Multitâche & objets

Modélisation objet des paradigmes multitâches



© Shebli Anvar

CEA — Institut de recherches sur les lois fondamentales de l'Univers Centre de Saclay — 91191 Gif-sur-Yvette — France shebli.anvar@cea.fr

0



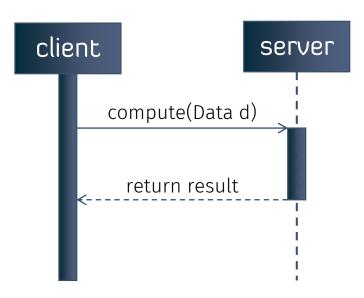
Communication inter-objets

- Un objet est une instance de classe
- La classe encapsule sa structure interne
- La classe spécifie une interface à base d'opérations



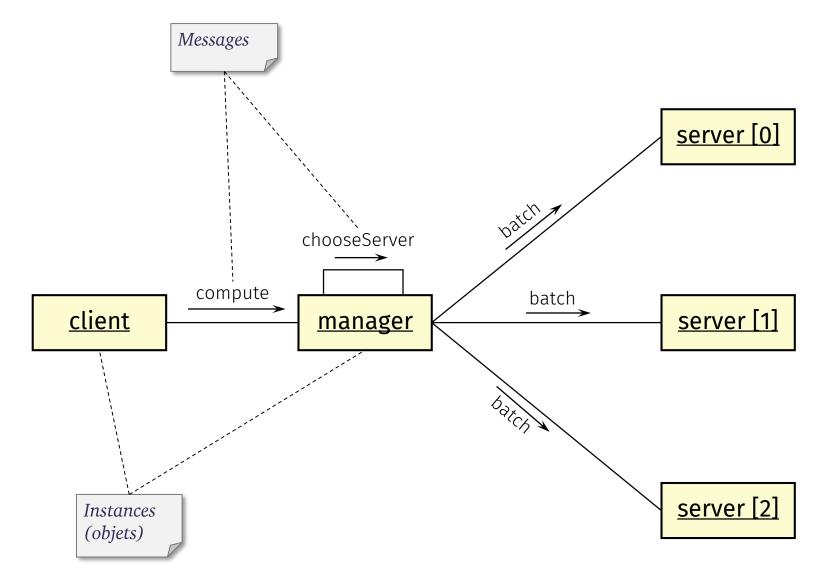
Communication inter-objets

appels d'opérations données échangées = paramètres



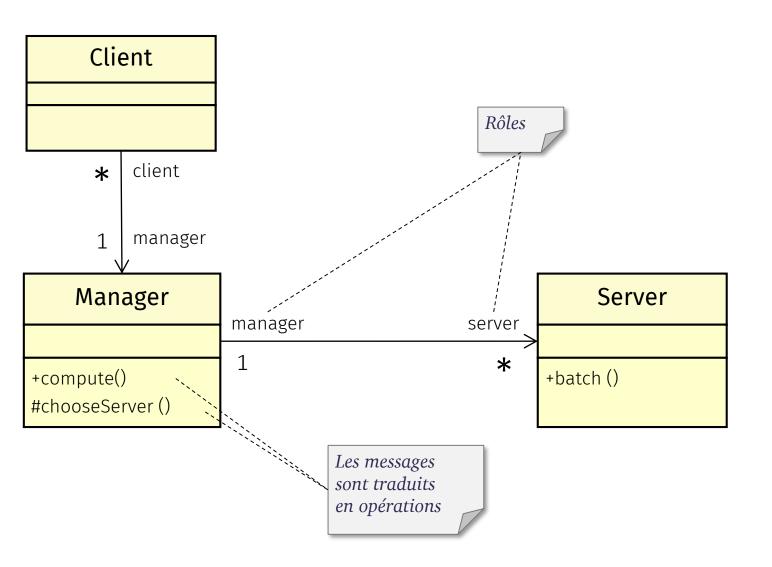


Collaboration entre instances





Abstraction des classes





```
class Client
{
private:
    Manager* manager;
};
```

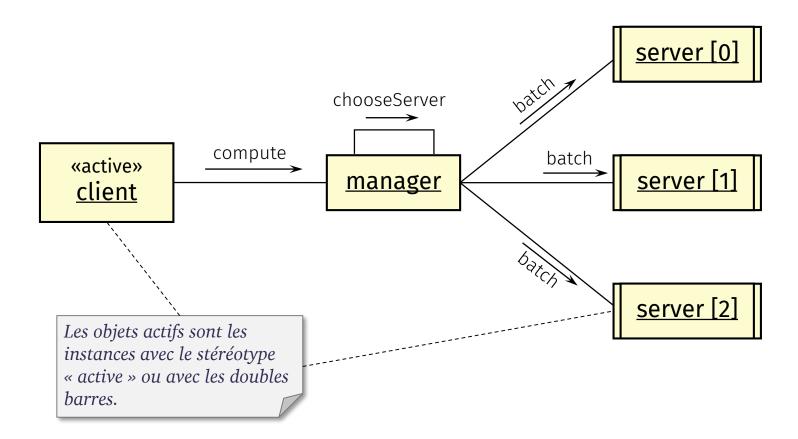
```
class Manager
{
private:
    std::vector<Server*> server;
public:
    double compute(Data input);
protected:
    int chooseServer();
};
```

```
class Server
{
public:
    double batch(Data input);
};
```



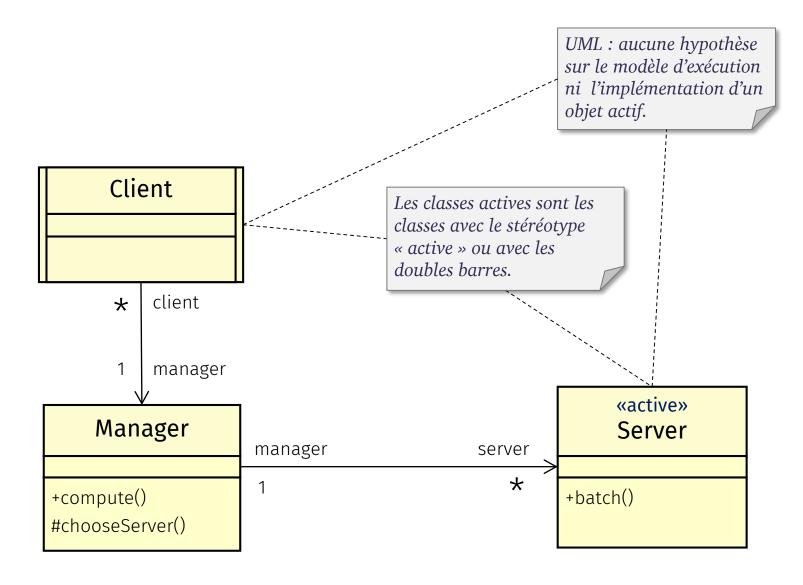
Objets actifs

Un « objet actif » est un objet qui encapsule son (ses) propre(s) fil(s) d'exécution





Abstraction des classes objets actifs





Notions à transposer

- Tâche (Thread)
 - Création, lancement
 - Endormissement, suspension
 - Arrêt, destruction
 - Attente d'arrêt (join)
- Mutex
 - Création, destruction
 - Types (simple, récursif...)
 - Prise et rendu de jeton
 - Rendu automatique
- Condition
 - Association avec Mutex
 - Attente et notification
 - Timeout

Sémaphore

- Binaire, à compte
- Conditions initiales
- Prise et rendu de jetons
- Timeout
- Communication
 - Asynchrone (file d'attente)
 - Synchronisation différée
 - □ À distance
 - Broadcast
- Encapsulation
 - Objets thread-safe
 - Objets actifs



La notion d'objet actif

- Fusion entre objet et tâche
 - Modèle d'exécution
 - monotâche
 - multitâche (une par opération)
 - Appel d'opération asynchrone
 - Appel d'opération à distance
- En pratique:
 - La classe dérive d'une classe « Thread »
 (ou implémente une interface ad hoc comme l'interface Runnable en Java)
 - Met en œuvre une file d'attente de requêtes d'exécution

Exemple de classe Thread

Création d'un objet Thread **Thread** non encore démarré. On distingue la création de l'objet Thread (constructeur) ... -posixId: pthread_t de son démarrage (start). «constructor» Création d'un objet Thread ·+Thread() à partir un thread Posix déjà démarré. «constructor» +Thread(posixId: pthread_t «destructor» +Thread() +start(prio:int) **Questions** +join(): void On peut redémarrer un objet Pourquoi la méthode Thread une fois qu'il terminé Thread::run() doit être: +join(timeout : int) : bool son exécution.

#run(): void --

- protected?

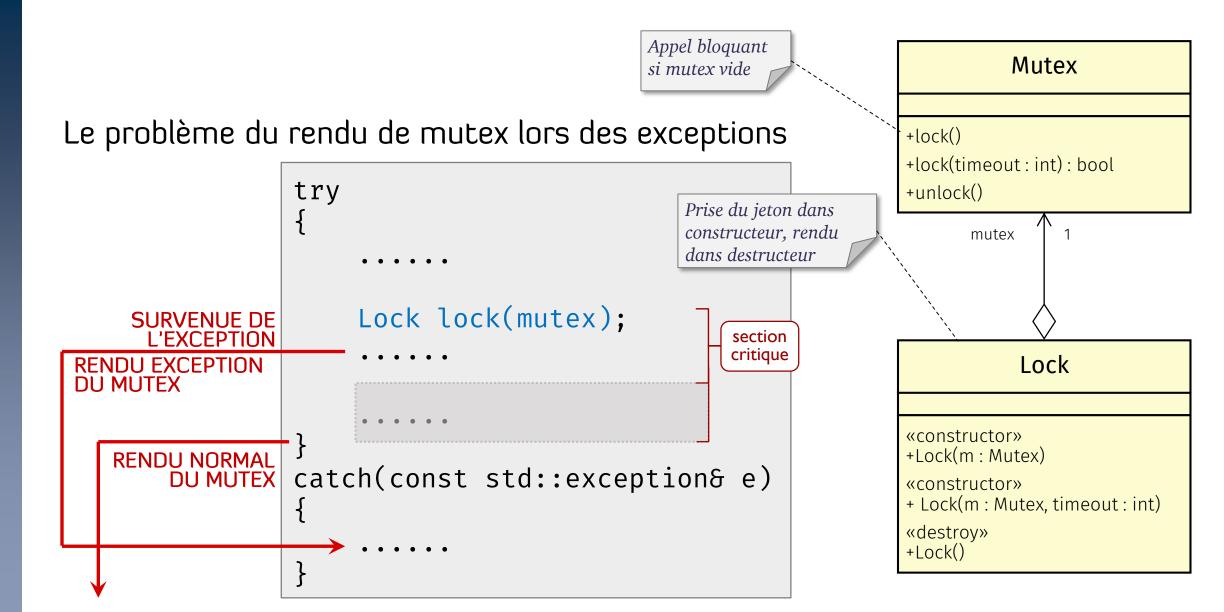
- virtuelle pure?

Les classes Mutex et Lock

Appel bloquant Mutex si mutex vide Le problème du rendu de mutex lors des exceptions +lock() +lock(timeout:int):bool try +unlock() Prise du jeton dans constructeur, rendu mutex dans destructeur mutex.lock(); SURVENUE DE L'EXCEPTION section critique Lock mutex.unlock(); «constructor» +Lock(m: Mutex) catch(const std::exception& e) «constructor» + Lock(m : Mutex, timeout : int) «destrov» +Lock() MUTEX NON RENDU

Isfu

Les classes Mutex et Lock





Les classes Mutex et Lock

Appel bloquant Mutex si mutex vide Le problème du rendu de mutex lors des exceptions +lock() +lock(timeout:int):bool try +unlock() Prise du jeton dans constructeur, rendu mutex dans destructeur Lock lock(mutex); **SURVENUE DE** section L'EXCEPTION critique Lock **RENDU EXCEPTION DU MUTEX** RENDU NORMAL DU MUTEX «constructor» +Lock(m: Mutex) catch(const std::exception& e) «constructor» + Lock(m : Mutex, timeout : int) «destrov» +Lock()



tâche

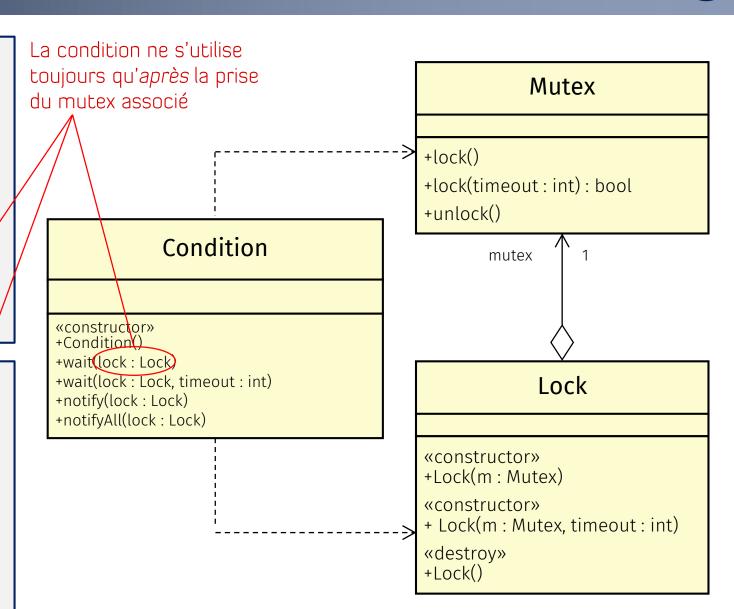
 $\mathbf{\omega}$

tâche

Classe Condition

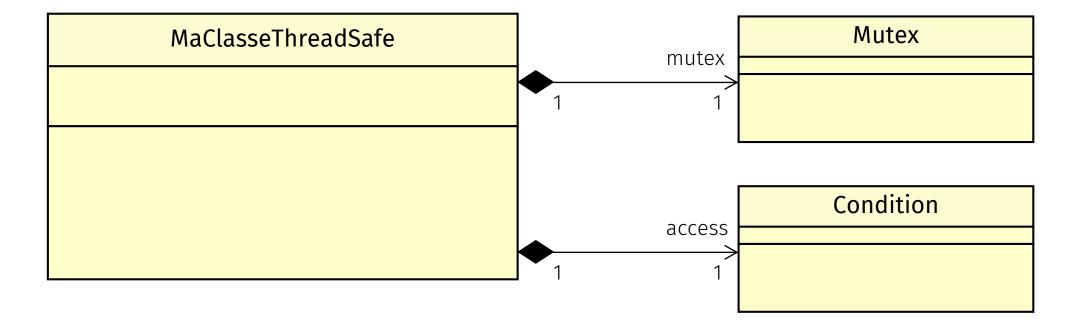
```
void waitStop (
    volatile int* pCommand,
    Mutex* mtx,
    Condition* cnd
)
{
    Lock lock(mutex);
    while (*pCommand != STOP)
    {
        cnd->wait(&lock);
    }
}
```

```
void doStop (
    volatile int* pCommand,
    Mutex* mtx,
    Condition* cnd
)
{
    Lock lock(mutex);
    *pCommand = STOP;
    cnd->notify((&lock);
}
```





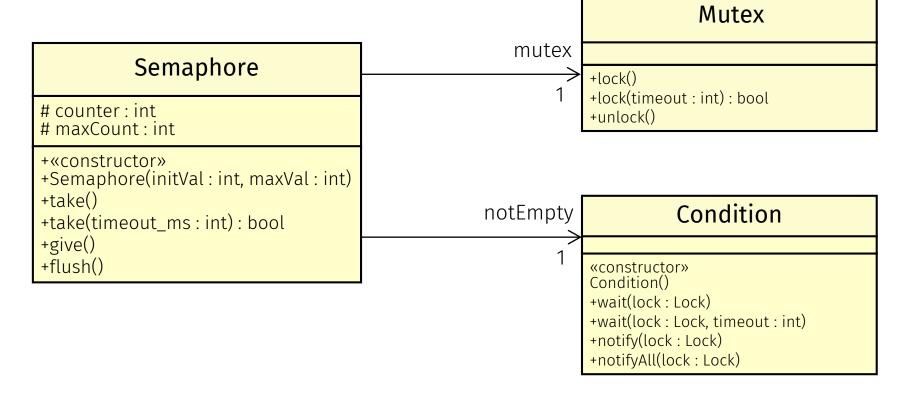
 Toutes les opérations d'une classe thread-safe doivent accéder ou modifier l'état de l'objet en garantissant les appels concurrents.





Classe Semaphore

- Un sémaphore est un compteur de jetons Lorsqu'il est vide, la demande de jeton bloque la tâche Déblocage : une autre tâche fournit un jeton
- Mécanisme de blocage déblocage ?
 - Variable partagée : compteur de jetons
 - Utilisation d'un Mutex et d'une Condition



Infa.

Objet passif: appel d'opération synchrone

```
:Client
                                                                             calc:Calculator
                                      crunch (param)
           result
Client::run(Calculator* calc)
        int param = 10;
        double result = calc->crunch(param);
```



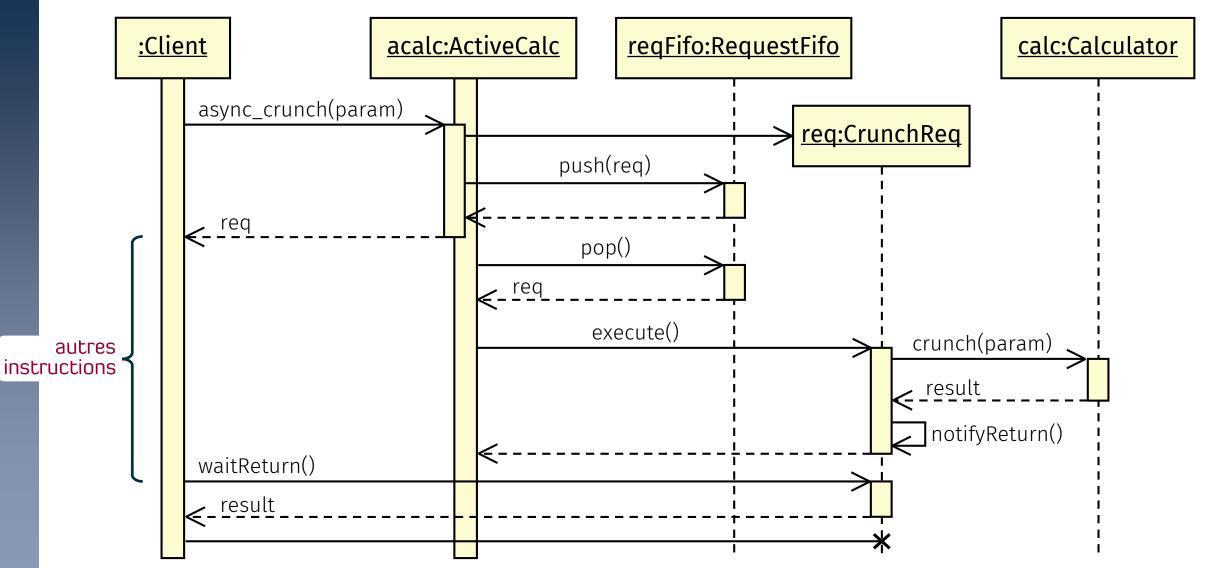
Objet actif: appel d'opération asynchrone

```
2017-2018
                :Client
                                                                                          acalc:ActiveCalc
                                            async_crunch (param)
       autres
  instructions
                        result
            Client::run(Calculator* calc)
                    int param = 10;
                    Request* req = acalc->async_crunch(param);
                    // ... Autres instructions
                    double result = req->waitReturn();
```



Shebli Anvar — Multitâches & objets







Objet actif: décomposition de l'appel asynchrone

void Client::main(ActiveCalculator* acalc) {

```
CrunchReq* req = acalc->async_crunch(10); // requête
   // ..... // Autres instructions
double result = req->waitReturn(); // Attente result
CrunchReq* ActiveCalc::async_crunch(double param) {
    CrunchReq* req = new CrunchReq(param); // Création de la requête d'exécution
    reqFifo.push(req);
                                       // Envoi de la requête
                                              // Transmission au client
    return req;
void ActiveCalc::run() {
    while(true)
        CrunchReq* req = reqFifo.pop();  // Réception de la requête
                                              // Exécution de la requête
        req->execute();
double CrunchReq::waitReturn() {
                                              // attente de la fin d'exécution du calcul
   returnSema.take();
                                              // renvoi du résultat de calcul à l'appelant
    return result;
void CrunchReq::execute() {
   result = calc->crunch(param);
                                              // exécution effective du calcul
                                              // notification du sémaphore « fin de calcul »
   returnSema.give();
```



Objet actif: classes principales

