



Multitâche & objets

Modélisation objet des paradigmes multitâches



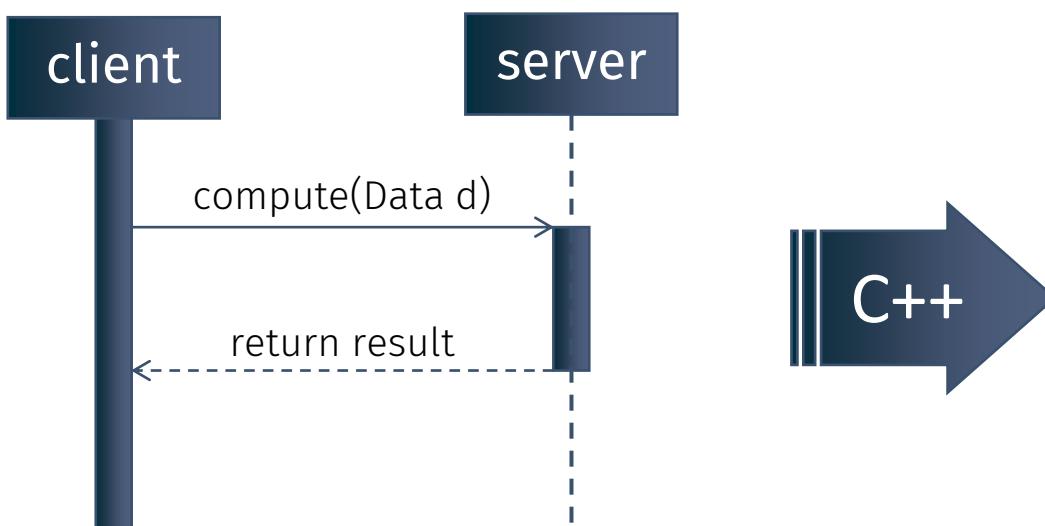
shebli.anvar@cea.fr
+33 1 69 08 61 60
+33 6 63 31 92 26

CEA Paris-Saclay
91191 Gif-sur-Yvette
France



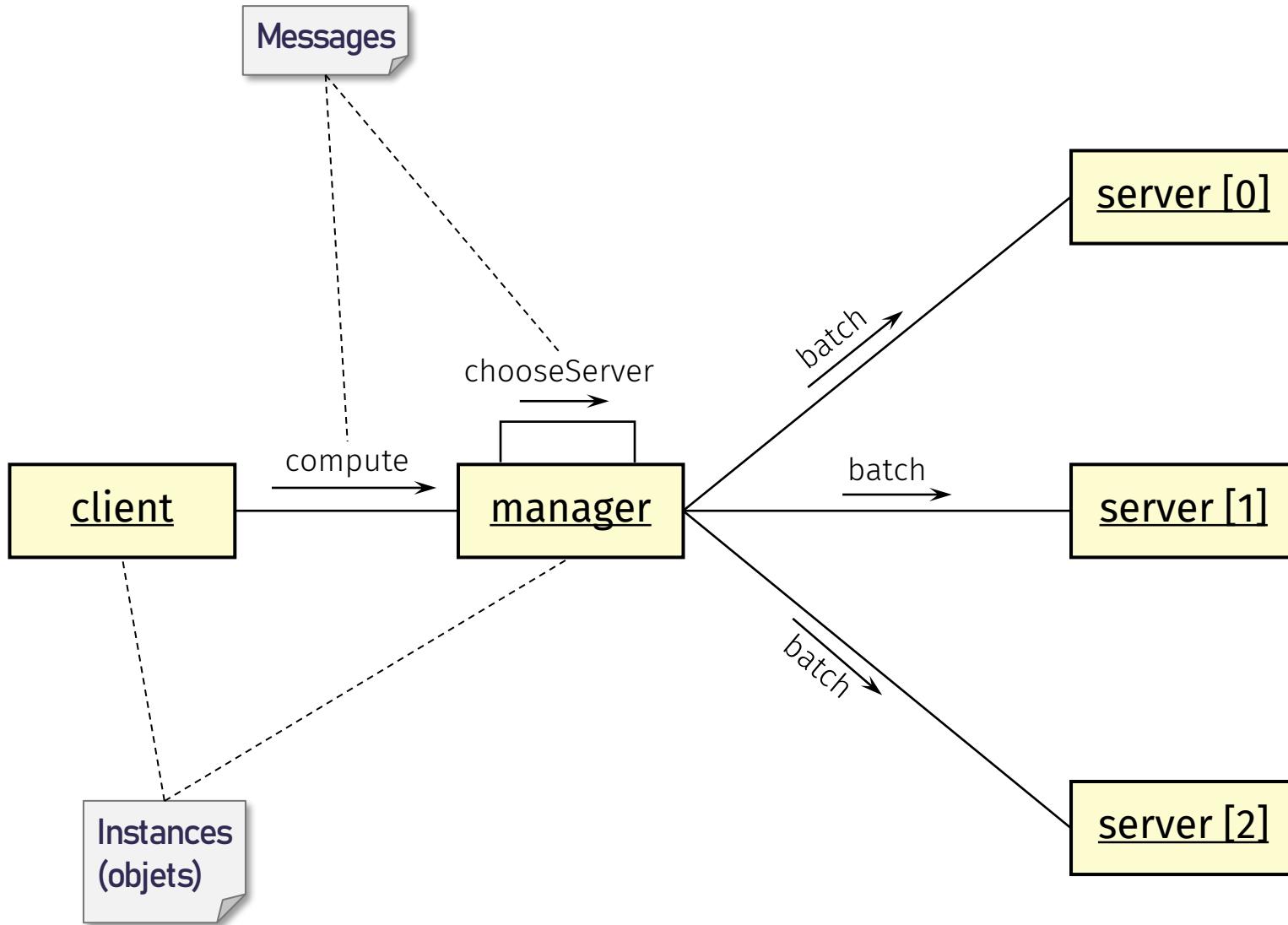
- Un objet est une instance de classe
- La classe encapsule sa structure interne
- La classe spécifie une interface à base d'opérations

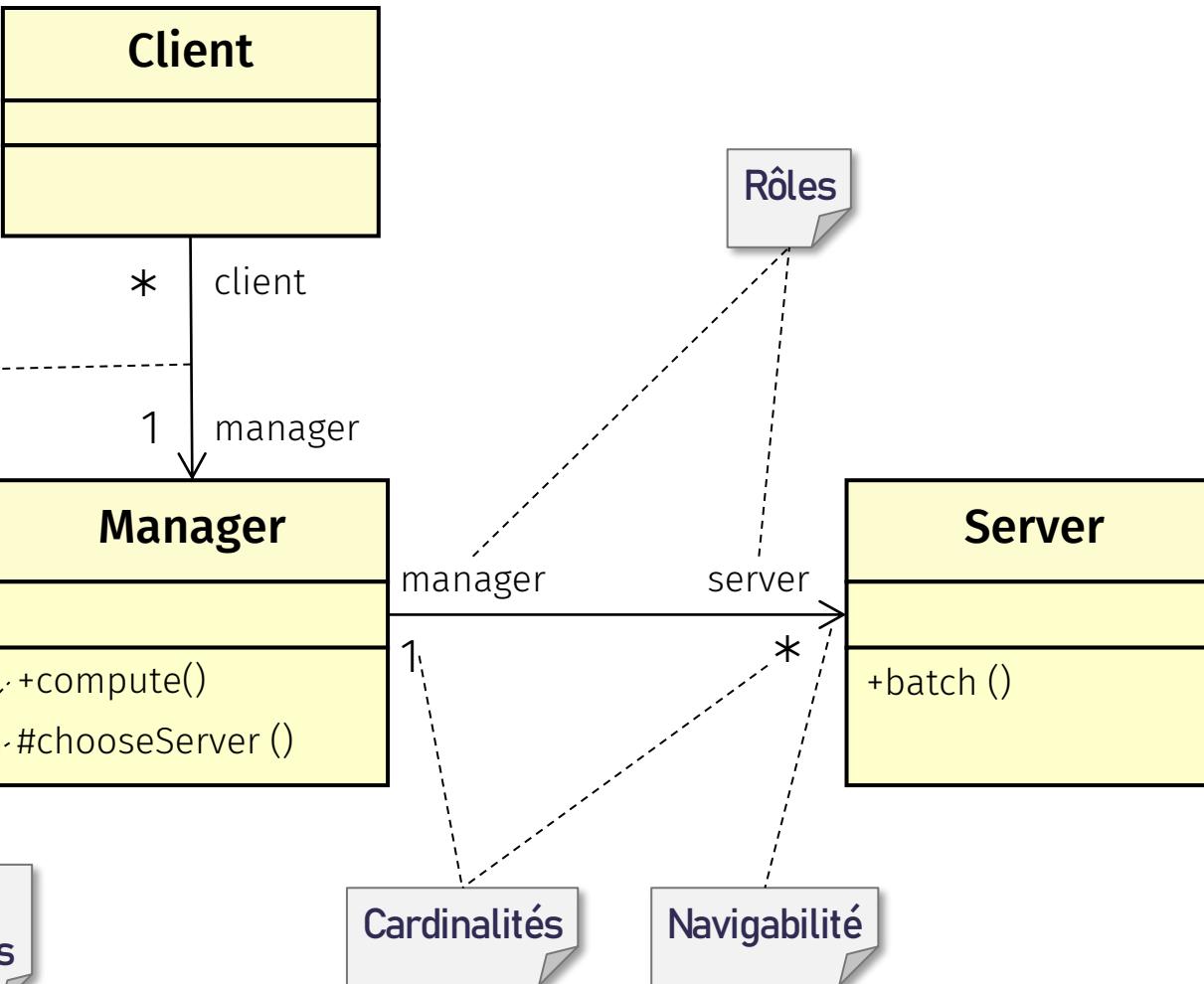
→ Communication inter-objets { appels d'opérations
données échangées = paramètres



||| C++ →

```
Client::run()
{
    result = server.compute(d);
};
```





```
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);
};
```

TD
2c, 2d, 2e

■ 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

Le problème du rendu de mutex lors des exceptions

```
try
{
    .....
    mutex.lock();
    .....
    mutex.unlock();
    .....
}
catch(const std::exception& e)
{
    .....
}
```

SURVENUE DE L'EXCEPTION

MUTEX NON RENDU

section critique

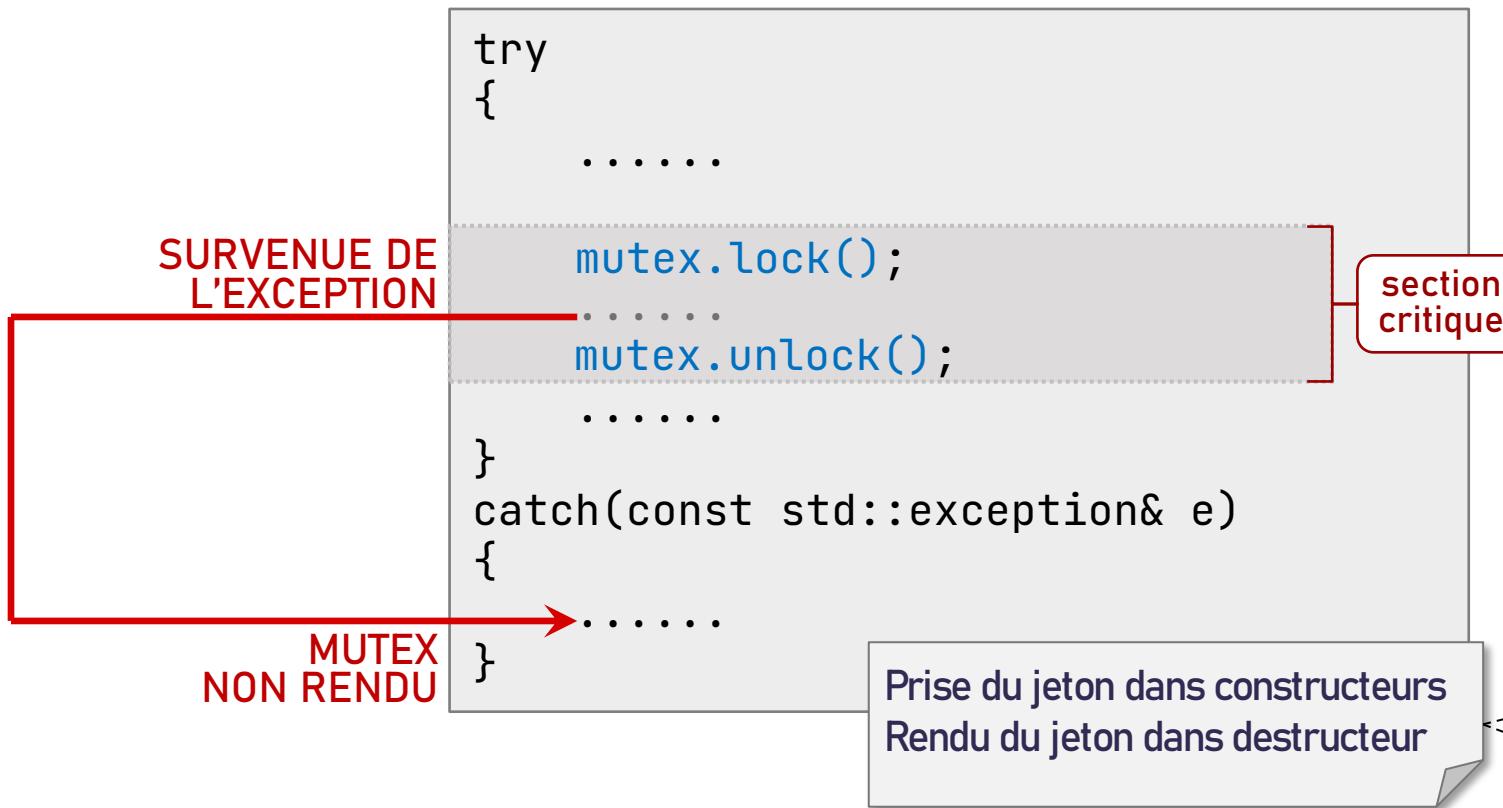
Appel bloquant si mutex vide

Mutex

-posixId : pthread_mutex_t
+Mutex(isInversionSafe : bool)
+~Mutex()
+lock()
+lock(timeout_ms : long) : bool
+unlock()

Le problème survient également lorsqu'on sort de la section critique par un « return » interne

Le problème du rendu de mutex lors des exceptions

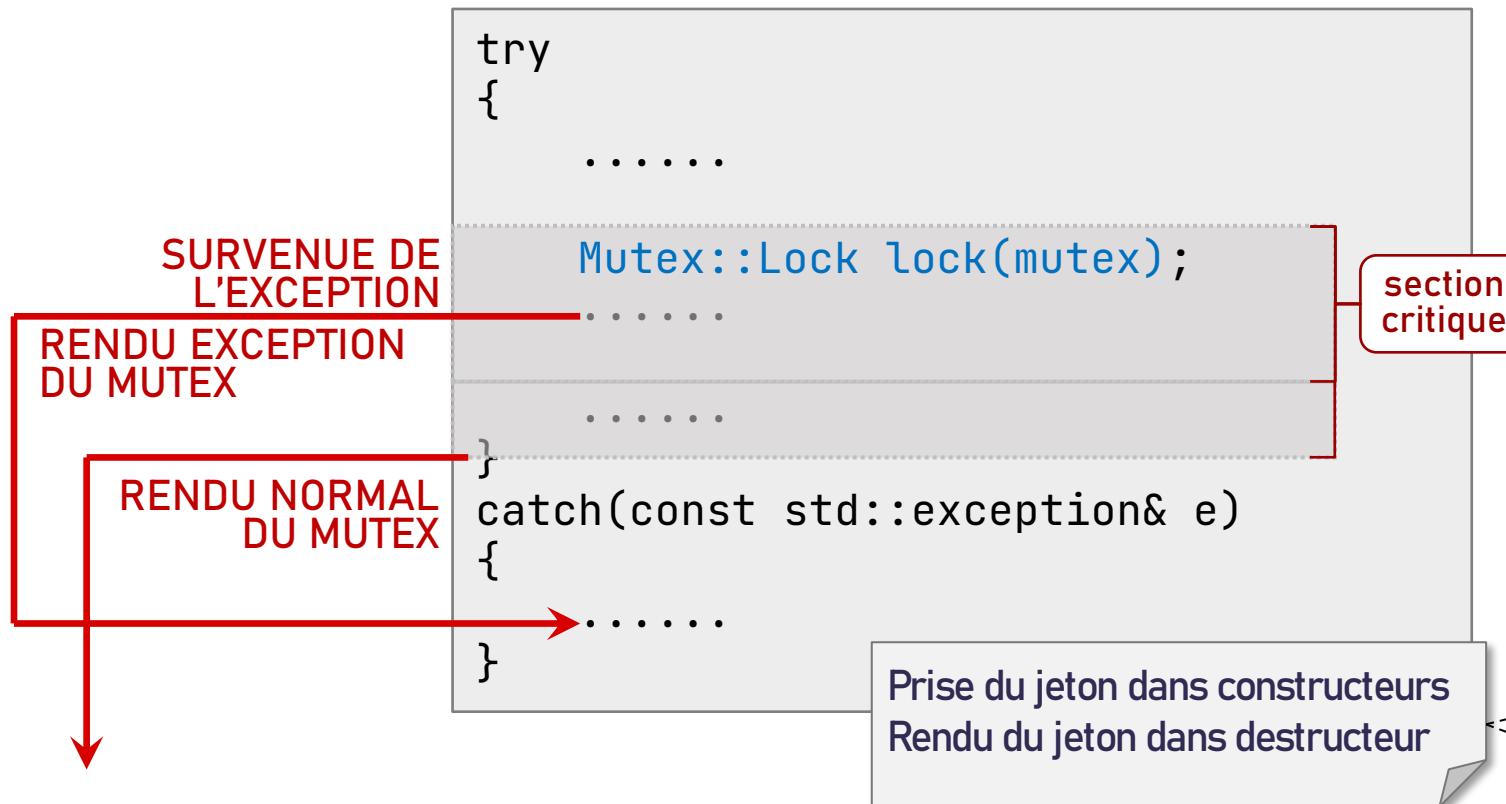


Mutex
-posixId : pthread_mutex_t
+Mutex(isInversionSafe : bool)
+~Mutex()
-lock()
-lock(timeout_ms : long) : bool
-unlock()

mutex ↑ 1

Mutex::Lock
+Lock(m : Mutex&)
+Lock(m : Mutex&, timeout_ms : long)
+~Lock()

Le problème du rendu de mutex lors des exceptions



Appel bloquant si mutex vide

Mutex

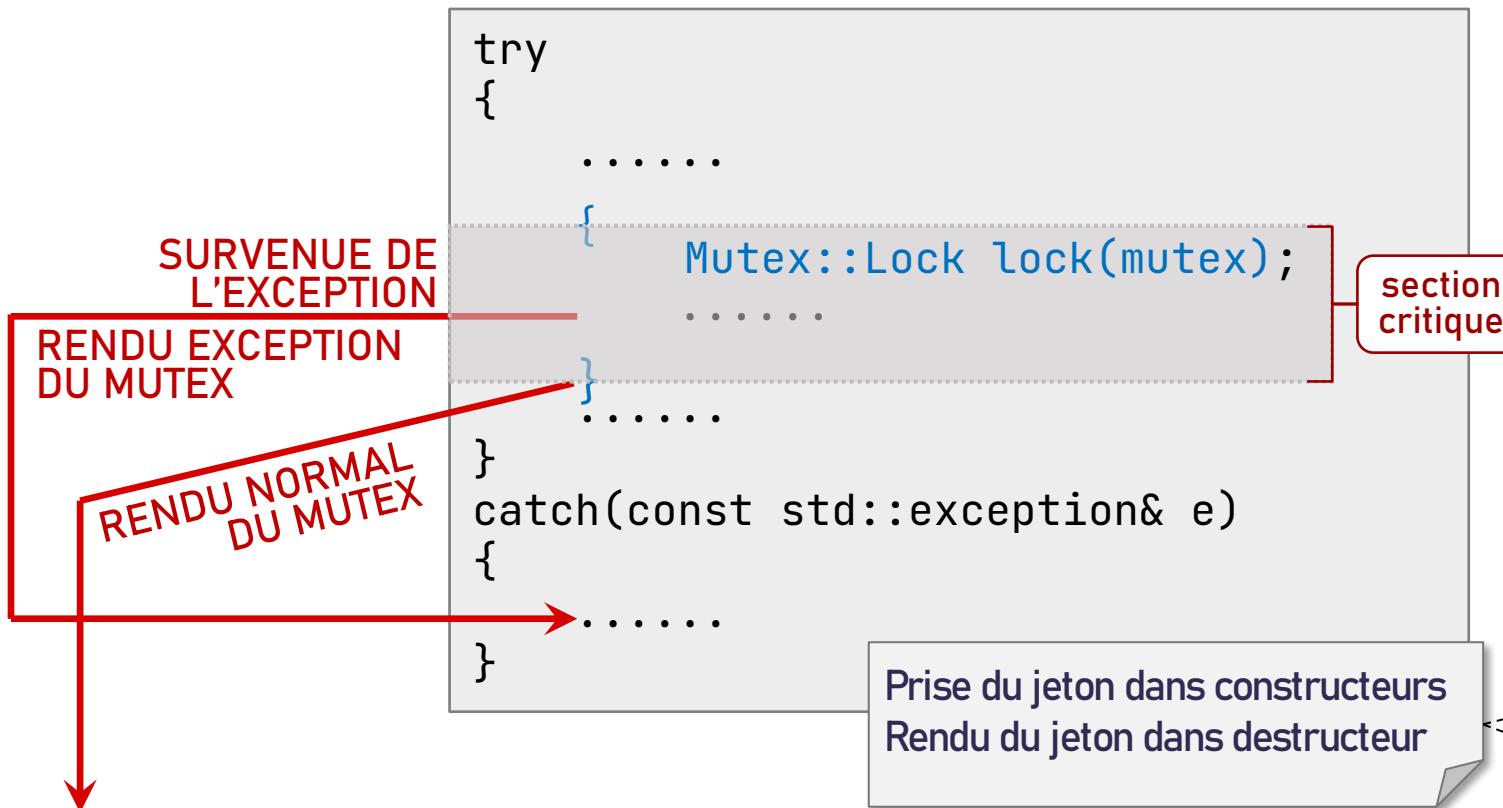
```
-posixId : pthread_mutex_t  
+Mutex(isInversionSafe : bool)  
+~Mutex()  
-lock()  
-lock(timeout_ms : long) : bool  
-unlock()
```

mutex ↑ 1

Mutex::Lock

```
+Lock(m : Mutex&)  
+Lock(m : Mutex&, timeout_ms : long)  
+~Lock()
```

Le problème du rendu de mutex lors des exceptions



Appel bloquant si mutex vide

Mutex

```
-posixId : pthread_mutex_t  
+Mutex(isInversionSafe : bool)  
+~Mutex()  
-lock()  
-lock(timeout_ms : long) : bool  
-unlock()
```

mutex ↑ 1

Mutex::Lock

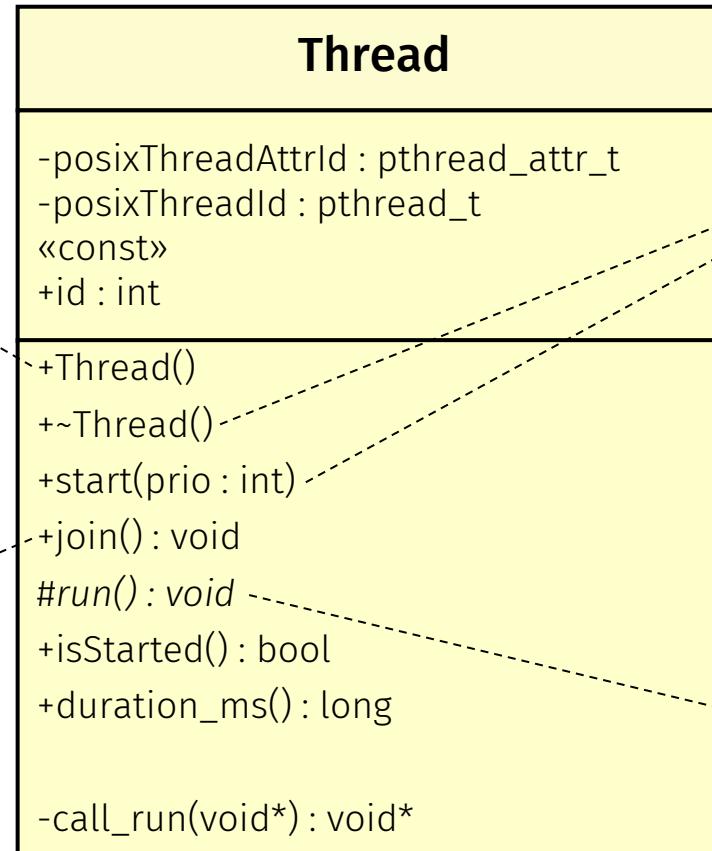
```
+Lock(m : Mutex&)  
+Lock(m : Mutex&, timeout_ms : long)  
+~Lock()
```

■ Resource Acquisition Is Initialization

- **encapsulate each resource into a class, where:**
 - the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
 - the destructor releases the resource and never throws exceptions;
- **always use the resource via an instance of a RAII-class that either:**
 - has automatic storage duration or temporary lifetime itself, or
 - has lifetime that is bounded by the lifetime of an automatic or temporary object.

Classes with `open()/close()`, `lock()/unlock()`, or `init()/copyFrom()/destroy()` member functions are typical examples of *non-RAII classes*.

Construction d'un objet Thread
non encore démarré.



On peut redémarrer un objet Thread
une fois qu'il terminé son exécution.

On distingue la création
de l'objet Thread (constructeur)
de son démarrage (start).

Questions
Pourquoi la méthode Thread::run() doit être :
- protected ?
- virtuelle pure ?

■ L'architecture des classes doit imposer les contraintes suivantes :

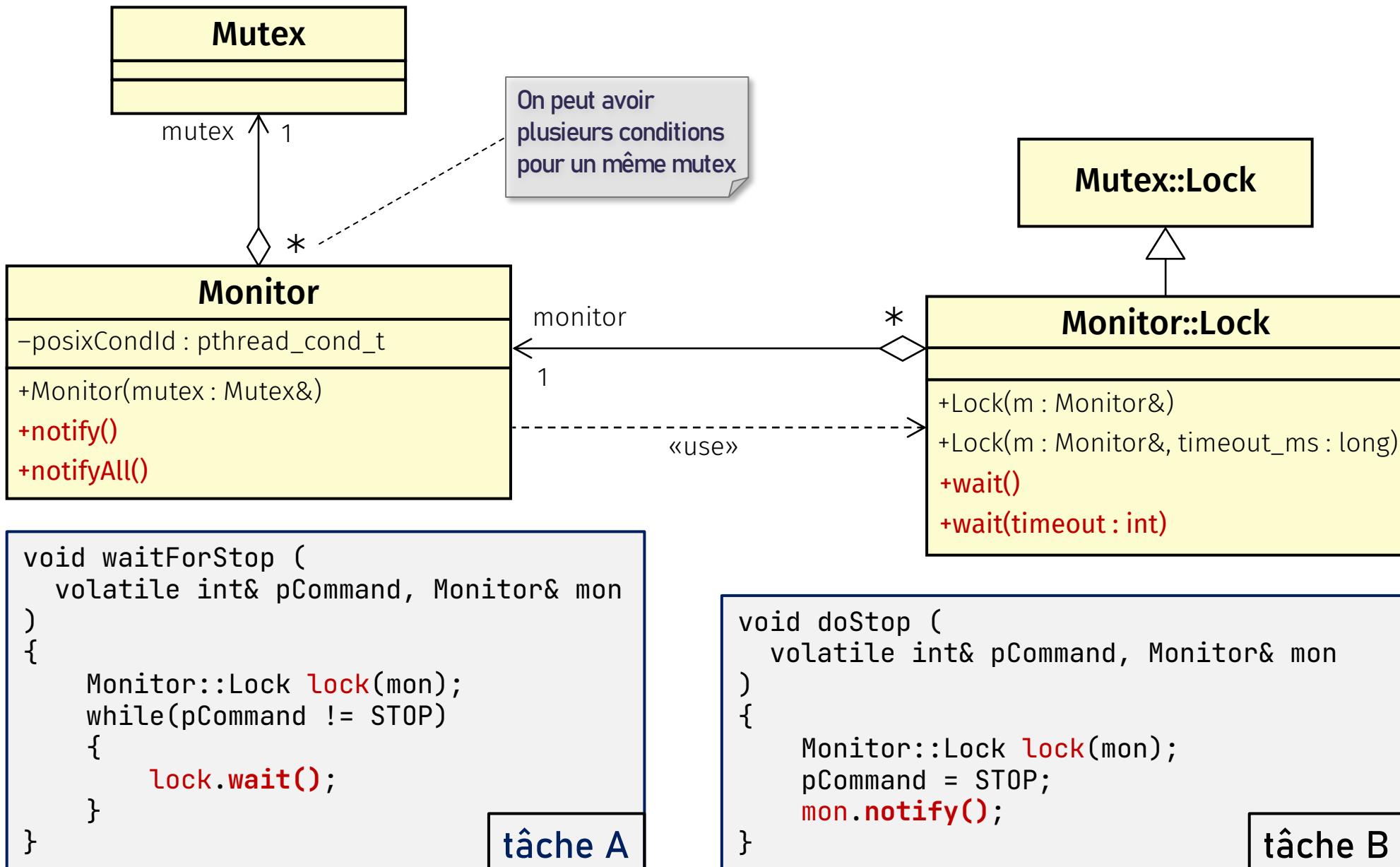
- un mutex et une condition doivent toujours être utilisés ensemble.
- le couple {mutex,condition} doit être partagé entre toutes les tâches qui en ont besoin.
- on ne doit pas faire un wait() sur une condition si le mutex n'a pas été verrouillé au préalable.
- on peut faire un notify() sur une condition sans avoir verrouillé le mutex.

```
void waitForStop (
    volatile int& pCommand, Monitor& mon
)
{
    Monitor::Lock lock(mon);
    while(pCommand != STOP)
    {
        lock.wait();
    }
}
```

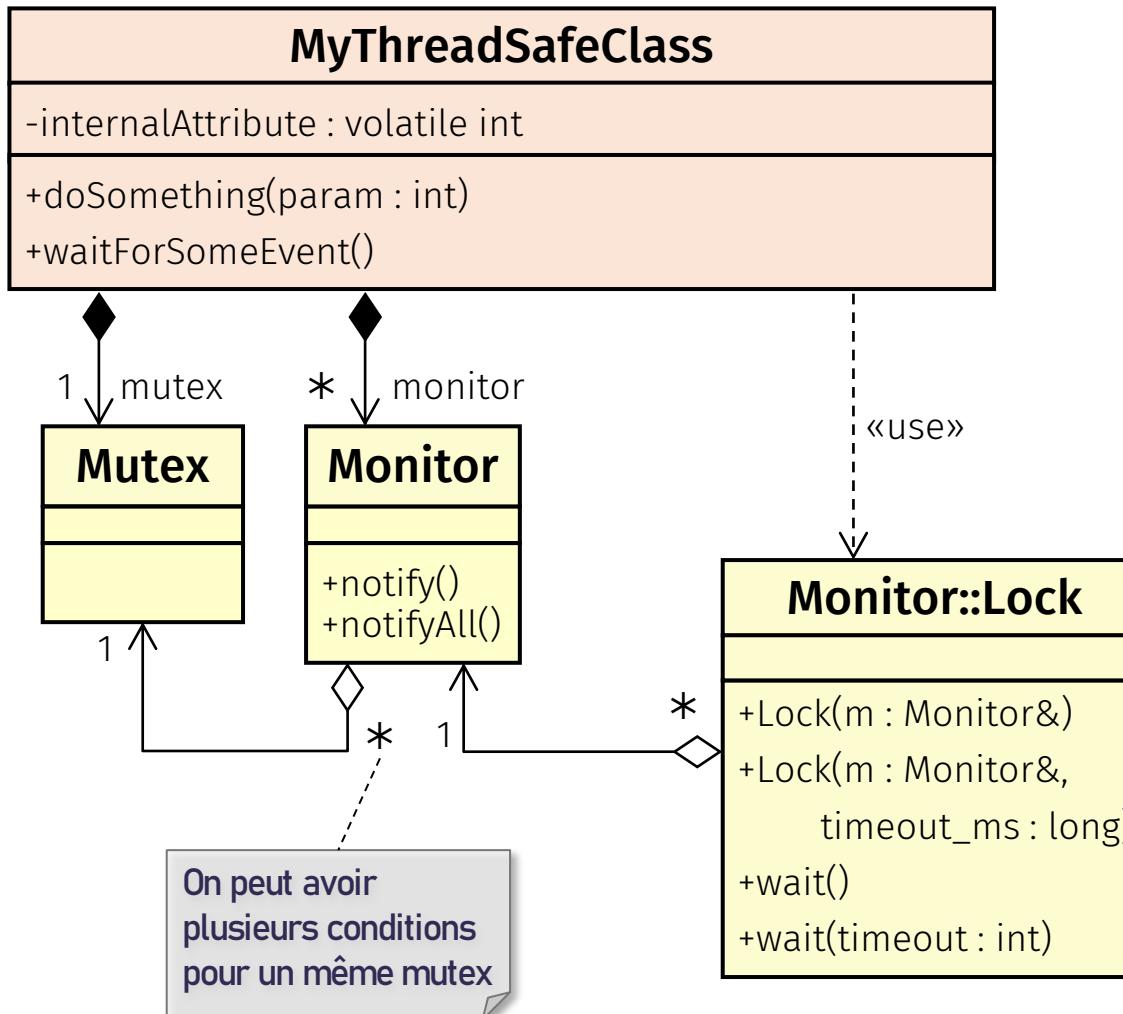
tâche A

```
void doStop (
    volatile int& pCommand, Monitor& mon
)
{
    Monitor::Lock lock(mon);
    pCommand = STOP;
    mon.notify();
}
```

tâche B



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



```

void MyThreadSafeClass::doSomething(int param)
{
    Monitor::Lock lock(monitor[2]);
    internalAttribute += param;
    monitor[2].notifyAll();
}

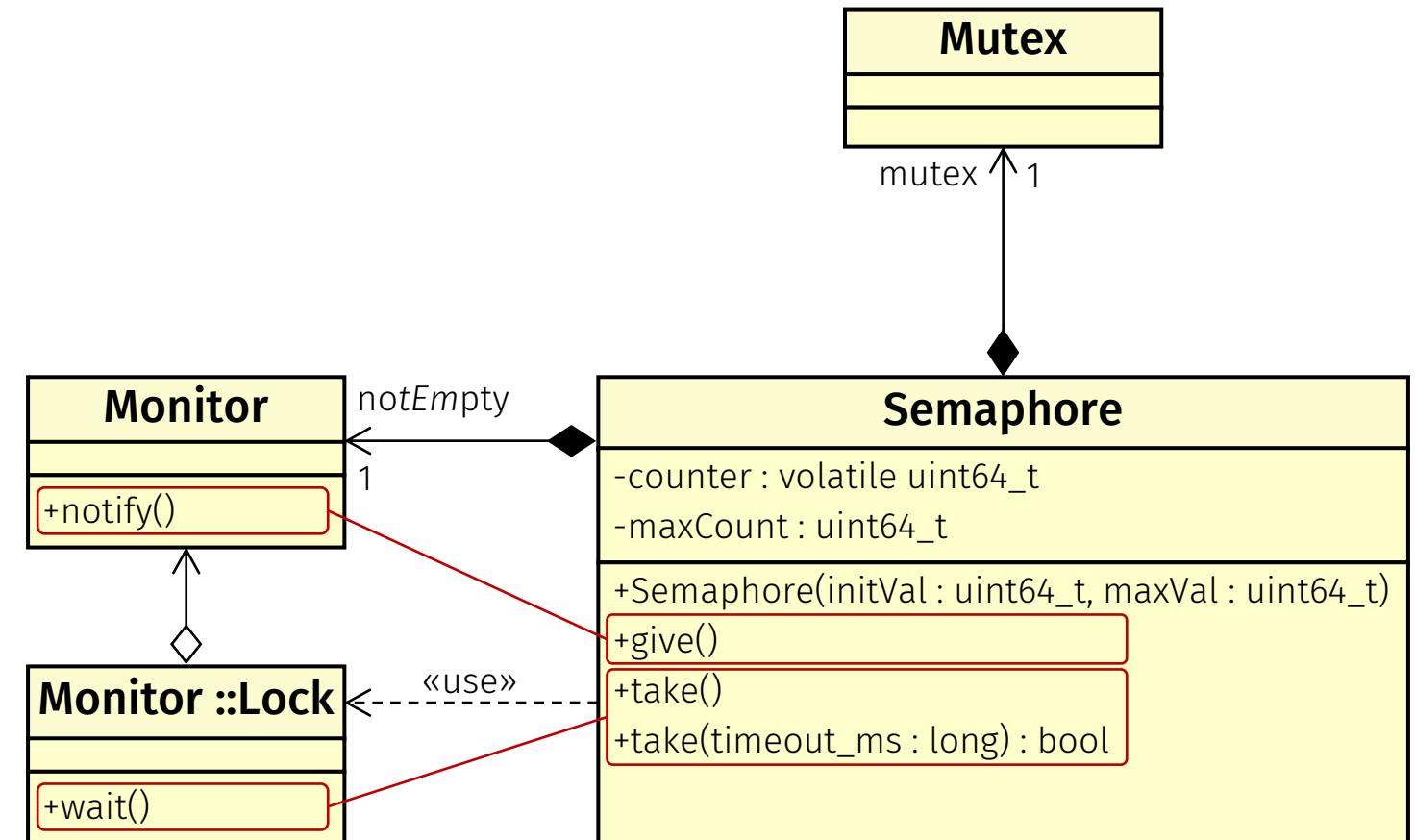
```

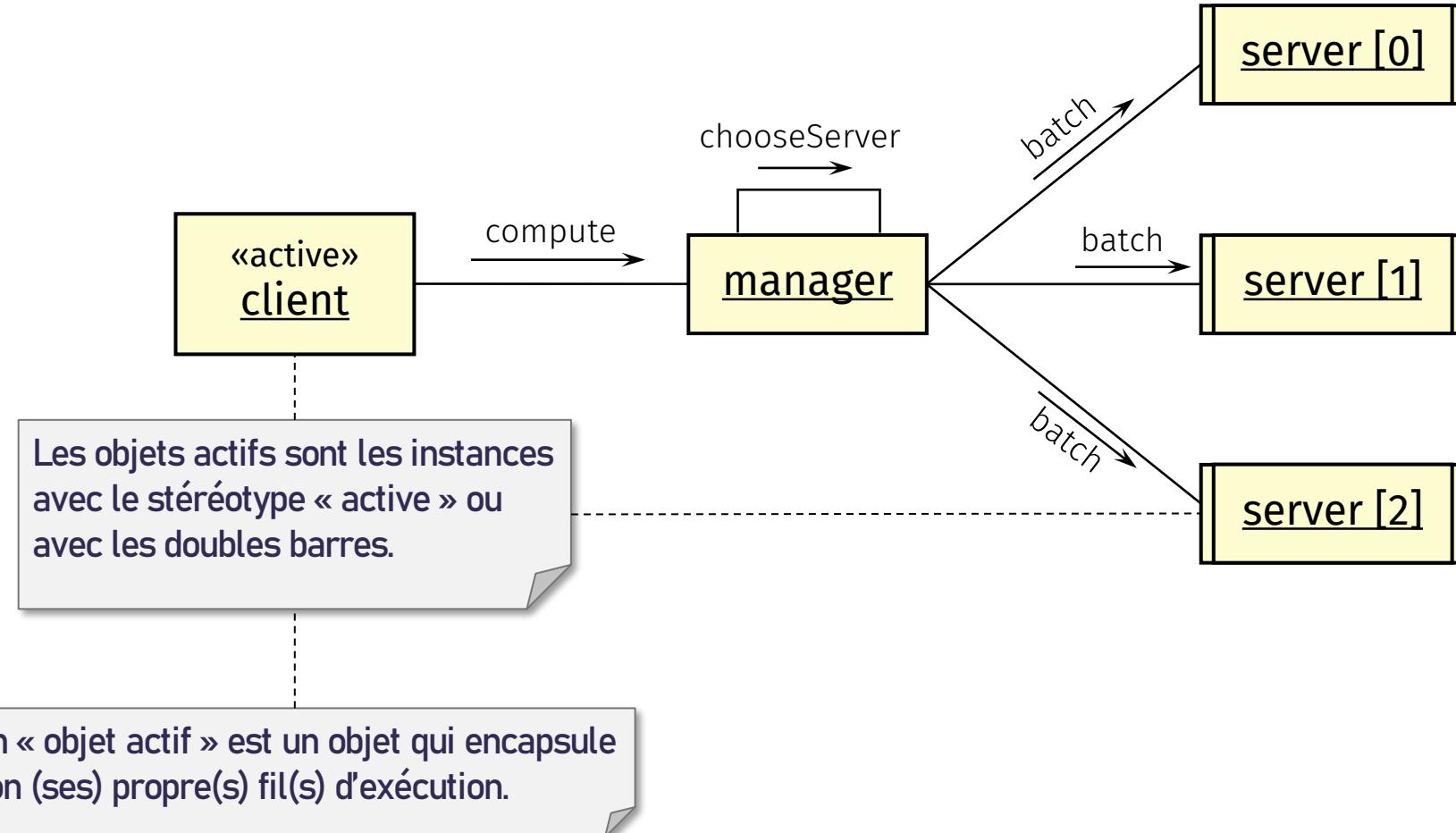
```

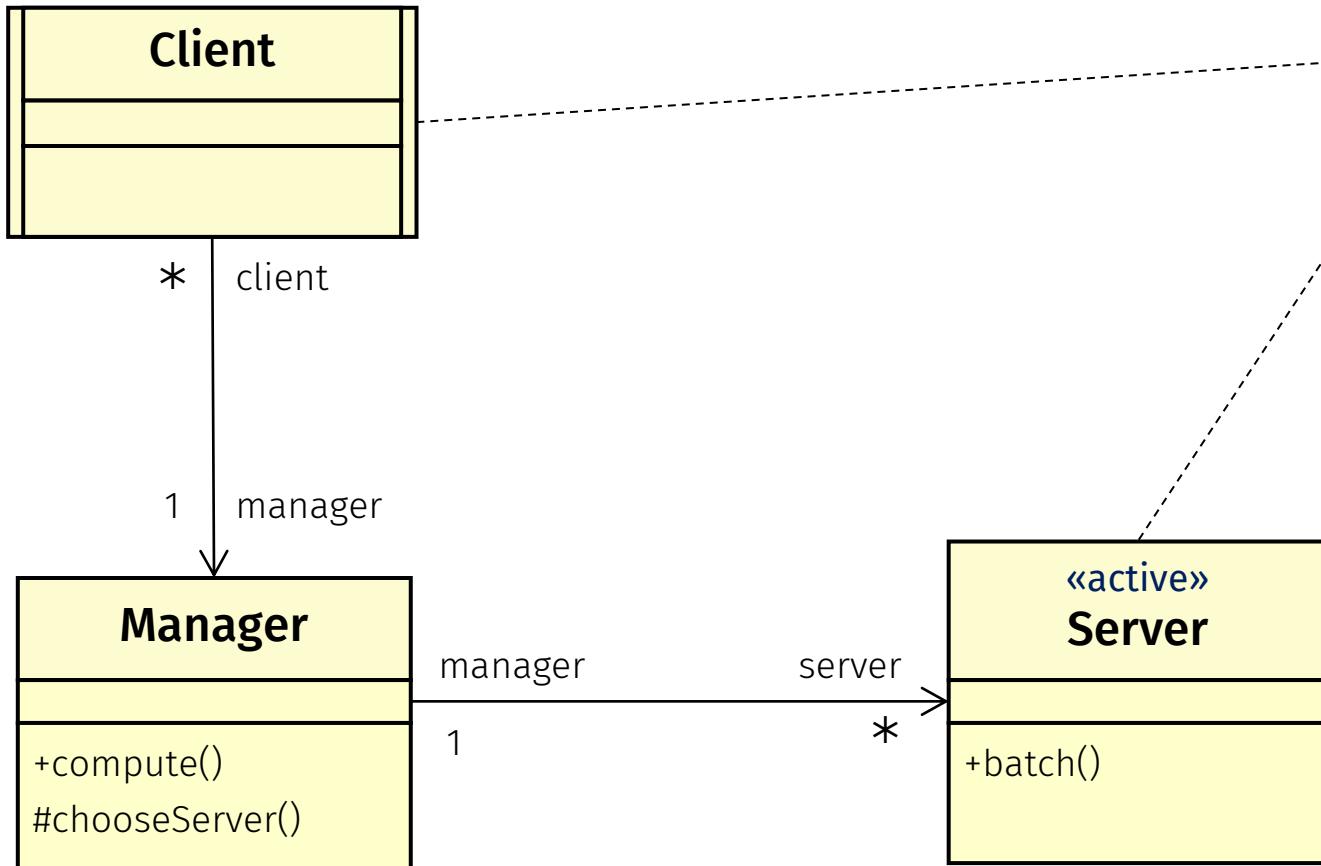
void MyThreadSafeClass::waitForSomeEvent()
{
    Monitor::Lock lock(monitor[2]);
    while(internalAttribute != 42)
    {
        lock.wait();
    }
}

```

- Un sémaforo 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'une condition







Les classes actives sont les classes avec le stéréotype « active » ou avec les doubles barres.

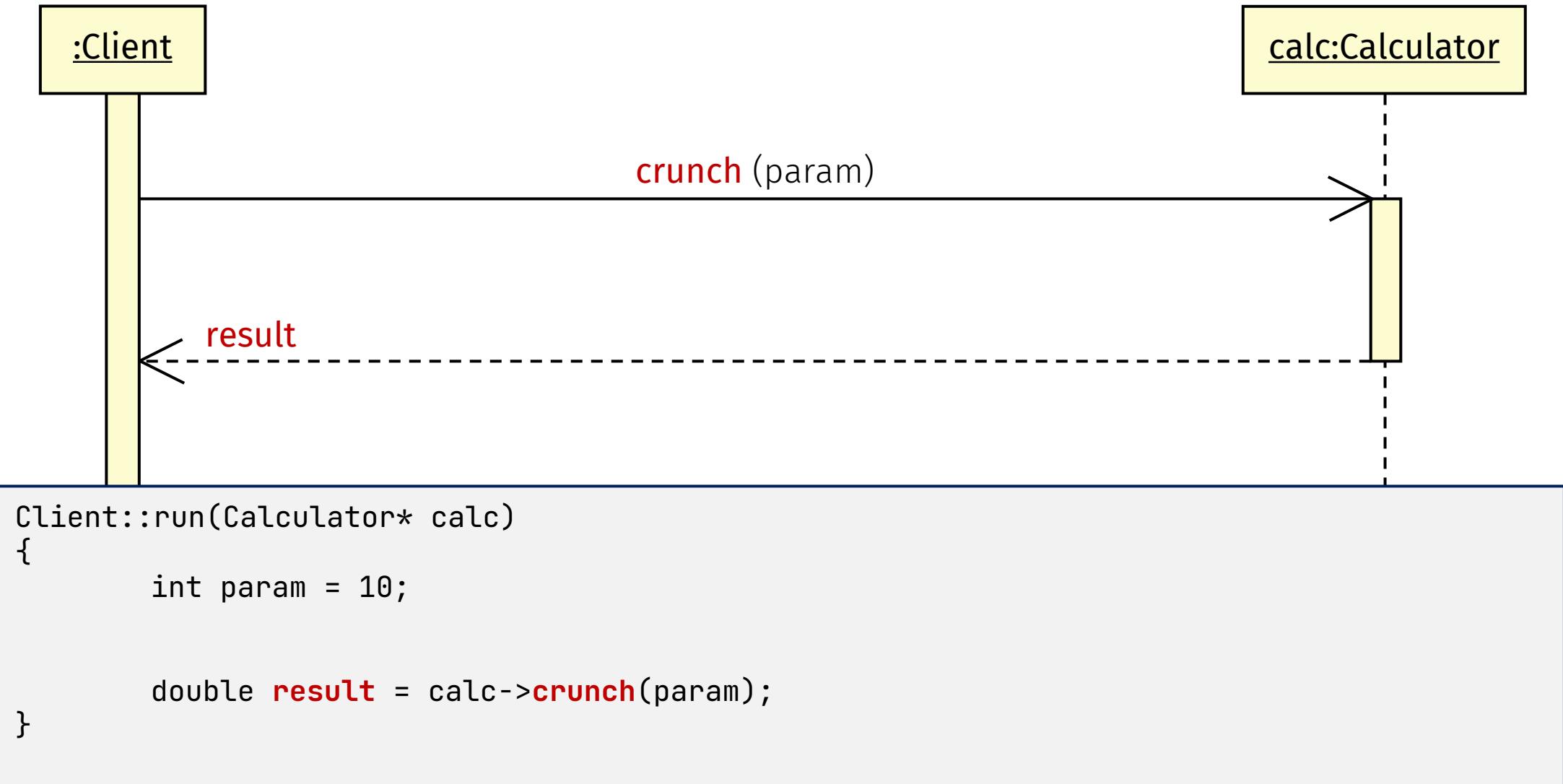
UML : aucune hypothèse sur le modèle d'exécution ni l'implémentation d'un objet actif.

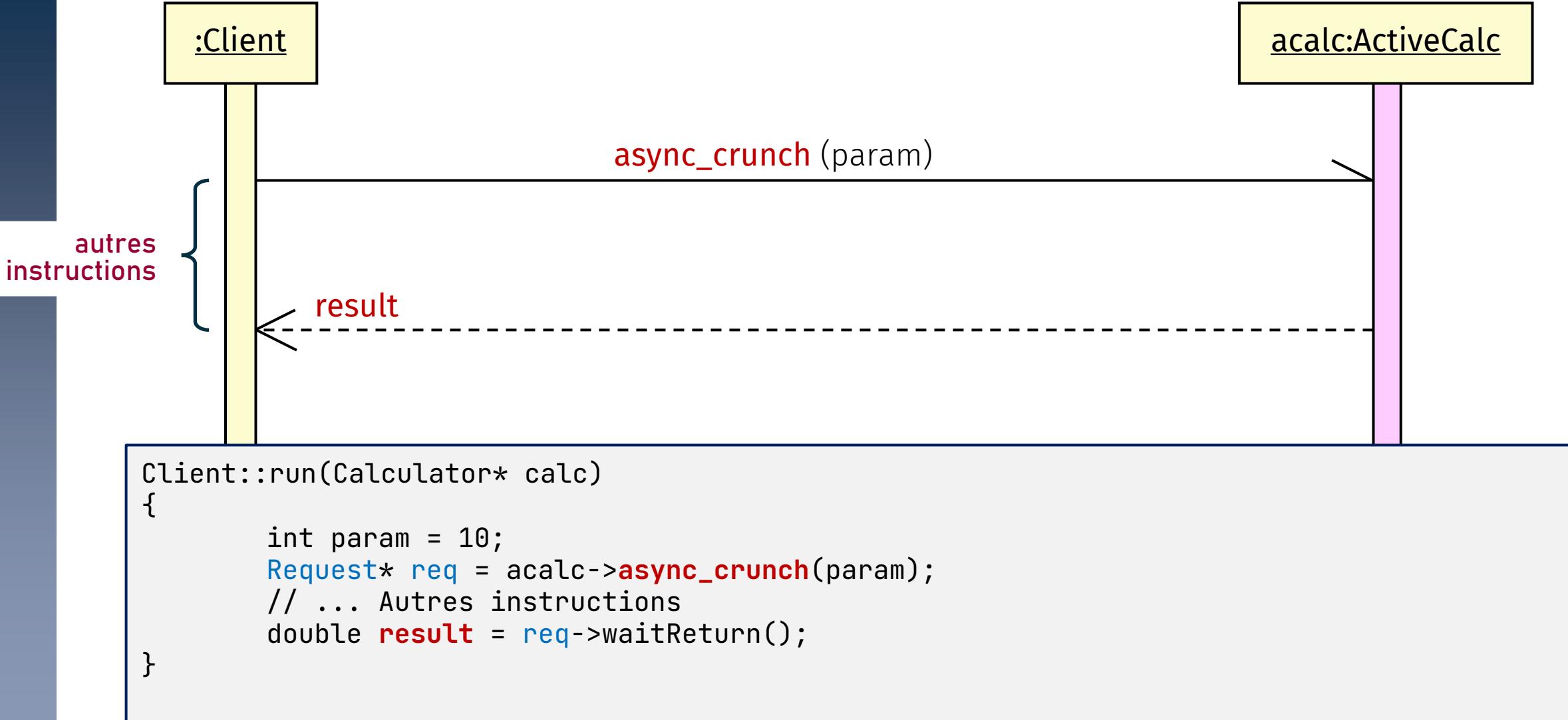
- Fusion entre objet et tâche

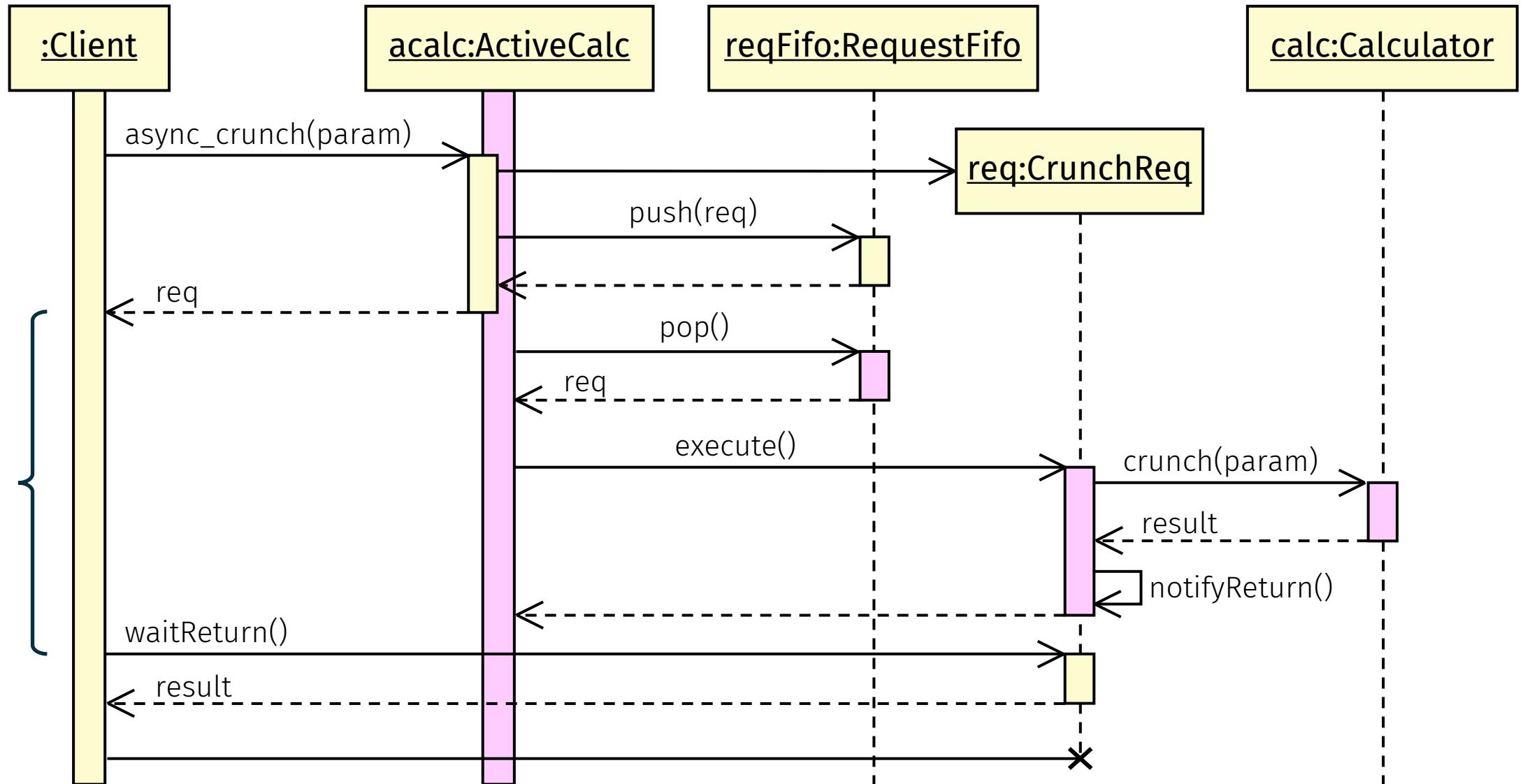
- Modèle d'exécution
 - monotâche
 - multitâche (une par opération)
 - Appel d'opération asynchrone
 - Appel d'opération à distance

- En pratique:

- La classe dérive d'une classe « Thread »
(ou implémente une interface ad hoc comme l'interface Runnable en Java)
 - La classe met en œuvre une file d'attente de requêtes d'exécution







```
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
    return req;                                // Transmission au client
}

void ActiveCalc::run() {
    while(true) {
        CrunchReq* req = reqFifo.pop();          // Réception de la requête
        req->execute();                         // Exécution de la requête
    }
}

double CrunchReq::waitReturn() {
    returnSema.take();                         // attente fin d'exécution du calcul (sémaphore)
    return result;                            // renvoi du résultat de calcul à l'appelant
}

void CrunchReq::execute() {
    result = calc->crunch(param);           // exécution effective du calcul
    returnSema.give();                      // notification de la fin de calcul (sémaphore)
}
```

