.1) L'interface sockets

Adresses IP

- Classe *InetAddress*'
  InetAddress.getByName("mozart.enseeiht.fr") : adresse d'un serveur à partir de son nom

Sockets UDP

- Classe *DatagramSocket*
  pseudo-connexion (*connect*), envoi (*send*) et réception (*receive*) de datagrammes
- Classe *DatagramPacket*
  initialisation d'un datagramme (constructeur), accès aux attributs (*getData*, *getPort*, *getAddress*)

Sockets TCP

- Classe *ServerSocket* : socket d'écoute (côté serveur, donc)
  - liaison et initialisation (listen+bind) sont réalisés par le constructeur
  - l'attente de requêtes (*accept*) retourne un socket de service (de classe *Socket*)
- Classe *Socket*
  établissement d'une connexion (constructeur), accès aux flots d'émission (*getOutputStream*) et de réception (*getInputStream*)

5.2) Mise en œuvre d'un serveur d'écho

Source : Coulouris, Dollimore, Kindberg : Distributed Systems, 4ème édition, http://www.cdk4.net/

a) avec le protocole UDP

Programme client (UDP)

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){ // args : message et nom serveur
    DatagramSocket aSocket = null;
    try {
       aSocket = new DatagramSocket();
       byte [] m = args[0].getBytes();
       InetAddress aHost = InetAddress.getByName(args[1]);
       int serverPort = 6789;
       DatagramPacket request = new DatagramPacket(m,args[0].length(),aHost,serverPort);
       aSocket.send(request);
       byte[] buffer = new byte[1000];
       DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
       aSocket.receive(reply);System.out.println("Rep:"+newString(reply.getData()));
    }catch (SocketException e){System.out.println("Socket:"+e.getMessage());
    }catch (IOException e){System.out.println("IO: " + e.getMessage());
    }finally {if(aSocket != null) aSocket.close();}
  }
}
```


Programme serveur (UDP)

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
    DatagramSocket aSocket = null;
    try{
     aSocket = new DatagramSocket(6789); // crée un socket sur le port convenu
      byte[] buffer = new byte[1000];
      while(true){
         DatagramPacket request=new DatagramPacket(buffer, buffer.length);
         aSocket.receive(request);
         DatagramPacket reply = new DatagramPacket(
                           request.getData(), request.getLength(),
                           request.getAddress(), request.getPort());
         aSocket.send(reply);
    }
    }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
    }catch (IOException e) {System.out.println("IO: " + e.getMessage());
    }finally {if(aSocket != null) aSocket.close();}
    }
}
```


b) avec le protocole TCP

Programme client (TCP)

```
import java.net.*;
import java.io.*;
public class TCPClient {
 public static void main (String args[]) { // arguments : message et hostname
    Socket s = null;
    try{
       int serverPort = 7896;
       s = new Socket(args[1], serverPort);
       DataInputStream in = new DataInputStream(s.getInputStream());
       DataOutputStream out = new DataOutputStream(s.getOutputStream());
       out.writeUTF(args[0]);      // UTF : codage des caractères
       String data = in.readUTF();    // lire une ligne du flot
       System.out.println("Received: "+ data) ;
    }catch (UnknownHostException e){System.out.println("Socket:"+e.getMessage());
    }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
    }catch (IOException e){System.out.println("readline:"+e.getMessage());
    }finally {if(s!=null) try {
       s.close();
       }catch (IOException e){System.out.println("close:"+e.getMessage());}}
     }
}
```

Programme serveur (TCP)

```
import java.io.*;
public class TCPServer {
 public static void main (String args[]) {
   try{
      int serverPort = 7896;
      ServerSocket listenSocket = new ServerSocket(serverPort);
      while(true) {
         Socket clientSocket = listenSocket.accept();
         Connection c = new Connection(clientSocket);
      }
   } catch(IOException e) {System.out.println("Listen socket:"+e.getMessage());}
 }
}

class Connection extends Thread {
// ... (détaillée dans le prochain transparent)...
}
```



```
class Connection extends Thread {
 DataInputStream in;
 DataOutputStream out;
 Socket clientSocket;

 public Connection (Socket aClientSocket) {
   try {
      clientSocket = aClientSocket;
      in = new DataInputStream(clientSocket.getInputStream());
      out =new DataOutputStream(clientSocket.getOutputStream());
      this.start();
   } catch(IOException e)
{System.out.println("Connection:"+e.getMessage());}
 }

 public void run(){
   try {    // serveur réalisant l'écho des flots reçus
      String data = in.readUTF(); // lire une ligne du flot d'entrée
      out.writeUTF(data);
   } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
   } catch(IOException e) {System.out.println("readline:"+e.getMessage());
   } finally{ try {
         clientSocket.close();
      }catch (IOException e){/* erreur sur fermeture du socket*/}}
 }
}
```