Introduction aux systèmes concurrents et aux systèmes répartis

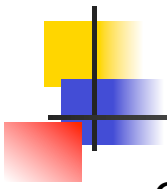
Laurent Pautet
Bertrand Dupouy
Thomas Robert
Ada Diaconescu

Laurent.Pautet@enst.fr

Version 1.4

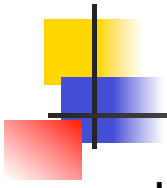
Paradigmes (1/3)

- Une activité logicielle consiste en une suite d'actions
  - Réception de données (valeurs, événements …)
  - Traitement des données reçues (calcul / conversion …)
  - Emission de données en sorties (valeurs, événements …)
- Toute activité consomme des ressources matérielles au cours du temps
- Le comportement d'un système logiciel ~ un ensemble d'activités partageant le matériel
- Deux approches pour ordonner l'exécution de ces activités sur le matériel :
  - Sérialisation des traitements : une activité après l'autre
    ⇒ il faut tout prévoir pour figer la séquence d'exécution
  - Traitement concurrent : partage temporel des ressources
    ⇒ il faut pouvoir interrompre et reprendre une activité
- Coût de la sérialisation prohibitif pour les systèmes complexes



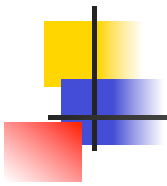
Paradigmes (2/3)

- Ordonnancement et processus : comment assurer le partage à l'exécution du matériel entre diverses activités sur un ordinateur ? (flexibilité)
- Programmation concurrente : comment coordonner des activités distinctes partageant leur mémoire et les média de communication ?
- Programmation répartie : comment coordonner des activités distinctes exécutées sur des matériels distincts ?



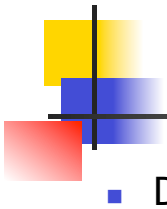
Paradigmes (3/3)

- Le système d'exploitation fournit les services élémentaires pour la programmation concurrente et répartie
- Cette programmation dépend des services de communication et synchronisation utilisés au niveau du langage de programmation.
  - Niveau « système d'exploitation » (POSIX & socket)
  - Niveau « machine virtuelle » (Moniteurs et RMI)
  - Niveau « intergiciel » (Objets distants)
- Une étude de cas pour tout illustrer…



Déroulement du cours

- Etude de la programmation concurrente
  - POSIX – concurrence basée sur une interface de programmation
  - Java – concurrence basée sur un langage de programmation
- Etude de la programmation réseau
  - Socket – répartition basée sur une interface de programmation
- Etude des objets répartis
  - RMI – répartition basée sur un langage de programmation
  - CORBA – répartition basée sur un langage de description
- Etude des design patterns de concurrence et répartition
- Contrôle continu
  - Tout TP se termine par un travail à la maison sur un cas d'étude
- Contrôle de connaissances
  - Il comportera des questions sur le travail fait sur un cas d'étude



Etude de cas (1/2)

- Des étudiants échangent un album de photos et chacun souhaite y rajouter de nouvelles photos.
- Pour ne perdre aucune photo, un seul étudiant doit avoir, à un moment donné, la possibilité de modifier l'album => l'album est muni d'un « verrou réparti»
- L'étude de cas concerne la réalisation de ce verrou réparti selon différentes méthodes de programmation
- Un « protocole » permet aux étudiants de :
  - se mettre d'accord sur qui manipule l'album à chaque instant
  - s'assurer qu'au plus une personne modifie l'album

Etude de cas (2/2)

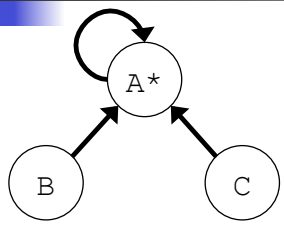
Principe

- Chaque étudiant se décompose en deux activités :
  - un « acteur » qui effectue la manipulation de l'album
  - un « agent » qui interagit avec les autres étudiants
- L'accès à l'album est un jeton échangé entre acteurs

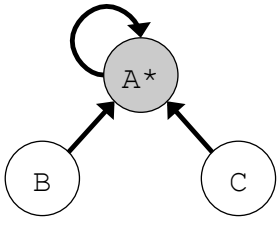
« Protocole »

- Chaque agent tient à jour l'identité du dernier demandeur dont il a reçu la requête (*lastOwner*)
- Chaque agent tient à jour l'identité du demandeur auquel il a promis d'envoyer le jeton lorsqu'il en disposera (*nextOwner*)
- Lorsqu'il reçoit une requête pour obtenir le jeton
  - S'il détient le jeton, il note qu'il doit faire suivre celui-ci lorsqu'il n'en a plus besoin en mettant à jour lastOwner et nextOwner
  - Sinon, il fait suivre la demande au dernier destinataire (lastOwner courant) et met à jour son lastOwner pour des demandes futures

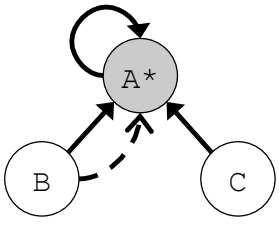
Scénario possible



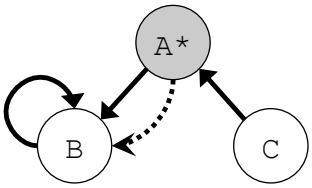
A possède O



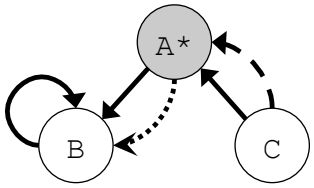
A modifie O



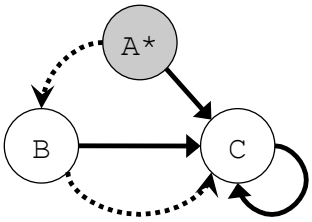
B veut modifier O



A désigne B comme propriétaire et lui enverra O



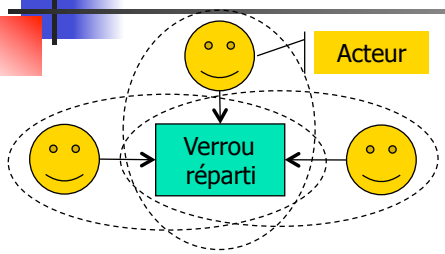
C veut modifier O
A redirige la demande vers B



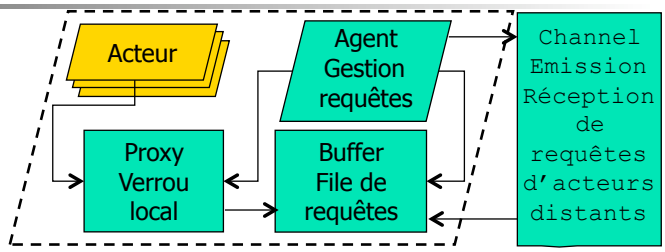
B désigne C comme propriétaire et lui enverra O

→ lastOwner ⇢ nextOwner ⇢ requête

Architectures



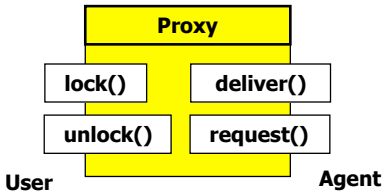
Architecture globale



Architecture détaillée d'un nœud

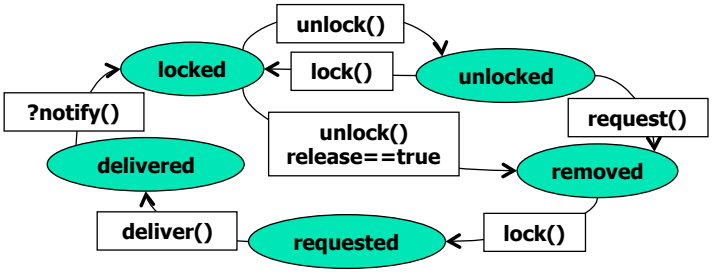
- Un nœud se compose d'un acteur (étudiant), d'un proxy (interface vers le verrou), d'un agent (algorithme) et d'une couche de communication
- Un proxy propose la même interface qu'un objet mais s'interpose entre lui et l'utilisateur afin de masquer la complexité de l'objet réel
- En local, un acteur accède au verrou réparti au travers d'un proxy
- Le proxy attribue en local le verrou (*lock* et *unlock*) mais cède ou reprend le verrou à son agent qui interagit avec les autres noeuds
- Un agent reçoit les requêtes des autres agents ou de son proxy
- Cet agent soit reprend le verrou au proxy et le transmet à un agent distant soit fait suivre la requête (éventuellement celle de son proxy)

Interface du proxy



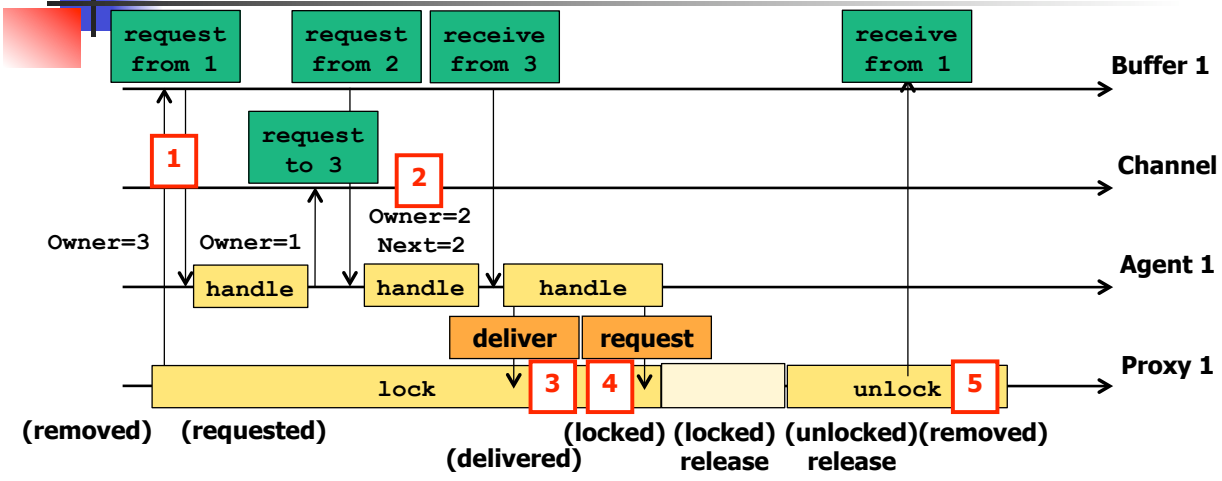
- Le proxy agit en verrou classique (*lock + unlock*)
- L'agent traite les requêtes d'autres nœuds ou du proxy.
- Il peut exiger du proxy de relâcher le verrou (*request*) pour le faire suivre à d'autres noeuds.
- Dans un *lock* en état *removed*, le proxy demande à l'agent de récupérer le verrou auprès d'autres nœuds.
- Lorsque l'agent délivre le verrou, il change l'état du proxy en *delivered* et le signale au proxy grâce à *deliver*.

Automate du proxy



- Le proxy dispose des états classiques *locked* et *unlocked*
- Lorsque l'agent requiert du proxy de rendre le verrou (*request*),
  - Si celui-ci le peut, il transmet le verrou et passe en état *removed*
  - Sinon, la variable *release* passe à true pour que lors du prochain *unlock* il transmette le verrou et passe en état *removed*
- Lors d'un *lock* en état *removed*, le proxy demande le verrou
  - Le proxy demande le verrou à l'agent grâce à une requête classique à l'agent
  - L'état passe à *requested* pour ne solliciter l'agent qu'une seule fois
- A la réception du verrou, le proxy passe dans l'état *delivered*.

Scenario à traiter avec attention



1. Proxy 1 n'a pas le verrou et le demande à Agent 1 qui fait suivre au Nœud 3
2. Agent 2 demande le verrou que Nœud 1 va recevoir,
   Nœud 1 prend en compte la demande en mettant à jour lastOwner et nextOwner
3. Agent 1 reçoit le verrou et le délivre à Proxy 1 qui débloque Actor 1 et dans la continuité …
4. Nœud 2 étant propriétaire (étape 2), Agent 1 demande à Proxy 1 de rendre le verrou (request)
5. A la suite de Unlock, Proxy 1 transmet le verrou à Agent 1 qui fait suivre à Nœud 2