LOG 6306: Patrons pour la compréhension de programme

Foutse Khomh

Patrons pour la conception d'applications distribuées

ÉCOLE POLYTECHNIQUE MONTRÉAL

Département de génie informatique et de génie logiciel
École Polytechnique de Montréal

© Khomh, 2010 - 2018

Plan de la séance

- **Rappels de la séance précédente**
- Présentation des patrons d'architecture distribuées (suite)
- Résume de la séance

Plan de la séance

- Rappels de la séance précédente
- **Présentation des patrons d'architecture distribuées (suite)**
- Résume de la séance

Conception d’applications distribuées

- Patrons pour la synchronisation
  - *Scoped Locking*
  - *Double-checked Locking Optimization*
  - *Strategized Locking*
  - *Thread-Safe Interface*
- Patrons pour la configuration d’accès simultanés
  - *Active Object*
  - *Monitor Object*
  - *Half-sync/Half-async*
  - *Leader/Follower*
  - *Thread-specific Storage*

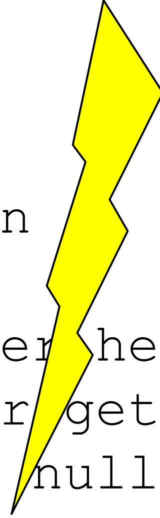
Patrons pour la synchronisation

- *Scoped Locking*
  - Assure qu'un jeton (lock) soit obtenu en entrant un bloc de contrôle (scope) et soit rendu automatiquement en sortant
- *Double-checked Locking Optimization*
  - Réduit les couts de verrouillages en évaluant d'abord une condition de verrouillage de façon non sure
- *Strategized Locking*
  - Stratégie de verrouillage des composants réutilisables
- *Thread-safe Interface*
  - Assure que les appels de méthodes intra-composants évitent les inter-blocages (deadlocks)

Patrons pour la synchronisation

- Le point focal des patrons suivant… le patron de conception Singleton

Exemple « courant »



```
// Single-threaded version
class Helper {
	private static Helper helper = null;
	public static Helper getHelper() {
		if (helper == null) {
			helper = new Helper();
		}
		return helper;
	}
	// Other functions and members...
}
```

Voir http://en.wikipedia.org/wiki/Double-checked_locking

Scoped Locking

- Contexte
  - Un programme concurrent contenant des données partagées qui sont accédées concurremment par plusieurs fils d'exécution
- Problème
  - Les jetons doivent toujours être obtenus et rendus correctement quand le flot de contrôle entre et quitte une section critique

Voir http://www.cs.wustl.edu/~schmidt/PDF/ScopedLocking.pdf

Scoped Locking

- Problème (suite)
  - Il est difficile de suivre tous les points de sorti du flot de contrôle à cause des *return*, *break*, *continue* et des exceptions non-attrapées

- Note
  - Différents langages de programmation amènent à différentes solutions

Scoped Locking for C++

- Solution pour C++
  - Définir une classe de garde dont
    - Le constructeur obtient le jeton
    - Le destructeur rend le jeton automatiquement

- Implantation

```
Thread_Mutex lock_;
...
Thread_Mutex_Guard guard (lock_);
```

Scoped Locking for C++

- Bénéfices
  - Augmentation de la robustesse
    - Empêche d'oublier de rendre le jeton
  - Réduction de l'effort de maintenance
    - Une seule implantation
    - Possibilité de sous-classes

Scoped Locking for C++

- Limitations
  - Possibilité d’un inter-blocage en cas d’appels de méthodes récursifs
    - Cf. *Thread-safe Interface*
  - Impossibilité d’implantation
    - Utilise les destructeurs de C++
    - Impossible en Java !
  - Avertissements intempestifs du compilateur
    - La variable locale *gard* n’est pas utilisée

Scoped Locking for C++

- Utilisations connues
  - *Adaptive Communication Environment* (ACE)
    - Ce patron est utilisé extensivement dans ce kit de programmation
  - Threads.h++
    - Cette bibliothèque défini un ensemble de classes de gardes modélisées suivant ce patron
  - Java !
    - Le langage Java définit le concept de *synchronized block* qui implémente cet idiome

Scoped Locking for Java

- Solution pour Java
  - Le langage introduit le mot-clé *synchronized* dont l'implantation est basée sur ce patron

```
// Multi-threaded but expansive version
class Foo {
      private Helper helper = null;
      public synchronized Helper getHelper() {
            if (helper == null) {
                  helper = new Helper();
            }
            return helper;
      …
```

Scoped Locking for Java

- Limitations
  - Le cout d’un appel à une méthode « synchronisée » peut être 100× celui à une méthode « normale »

Double-checked Locking Optimization

- Contexte
  – Un programme concurrent contenant des données partagées qui sont accédées concurremment par plusieurs fils d’exécution

Double-checked Locking Optimization

- Problèmes
  - En C++ : relations entre ordre d'exécution des instructions, séquence de points d'exécution, optimisation des compilateurs et du matériel
  - En Java : le cout d'un appel à une méthode « synchronisée » peut être 100× celui à une méthode « normale »

Double-checked Locking Optimization for Java

- Solution pour Java
  - Combinaison de *synchronized* et *volatile*
    - Synchroniser les méthodes et instructions pour empêcher deux fils d'exécution d'accéder à la même méthode / bloc d'instructions en même temps
    - Rendre *volatile* les variables qui sont lues/modifiées directement, sans qu'un cache ne soit maintenu dans les fils d'exécution ; leur lecture et écriture est « comme » dans un bloc synchronisé

Voir http://en.wikipedia.org/wiki/Double-checked_locking

Double-checked Locking Optimization for Java

- Solution (partielle et incorrecte) pour Java

```
// Multi-threaded but incorrect version
class Foo {
      private Helper helper = null;
      public Helper getHelper() {
            if (helper == null) {
                  synchronized(this) {
                        if (helper == null) {
                              helper = new Helper();
                        }
                  }
            }
            return helper;
      …
```

Double-checked Locking Optimization for Java

- Solution (complète et correcte) pour Java

```
// Multi-threaded and correct version
class Foo {
      private volatile Helper helper = null;
      public Helper getHelper() {
            if (helper == null) {
                  synchronized(this) {
                        if (helper == null) {
                              helper = new Helper();
                        }
                  }
            }
            return helper;
      …
```

Double-checked Locking Optimization for Java

- Bénéfices
  - Réduit les surcouts lors de l'initialisation fainéante et d'accès multiples

Double-checked Locking Optimization for C++

- Solution pour C++
  - pInstance = new Singleton;

  - Allouer une zone mémoire pour l'objet Singleton
  - Construire l'objet dans la zone mémoire allouée
  - Faire pointer pInstance sur la zone mémoire

Voir http://www.drdobbs.com/184405726

Double-checked Locking Optimization for C++

```
class Singleton {
        public:
                static volatile Singleton* volatile instance();
        private:
                // one more volatile added
                static volatile Singleton* volatile pInstance;
};

        …

        // from the implementation file
        volatile Singleton* volatile Singleton::pInstance = 0;
        volatile Singleton* volatile Singleton::instance() {
        if (pInstance == 0) {
                Lock lock;
                if (pInstance == 0) {
                        // one more volatile added
                        volatile Singleton* volatile temp = new volatile Singleton;
                        pInstance = temp;
                }
        }
        return pInstance;
}
```

Voir http://www.drdobbs.com/184405726

Double-checked Locking Optimization for C++

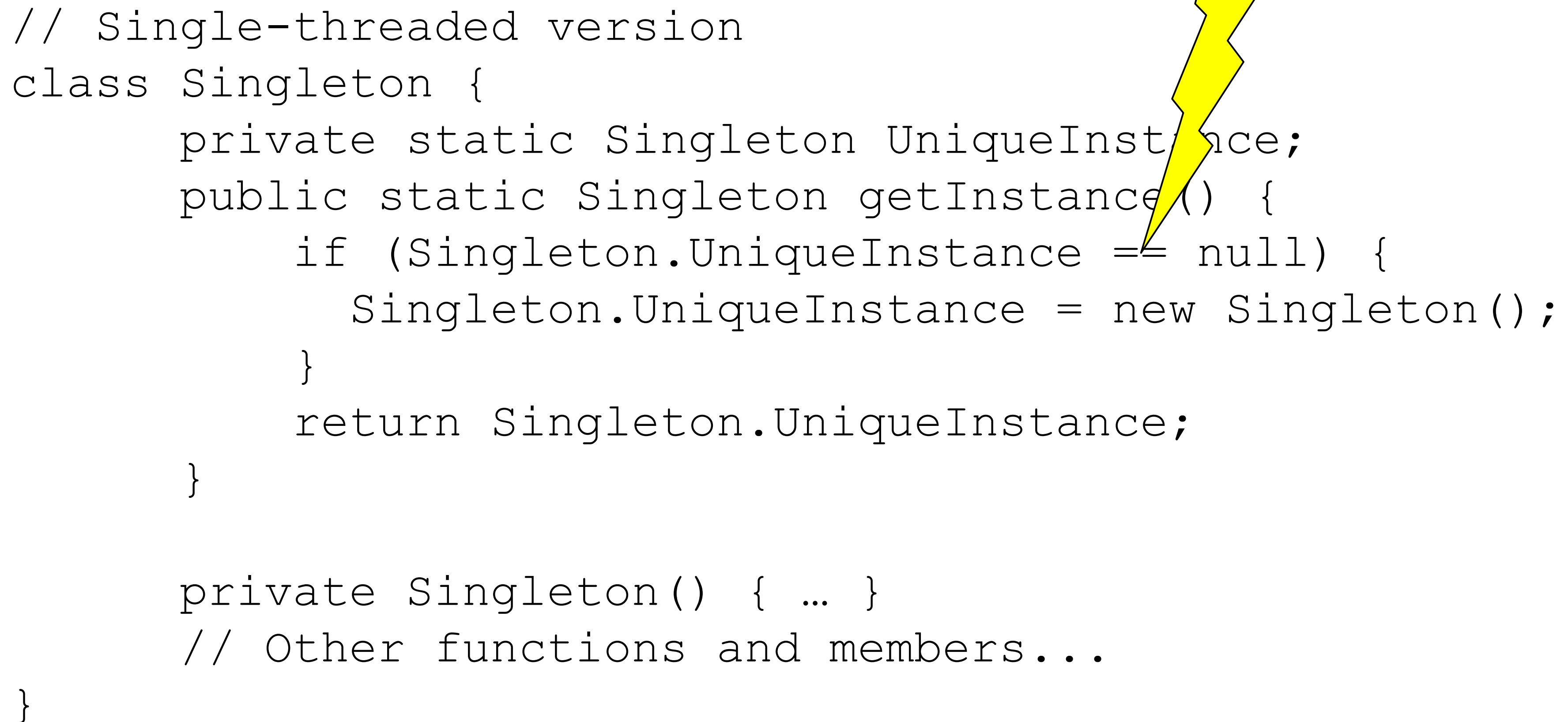
- Bénéfices
  - Prévient plus ou moins le reordonnancement des instructions par le compilateur
  - Réduit les surcoûts lors de l'initialisation fainéante et d'accès multiples
- Limitations
  - Multi-processeurs
  - Possibilité d'erreur lies à la portabilité sur des plate-forme avec pointeur non-atomique
  - Optimisations
    - Un Singleton par fil d'exécution ?

Scoped Locking, Double-checked Locking Optimization and Java

- Il est très important de bien connaître le langage de programmation que vous utilisez, surtout s'il utilise une machine virtuelle

- Par example, le mécanisme de chargement des classes peut simplifier (grandement) l'implantation du Singleton et en garantir la correction dans un contexte avec plusieurs fils d'exécution !

Scoped Locking, Double-checked Locking Optimization and Java



```
// Single-threaded version
class Singleton {
      private static Singleton UniqueInstance;
      public static Singleton getInstance() {
            if (Singleton.UniqueInstance == null) {
              Singleton.UniqueInstance = new Singleton();
            }
            return Singleton.UniqueInstance;
      }

      private Singleton() { … }
      // Other functions and members...
}
```

Voir http://en.wikipedia.org/wiki/Singleton_pattern

Scoped Locking, Double-checked Locking Optimization and Java

- Sans initialisation fainéante

```
// Multi-threaded version
class Singleton {
	// Could also be final for further optimisation
	private static Singleton UniqueInstance = new Singleton();
	public static Singleton getInstance() {
		return Singleton.UniqueInstance;
	}

	private Singleton() { … }
	// Other functions and members...
}
```

Rôle du chargeur de classes ?

Scoped Locking, Double-checked Locking Optimization and Java

- Avec initialisation fainéante, mais lente !

```
// Multi-threaded version
class Singleton {
	// Cannot be final
	private static Singleton UniqueInstance = null;
	public static synchronized Singleton getInstance() {
		if (Singleton.UniqueInstance == null) {
			Singleton.UniqueInstance = new Singleton();
		}
		return Singleton.UniqueInstance;
	}

	private Singleton() { … }
	// Other functions and members...
}
```

Scoped Locking, Double-checked Locking Optimization and Java

- Avec initialisation fainéante, sans *synchronized*

```
// Multi-threaded version
class Singleton {
        private static class SingletonHolder {
                public static final Singleton UniqueInstance
                        = new Singleton();
        }
        public static Si  gleton getInstance() {
            return Singl  onHolder.UniqueInstance;
        }

        private Singleto      … }
        // Other functio      members...
}
```

Rôle du chargeur de classes ?

Scoped Locking, Double-checked Locking Optimization and Java

- Sans initialisation fainéante en Java 5+

```
// Multi-threaded version
enum Singleton {
        UNIQUE_INSTANCE;
}
```

Synchronisation en Java et communication entre fils d'execution

- La lecture/écriture dans une variable booléenne est atomique
- Combien de temps va s'exécuter le « backgroundThread » ?

```
public class StopThread {
    private static boolean stopRequested;

    public static void main(final String[] args)
        throws InterruptedException {

        final Thread backgroundThread =
            new Thread(new Runnable() {

            public void run() {
                int i = 0;
                while (! StopThread.stopRequested) {
                    i++;
                }
            }
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        StopThread.stopRequested = true;
    }
}
```

Synchronisation en Java et communication entre fils d'execution

- Sur un JVM « server »

```
while (!stopRequested) {
    i++;
}
```

devient :

```
if (!stopRequested) {
    while (true) {
        i++;
    }
}
```

- Pour éviter cette « liveness failure » :

```
public class StopThread {
    private static boolean stopRequested;

    private static synchronized void requestStop() {
        StopThread.stopRequested = true;
    }
    private static synchronized boolean stopRequested() {
        return StopThread.stopRequested;
    }
    public static void main(final String[] args)
        throws InterruptedException {

        final Thread backgroundThread =
            new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while (!StopThread.stopRequested())
                    i++;
            }
        });
        backgroundThread.start();

        ...
```

Strategized Locking

- Contexte
  – Un programme dans lequel des composants réutilisables doivent être accéder concurremment par plus d'un fil d'exécution
- Problème
  – Difficile d'ajuster le mécanisme de concurrence quand il est codé « en dur »
  – Difficile de maintenir et d'améliorer les implantations

Voir http://www.cs.wustl.edu/~schmidt/PDF/locking-patterns.pdf

Strategized Locking

- Solutions
  - Externaliser les mécanismes d’acquisition et de relâche du jeton

Strategized Locking

- Implantation
  - Définir l'interface du composant et son implantation (sans synchronisation)
  - Choisir la stratégie de synchronisation, à l'aide
    - Du polymorphisme (C++ et Java)
    - Des types paramétrés
  - Définir une famille de stratégies
    - Penser en particulier au verrou « null »

Stratégized Locking for Java

- Implantation
  - « À la main »
  - Avec l'interface Lock
    - ReentrantLock
    - ReentrantReadWriteLock .ReadLock
    - ReentrantReadWriteLock .WriteLock

```
// Multi-threaded version with Lock
class Foo {
   private volatile Helper helper = null;
      public synchronized Helper getHelper() {
         if (helper == null) {
            final Lock l = …
            l.lock();
            try {
               if (helper == null) {
                  helper = new Helper();
               }
            }
            finally {
               l.unlock();
            }
         }
         return helper;
         …
```

Strategized Locking for C++

- Implantation polymorphique
  - « À la main » avec le patron Pont (Bridge)

```
class File_Cache {
   public:
      // Constructor
      File_Cache (Lock lock): lock_(lock) {}
      // A method.
      const char *find(const char *pathname) {
         // Use the Scoped Locking idiom to
         // acquire and release the <lock_>
         Guard<Lock> guard(lock_);
         // Implement the find() method.
      }
      // ...
   private:
      // The polymorphic locking object.
      Lock lock_;
      // Other File_Cache members and methods
      // ...
```

```
class Lockable {
   public:
   // Acquire the lock.
   virtual int acquire(void) = 0;
   // Release the lock.
   virtual int release(void) = 0;
   // ...
};

class Lock {
   public:
      // Constructor stores a reference
      // to the base class.
      Lock (Lockable &l): lock_(l) {};
      // Acquire the lock by forwarding to
      // the polymorphic acquire() method.
      int acquire (void) { lock_.acquire(); }
      // Release the lock by forwarding to the
      // polymorphic release() method.
      int release (void) { lock_.release(); }
   private:
      // Maintain a reference to the lock.
      Lockable &lock_;
};

class Thread_Mutex_Lockable : public Lockable {
   public:
      …
```

Strategized Locking for C++

- Implantation polymorphique
  - La classe Lock qui fait le pont entre un client et un objet Lockable est importante pour utiliser Lock comme un objet et donc pouvoir utiliser le patron *Scoped Locking*

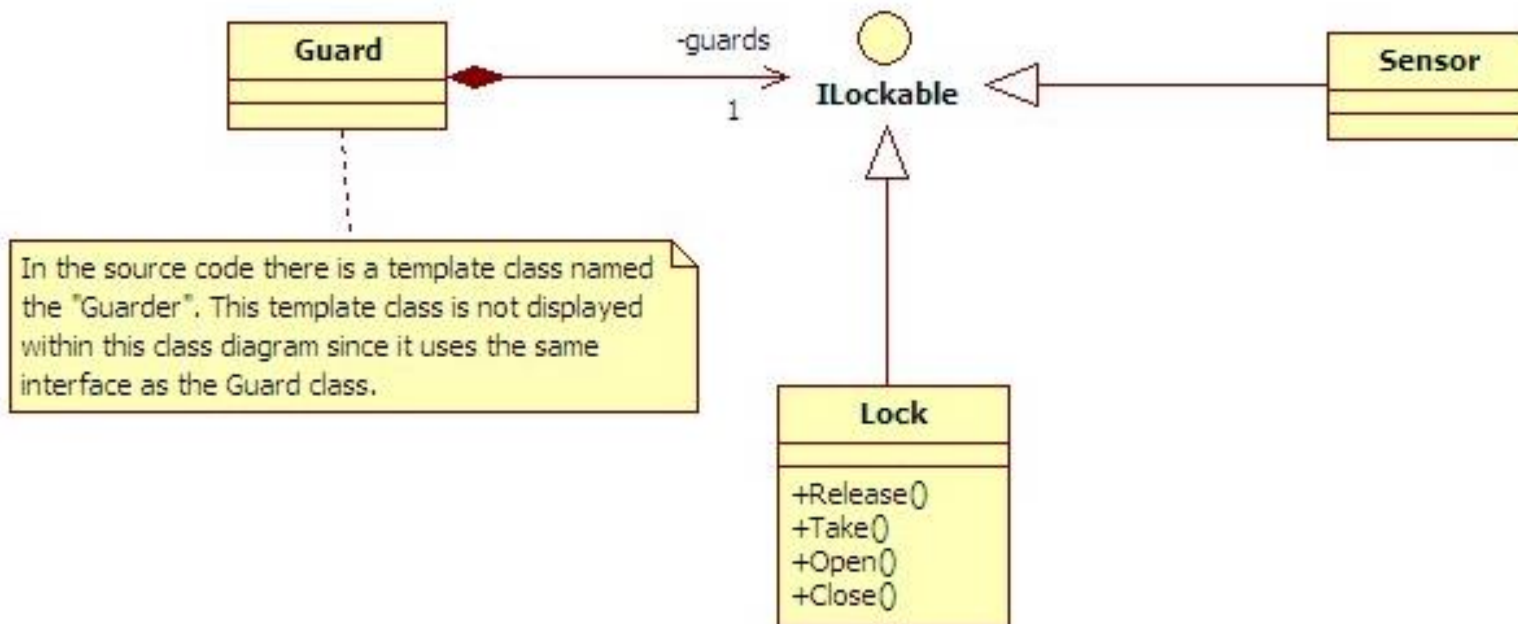
Strategized Locking for C++

- Implantation avec types paramétrés
  - Similaire à l'approche polymorphique
  - À préférer si le mécanisme de verrouillage est connu à la compilation

```
template <class LOCK>
class File_Cache {
   public:
      const char *find(const char *pathname) {
         // Use the Scoped Locking idiom to
         // acquire and release the <lock_>
         Guard<LOCK> guard(lock_);
         // Implement the find() method.
      }
      // ...
   private:
      // The polymorphic locking object.
      LOCK lock_;
      // Other File_Cache members and methods
      // ...
```

Strategized Locking Pattern

Exemple: classe Sensor



- `Lock`: The wrapper for the Windows `Mutex` calls
- `Guard`: Responsible for taking and releasing the `Mutex`
- `ILockable`: Responsible for providing the interface for locking shared data within lockable objects

http://www.codeproject.com/KB/architecture/StrategizedLocking.aspx

Strategized Locking

- Bénéfices
  - Meilleur flexibilité et possibilité d'ajustement pour des raisons des performances
  - Effort de maintenance réduit
- Limitations
  - La version types paramétrés expose la stratégies de verrouillage au code client
  - Certains compilateurs gèrent mal les types paramétrés

Thread-safe Interface

- Contexte
  – Composants dans un programme avec de multiples fils d'exécution et des appels de méthodes intra- et inter-composants
- Problèmes
  – Éviter les inter-blocages dus à des appels de méthodes intra- et inter-composants
  – Éviter les surcoûts de synchronisation pour les appels de méthodes intra-composants

Thread-safe Interface

- Solution
  - Vérification de la synchronisation pour toutes les méthodes publique de l'interface
  - Confiance en la sureté vis-à-vis des fils d'exécution des implantations des méthodes privées

Thread-safe Interface

- Implantation
  - Séparer méthodes publiques et synchronisation d’une part et implantation d’autre part

```
template <class LOCK>
class File_Cache {
  public:
    // Return a pointer to the memory-mapped
    // file associated with <pathname>, adding
    // it to the cache if it does not exist.
    const char *find(const char *pathname) {
      // Use the Scoped Locking idiom to
      // acquire and release the <lock_>.
      Guard<LOCK> guard(lock_);
      return find_i(pathname);
    }
    // Add <pathname> to the file cache.
    void bind(const char *pathname) {
      // Use the Scoped Locking idiom to
      // acquire and release the <lock_>.
      Guard<LOCK> guard(lock_);
      bind_i(pathname);
    }
  private:
    // The strategized locking object.
    LOCK lock_;
    // The following implementation methods do not
    // acquire or release <lock_> and perform their
    // work without calling any interface methods.
    const char *find_i (const char *pathname) {
      const char *file_pointer =
        check_cache_i (pathname);
      ...
```

Thread-safe Interface

- Utilisations connues
  - java.util.Hashtable
    - put(Object key, Object value)
      - Protégé pour la synchronisation
    - rehash()
      - Pas protégé pour la synchronisation
      - Le pourrait mais réduction des performances

        (Il n'y aurait pas d'inter-blocage parce que les verrous de Java sont réentrants)

Thread-safe Interface

- Bénéfices
  - Robustesse améliorée
  - Performances améliorées
- Limitations
  - Indirections additionnelles
  - Plus de méthodes

Patrons pour la configuration d’accès simultanés

- *Active Object*
  - Découpler invocation et exécution des méthodes
- *Monitor Object*
  - Assurer qu’une seule méthode sur un objet à un moment donné dans le temps
- *Half-sync/Half-async*
  - Séparer les accès synchrones et asynchrones aux entrées–sorties
- *Leader/Follower*
  - Partager des sources d’événements entre de multiples fils d’exécution
- *Thread-specific Storage*
  - Permettre le partage de données entre fils d’exécution sans encourir les surcouts de synchronisation

Active Object

- Contexte
  – Un client invoque des méthodes sur des objets qui s’exécutent dans différents fils d’exécution
- Problème
  – Méthodes invoquées sur une objet ne devraient pas bloquer les autres méthodes du même objet
  – L’accès synchronisé aux objets partagés devrait être simple
  – Le programme devrait utiliser au mieux les possibilités de parallèlisation de la plateforme

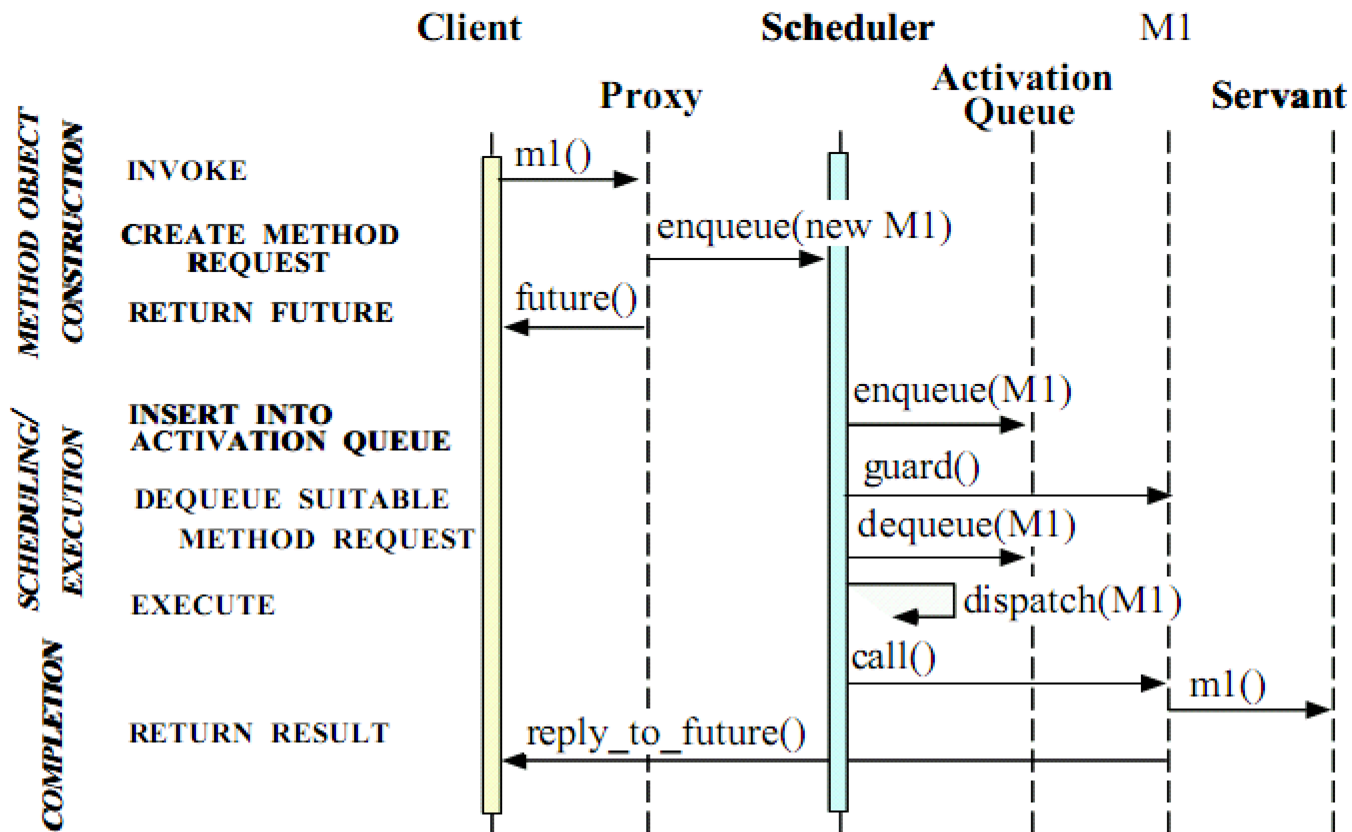
Voir http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf

Active Object

- Solution
  - *Method Request* : passer le contexte à un mandataire (*Proxy*) et garder les appels
  - *Activation Queue* : garder un tampon limité de *Method Requests* et découpler le fil d'exécution du client de celui du servant
  - *Scheduler* : décider quelle *Method Request* exécuter dans un fil d'exécution séparé
  - *Future* : mémoriser et donner accès aux résultats de *Method Requests*

Active Object

- Implantation



Active Object

- Utilisations connues
  - CORBA ORBs
  - ACE
    - Des implémentations réutilisables de *Method Request*, *Activation Queue* et *Future* sont disponibles
  - Siemens MedCom

Active Object

- Bénéfices
  – Amélioration et simplification de la concurrence
  – Utilisation transparente du parallélisme
  – Différence dans l'ordre d' invocation et d'exécution des méthodes
- Limitations
  – Surcouts en performance
  – Débogage plus compliqué

Monitor Object

- Contexte
  - Programme dans lequel de multiples fils d'exécution accèdent à des objets simultanément
- Problèmes
  - Les méthodes d'un objet sont synchronisées
  - Les objets, par leurs clients, sont responsables de la synchronisation de leurs méthodes
  - Les objets exécutent leurs méthodes de façon coopérative

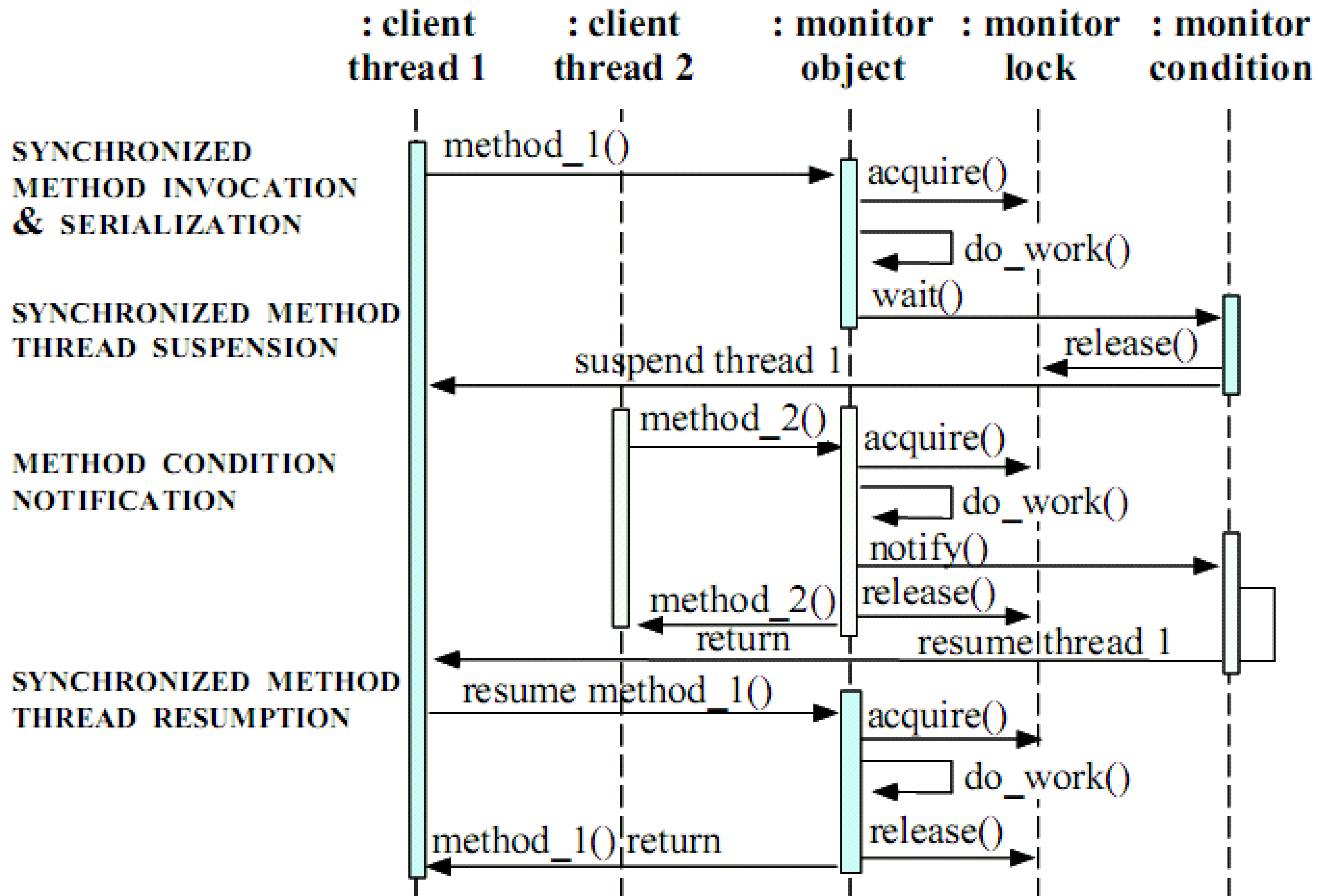
Voir http://www.cs.wustl.edu/~schmidt/PDF/monitor.pdf

Monitor Object

- Solution
  - Synchroniser toutes les méthodes de l’objet
  - Assurer qu’une seule méthode synchronisée s’exécute sur un objet à la fois

Monitor Object

- Implantation



Monitor Object

- Utilisations connues
  - Dijkstra/Hoare Monitors
    - La fonctionnalité monitors encapsule les fonctions et leurs variables internes dans un thread-safe module
  - Tout objets Java
    - wait()
    - notify()
    - notifyAll()
  - ACE Gateways

Monitor Object

- Bénéfices
  - Synchronisation simplifiée
  - Ordonnancement coopératifs de l'exécution des méthodes synchronisées
- Limitations
  - Couplage fort entre fonctionnalités et mécanismes de synchronisation d'un objet
  - « Lockout » en cas d'objets moniteurs imbriqués

Half-sync/Half-async

- Contexte
  - Synchronisation entre processus de bas et de haut niveau, par exemple dans un noyau Linux BSD lors de la réception de paquets depuis des ports d'entrée–sortie
- Problème
  - Besoin de simplifier l'implantation
  - Besoin de performances

Voir http://www.cs.wustl.edu/~schmidt/PDF/PLoP-95.pdf

Half-sync/Half-async

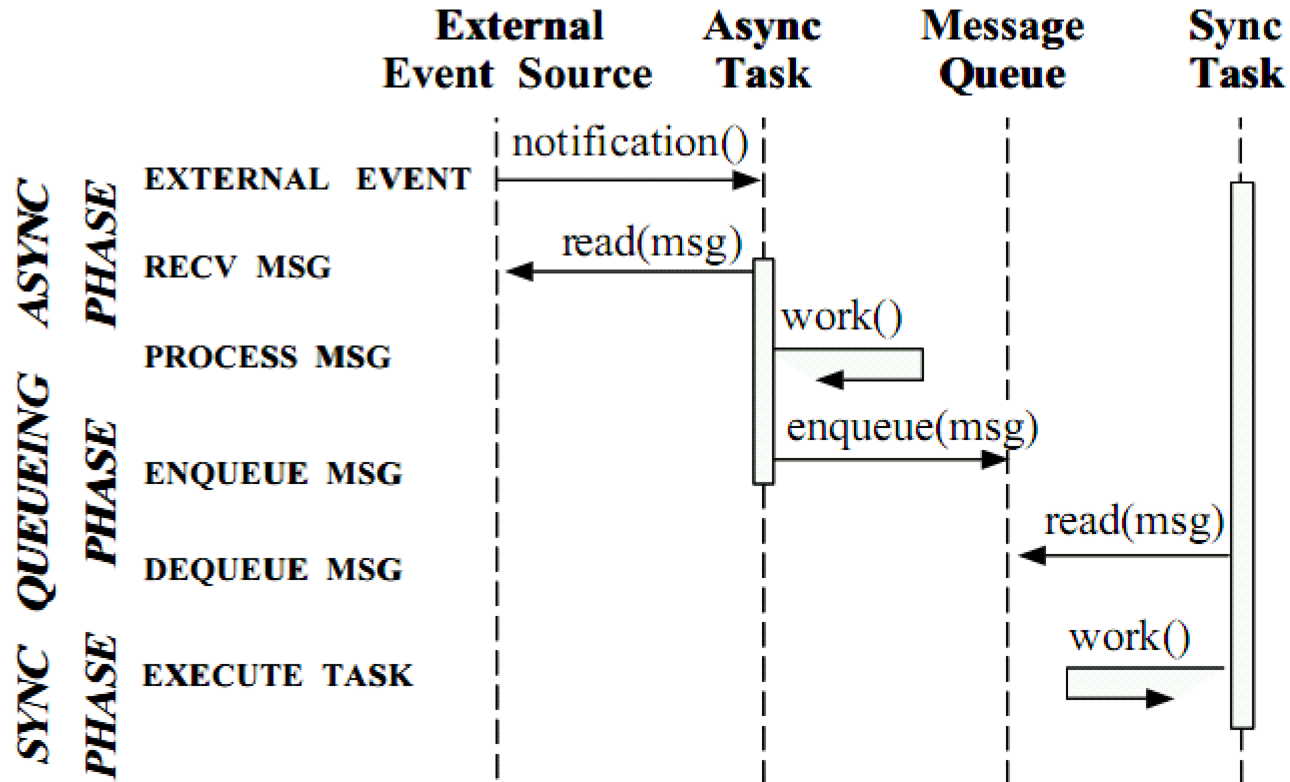
- Solution
  - Couche synchrone (e.g., processus clients)
    - Transfert des données de façon synchrone à la couche Queueing Layer
    - Active Objects qui peuvent « bloquer »
  - Queueing Layer (e.g., socket)
    - Synchronisation et tampon entre les couches synchrones et asynchrones
    - Événements d'entrée–sortie mis dans un tampon par les tâches asynchrones/synchrones

Half-sync/Half-async

- Solution
  - Couche asynchrone (e.g., noyau Linux BSD)
    - Gère les événements de bas niveau
    - Objets passifs qui ne peuvent pas bloquer
  - Sources d’événement (e.g., interface réseau)
    - Génèrent des événements reçus par la couche asynchrone

Half-sync/Half-async

- Implantation



Half-sync/Half-async

- Bénéfices
  - Simplification des tâches de haut niveau car programmation synchrones
  - Possibilité d’avoir des politiques de synchronisation propre à chaque couches
  - Communication entre couche localisée dans un seul point
  - Performances améliorées sur des systèmes avec plusieurs processeurs

Half-sync/Half-async

- Limitations
  - Surcout lors du passage d’une couche a une autre a cause de la copie des données et du changement de contexte
  - Entrées–sorties asynchrones ne sont pas disponibles pour les tâches de haut niveau

Leader/Follower

- Contexte
  - Un programme dans lequel des événements doivent être partagés et traites par de multiples fils d'exécution qui partagent ces événements
- Problème
  - Démultiplier événements et fil d'exécution
  - Minimiser le surcout du à la concurrence
  - Empêcher les situation de compétition (race conditions) sur les sources d'événements

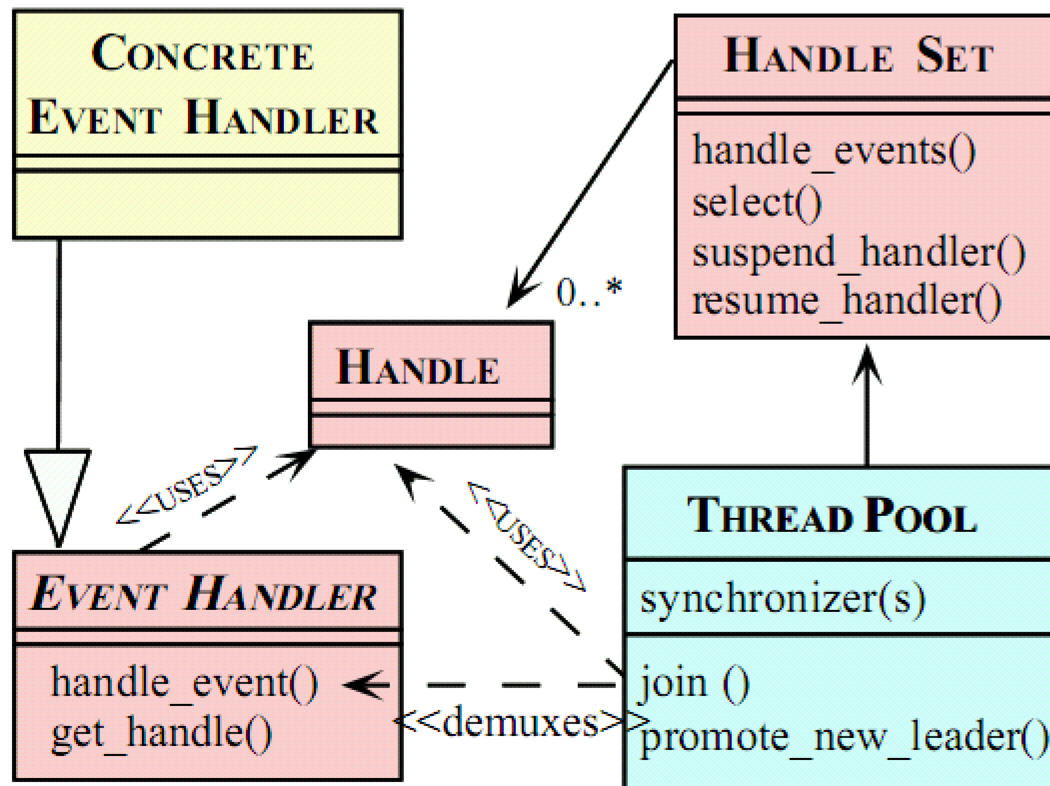
Voir http://www.cs.wustl.edu/~schmidt/PDF/lf-PLOPD.pdf

Leader/Follower

- Solution
  - Avoir un ensemble de fil d’exécution
  - À tout moment, un seul fil, le Leader, attend qu’un événement se produise dans un ensemble de sources
  - Quand un événement survient, le Leader promeut un nouveau fil et traite l’événement
  - Le nouveau Leader peut s’exécuter pendant que l’exécution du précédent se poursuit

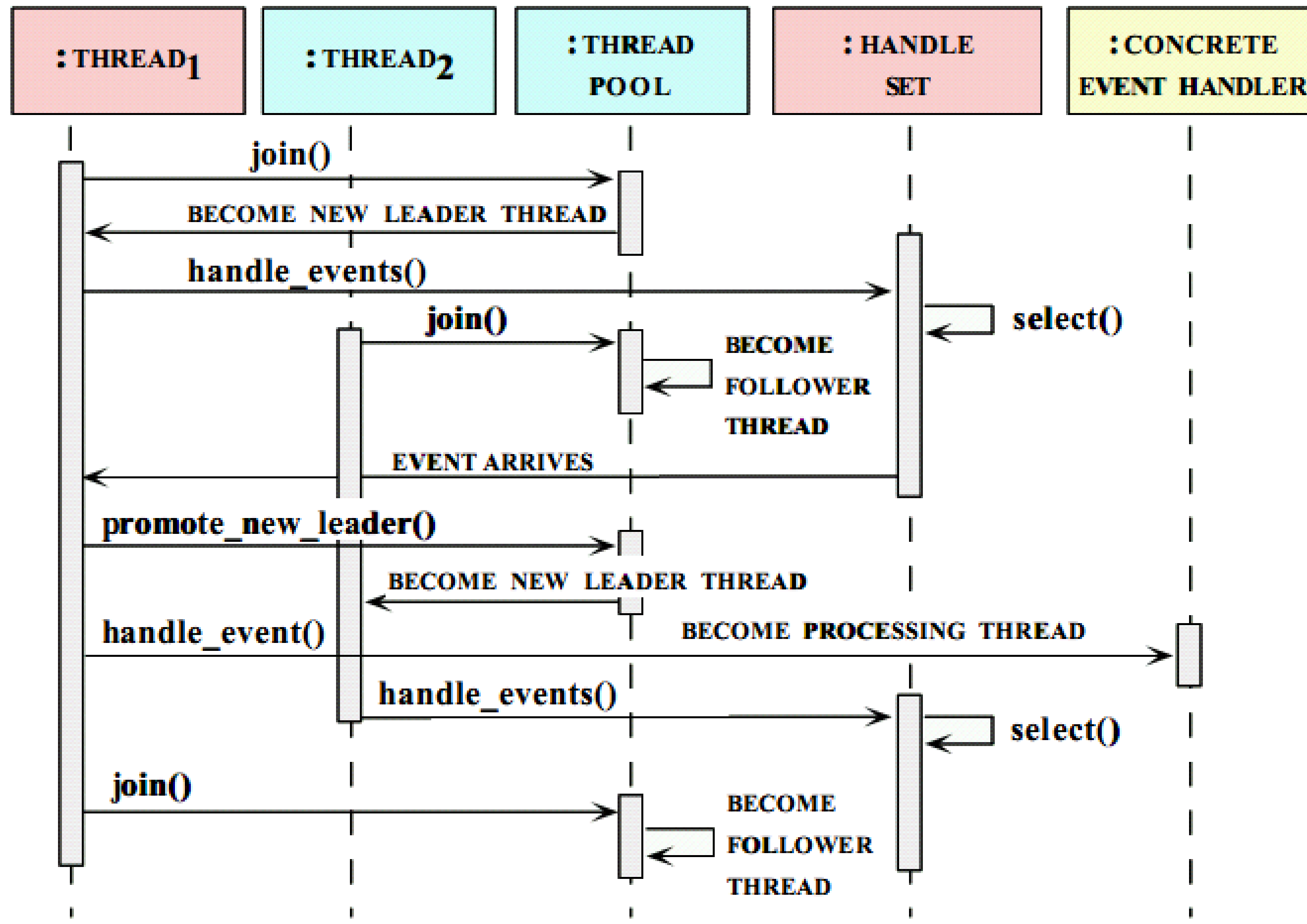
Leader/Follower

- Implantation



Leader/Follower

- Implantation



Leader/Follower

- Utilisations connues
  - CORBA ORBs
  - Serveur Web JAWS

  - … Arrêt de taxis à l’aéroport

Leader/Follower

- Bénéfices
  - Amélioration des performances
  - Simplification de la programmation
- Limitations
  - Complexification de l’implantation
  - Manque de flexibilité
  - Goulot d’étranglement au niveau des entrées–sorties réseaux

Thread-specific Storage

- Contexte
  – Un programme dans lequel de multiples fil d'exécution affecte une valeur a une variable globale qui est plus tard utilisée dans un test

- Problème
  – Un fil d'exécution affecte une valeur, un autre la change, le premier l'utilise dans un test
  – Éviter le surcoût du blocage de la variable

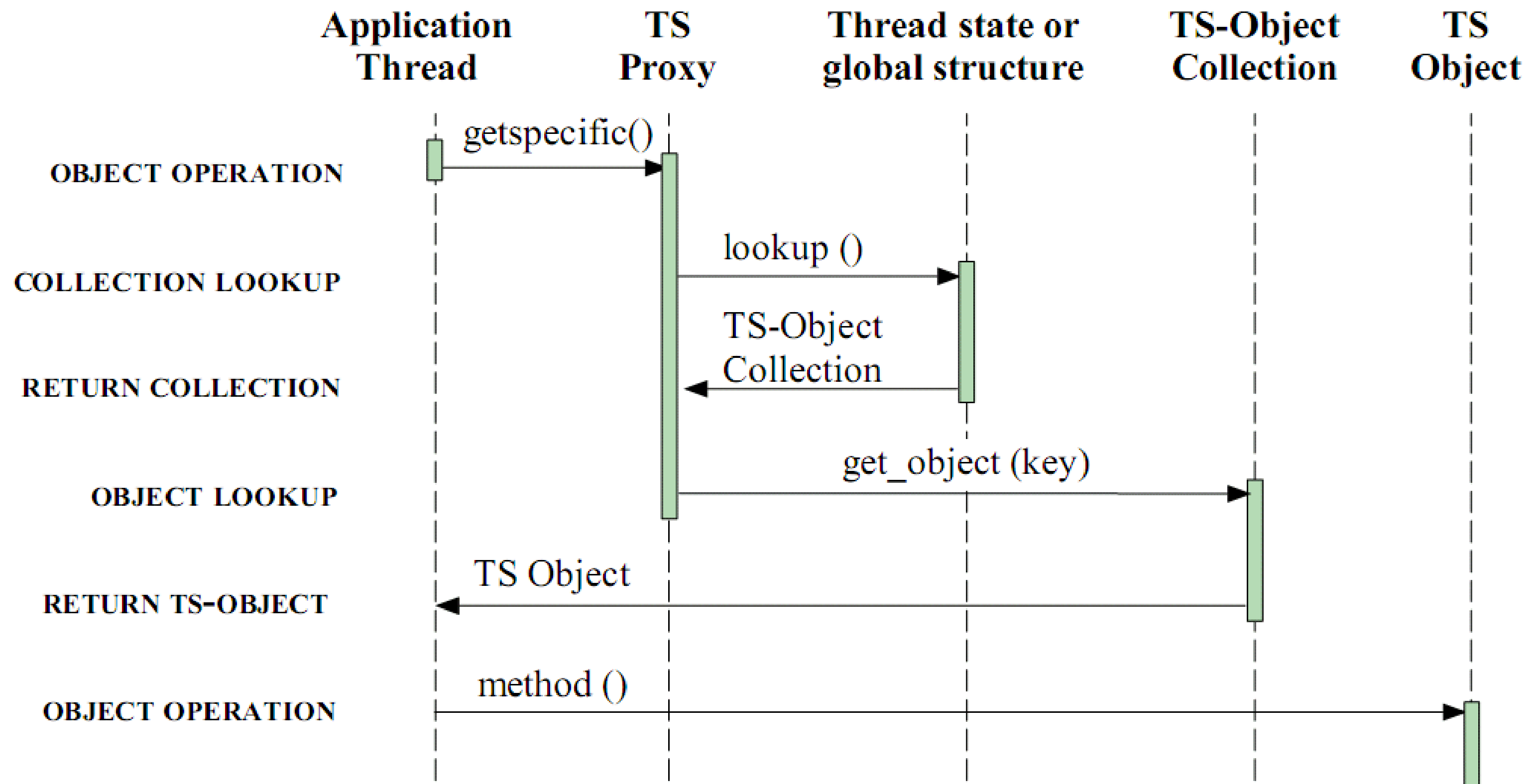
Voir http://www.cs.wustl.edu/~schmidt/PDF/TSS-pattern.pdf

Thread-specific Storage

- Solution
  - Un fil d'exécution du programme utilise des TS Object Proxies pour accéder au TS Objects qui sont stockés dans une collection TS Objects Collection propre a chaque fil
  - TS Object Proxy définit l'interface des TS Objects, un Proxy par type de TS Objects
  - TS Objects sont les instances particulières à un fil d'exécution d'un objet spécifique au fil
  - TS Objects Collection collecte les TS Objects

Thread-specific Storage

- Implantation



Thread-specific Storage

- Utilisations connues
  - Le mécanisme *errno* dans les systèmes d'exploitation supportant POSIX et l'API de fil d'exécution de Solaris
  - Les fenêtres dans Win32 appartiennent chacune à un fil d'exécution qui utilise une queue d'événements stockée dans un TSS

Thread-specific Storage

- Bénéfices
  - Efficacité
  - Simplicité d'utilisation
- Limitations
  - Encourage l'utilisation de variables globales
  - Rend plus difficile la compréhension

Plan de la séance

- Rappels de la séance précédente
- Présentation des patrons d’architecture distribuées (suite)
- **Résume de la séance**

Voir aussi…

- http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/locks/Lock.html
- http://www2.research.att.com/~bs/bs_faq2.html#finally