Corso di Programmazione Concorrente

Tecniche di Programmazione Thread-Safe

Valter Crescenzi

http://crescenzi.inf.uniroma3.it

Sommario

- Introduzione alle
  Tecniche di Programmazione Thread-Safe
- Tecnica degli Oggetti Immutabili
- Tecnica delle Classi Completamente Sincronizzate
- Tecniche di Confinamento per:
  - *Metodo*
  - *Thread*
  - *Oggetto*
  - *Gruppo*
- Tecniche di Programmazione Thread-Safe e Modello ad Attori

Tecniche di Programmazione Thread-Safe

- Elementi metodologici per la scrittura di codice Thread-Safe
  - Se spiegati con riferimento ad un linguaggio concreto come java: *tecniche di programmazione*
- Utili sia per progettare nuovi classi, sia per comprendere le scelte progettuali sottostanti alcune librerie thread-safe di ampio utilizzo.

  Ad esempio porzioni di:
  - `java.util.concurrent`
  - `javax.swing`
- Utili anche per analizzare interi paradigmi di programmazione (come il modello ad attori) con riferimento ai problemi della PC

Thread Safeness & Condizioni di Bernstein

- Dato un programma multithread, quali strutture dati bisogna proteggere per garantire la thread safeness?

  Tutte le strutture dati oggetto di accessi concorrenti che violano le condizioni di Bernstein

  in altre parole,

  le strutture dati oggetto di scritture concorrenti da parte di due o più thread

Tre Approcci per la Thread-Safeness

- Per evitare interferenze bisogna garantire che tutte le strutture dati siano thread-safe
- A tal fine, per ciascuna struttura dati bisogna invalidare il predicato

oggetto di scritture concorrenti

da parte di due o più thread

- Diverse alternative
  - evitare le scritture
  - evitare gli accessi concorrenti da parte di due o più thread
  - disciplinare gli accessi da parte dei thread

    (...con molte sfumature diverse>>)

Tecniche per la Thread-Safeness

- *Tecnica degli oggetti immutabili*
  - evitare completamente le scritture
- *Tecnica delle classi completamente sincronizzate*
  - disciplinare gli accessi nello spirito dei monitor *astratti*
- *Tecnica di confinamento per thread*
  - permettere ad un solo thread di accedere alla struttura dati
- *Tecnica di confinamento per oggetto/metodo/gruppo*
  - rendere la struttura accessibile ad un solo thread e nascosta a tutti gli altri (molte varianti: il thread è scelto da un *dominio*>>)

Tecnica degli Oggetti Immutabili

- Una tecnica spesso usata per prevenire interferenze è quella di evitare del tutto le scritture utilizzando solo oggetti immutabili
- Per ogni metodo che richiederebbe una scrittura nello stato dell'oggetto:
  - si crea una *copia* dell'oggetto sul quale il metodo è stato invocato
  - all'atto della creazione tale copia risulta non condivisa e pertanto sicura
  - le scritture necessarie per svolgere l'operazione richiesta si effettuano sulla copia e *non* sull'originale
  - la copia viene restituita come risultato del metodo

Oggetti Immutabili vs Referential-Transparency

- Una delle principali motivazioni alla base del recente ritorno di interesse per i linguaggi funzionali:
  - fanno uso di strutture dati immutabili (in quel contesto denominate “persistenti”) per ottenere la cosidetta proprietà della *referential-transparency*
  - facilitano la scrittura di codice thread-safe e quindi lo sfruttamento delle moderne architetture multi-core
  - i programmatori possono disinteressarsi di alcune delle problematiche della programmazione concorrente affrontate in questo corso
    - ovviamente *non sempre!...*
    - Ma in molti contesti ottengono ragionevoli speed-up senza quasi alcuno sforzo

```
class Fraction {                                  // Fragments
  protected long num;
  protected long den;
  public Fraction(long n, long d) {
    boolean sameSign = (n >= 0) == (d >= 0);
    long num = (n >= 0)? n : -n,  den = (d >= 0)? d : -d;
    long g = gcd(num, den); // normalize
    this.num = (sameSign)? num / g : -num / g;
    this.den = den / g;
  }

  static long gcd(long a, long b) { /* …compute gcd… */ }

  public Fraction plus(Fraction f) {
    return new Fraction(num*f.den+f.num* den,den* f.den);
  }

  public boolean equals(Object other) { // override default
    if (! (other instanceof Fraction) ) return false;
    Fraction f = (Fraction)(other);
    return num * f.den == den * f.num;
  }
  public int hashCode() {  /* override default */ }
}
```

Conseguenze dell'Immutabilità

- Vantaggi
  - semplice ed efficace
  - concettualmente elegante
- Svantaggi
  - tendenza a creare molti oggetti
- Quindi questa tecnica risulta vantaggiosa in caso di
  - oggetti "leggeri", ovvero con stato di dimensione contenuta
    - es. `Integer`, `Character`
  - pochi aggiornamenti
    - es. `Configuration`
  - In reatà risultava già raccomandata anche da Design Pattern nati fuori dal contesto della PC come "Flyweight" >>

Flyweight Design Pattern

- Questo pattern prevede la condivisione degli oggetti immutabili per evitare inutili e costose `new()`
- Si utilizza un pool di oggetti che saranno gli unici esemplari utilizzabili
- Esempio classico:
  - un documento ASCII come lista di `java.lang.Character`
  - è inutile creare nuovi oggetti istanza di `java.lang.Character` per ciascuna occorrenza dello stesso carattere
  - meglio creare solamente 256 oggetti, uno per carattere, che coprano l'intera tavola ASCII ad 8 bit
  - i documenti sono rappresentati come liste di riferimenti a questi oggetti immutabili condivisi, uno per carattere

Quando Usare Classi Immutabili?

- Dipende… vedi vantaggi/svantaggi
- In alcuni casi è chiaramente applicabile
  - classi con stato “piccolo”: classi Wrapper Java
  - classi di scarsa “dinamicità”
    - ad es. quelle usate per modellare configurazioni
    - l’elenco dei *listener* di un widget grafico>>
- Soprattutto nel caso di classi ad uso generale (ad esempio *stringhe*) la convenienza o meno dell’immutabilità dipende fortemente da come viene utilizzata la classe
- Talvolta non basta una sola classe per soddisfare tutti i possibili utilizzatori...

Classi Gemelle

- Per questo motivo spesso si preferisce lasciare all'utente la scelta rendendo disponibili due (o più) versioni "gemelle" di una classe
  - una versione immutabile e non sincronizzata
  - una versione mutabile e sincronizzata

- Si forniscono due alternative in *classi gemelle*
  - *Classe immutabile*: con i metodi di aggiornamento non sincronizzati ma che creano nuove copie
  - *Classe mutabile*: con i metodi di aggiornamento sincronizzati

Esempio Classi Gemelle

- Ad es. nella libreria standard java:
  - La tecnica della immutabilità è usata da `java.lang.String`
    - i suoi metodi non sono `synchronized`
  - Per evitare di creare troppi oggetti di tipo `String` si può invece usare `java.lang.StringBuffer`
    - i suoi metodi sono `synchronized`
  - Terza arrivata (java 1.5): `StringBuilder`
    - pensata per codice mono-thread che vuole usare stringhe mutabili ma non metodi sincronizzati (nelle prime versioni della JVM erano sensibilmente più costosi)
    - obsoleta dopo solo un cambio di versione (1.5 –>1.6!), per il rilevante miglioramento di prestazioni dei metodi sincronizzati in ambiente mono-thread

Tecnica delle *Classi Completamente Sincronizzate*

- Sfruttando i java monitor, si disciplinano gli accessi da parte dei thread alle strutture condivise come avviene per i *monitor astratti*
- Come progettare una classe che sia thread-safe grazie all'utilizzo dei java monitor?
  - Basta ispirarsi ai monitor *astratti* già visti
  - Consideriamo il caso standard in cui la risorsa/struttura condivisa sia rappresentata da una sola classe
    - Il cui stato è conservato in variabili di istanza private che sono le sole informazioni oggetto di accessi concorrenti
    - i cui metodi rappresentano tutte e sole le operazioni per accedere alla struttura

Classi Completamente Sincronizzate

- In una classe *completamente sincronizzata*:
  - non esistono campi pubblici
    - più in generale, l'incapsulamento è rispettato
  - tutti i metodi pubblici sono sincronizzati
  - lo stato è consistente all'inizio ed alla fine di ciascun metodo pubblico (anche in presenza di eccezioni)
- Ed inoltre:
  - il costruttore inizializza l'oggetto ad uno stato consistente
  - riferimenti all'oggetto in corso di costruzione non devono *sfuggire*
    - nessun thread, al di fuori di quello che sta eseguendo il costruttore, può ottenere un riferimento all'oggetto in costruzione
    - Attenzione: Il popolare Design Pattern *Singleton* induce in errore...

Grado di Parallelismo delle Classi Completamente Sincronizzate

- Uno dei vantaggi è che si possono agevolmente aggiungere altre operazioni atomiche sotto-forma di nuovi metodi pubblici `synchronized`
- Tuttavia, come nel caso dei monitor astratti, il grado di parallelismo può risultare eccessivamente conservativo
  - tutti i metodi sono eseguiti in mutua esclusione (sono sincronizzati)
  - anche quando la natura degli accessi consentirebbe di rilassare la disciplina di accesso a beneficio del livello di parallellismo
  - le deroghe alla regola dipendono dalla natura della classe e richiedono uno studio di caso in caso e non generalizzabile

Tecniche di Confinamento

- Consistono nella creazione di “domini” all’interno dei quali viene strutturalmente garantito che al più un thread alla volta può accedere un oggetto
- A seconda del tipo di dominio distinguiamo tecniche di confinamento
  - per *metodo*
  - per *thread*
  - per *oggetto*
  - per *gruppo*

Confinamento per Metodo

- Un oggetto è creato all'interno di un metodo e non può sopravvivere all'esecuzione di tale metodo o di una sequenza di metodi che lo seguono

```
class Plotter {
  // ...
  public void showNextPoint() {
    Point p = new Point();
    p.x = computeX(); p.y = computeY();
    display(p);
  }

  int computeX() { return 1; }
  int computeY() { return 1; }
  protected void display(Point p) {
    // somehow arrange to show p.
  }
}
```

Riferimenti che Sfuggono

- un metodo *m* può farsi sfuggire un riferimento *r* ad un oggetto *x* in quattro modi
  - *m* passa *r* come argomento in una invocazione di metodo o di costruttore
  - *m* passa *r* come valore restituito
  - *m* memorizza *r* in campi accessibili ad altri (in particolare campi statici)
  - *m* si fa sfuggire un riferimento, in uno dei metodi di cui sopra, che opportunamente inseguito conduce, *indirettamente*, ad *x*

Sequenze *Hand-Off* e *Sessioni*

- Il confinamento per metodo è un caso particolare di *hand-off* intra-thread
  - consegna e propagazione di un oggetto ad un thread con la garanzia che solo lui lo può manipolare
  - altro es. di sequenze di hand-off sono le *sessioni* per la gestione di una serie di richieste, ovvero sequenze di hand-off inter-thread
  - alle sessioni sono associati oggetti creati nel momento in cui sono instaurate
    - ad es. uno stream di output (cfr. servlet)
  - propagati ad un *worker-thread* e da questo, e *solo da questo*, utilizzato per tutte le operazioni che compongono la sessione
  - deallocati al termine della sessione
- Il worker-thread, a seconda delle *garanzie* ottenute dal thread fornitore, potrebbe o meno creare una copia dell'oggetto "in consegna"
  - per ciascuna copia bisogna cmq progettarne il *ciclo di vita*

Confinamento per Thread (1)

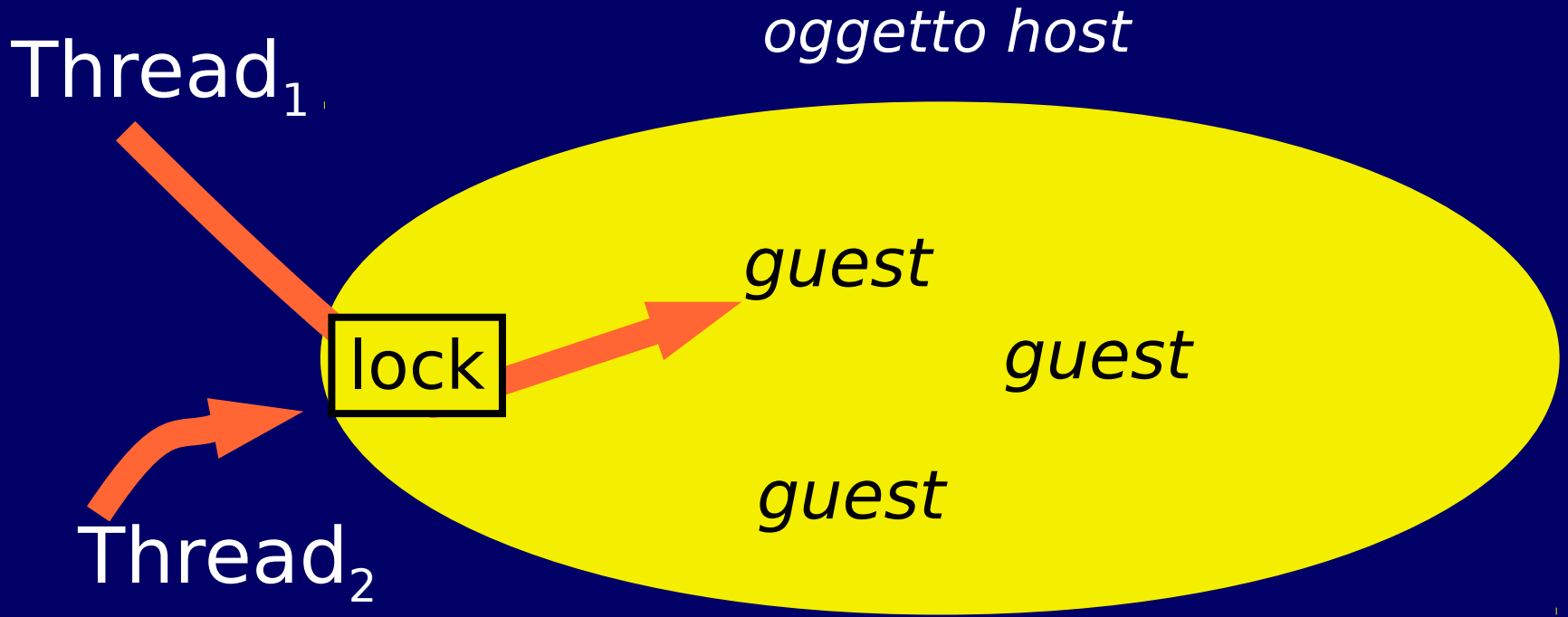
- Un oggetto è confinato in un thread quando solo un certo thread (il suo possessore) può accedervi
- Tutti gli altri f.d.e. devono lavorare per *delega* al f.d.e. possessore, in sostanza passandogli la specifica delle operazioni da compiere
- Quando si crea un thread per ogni sessione di fatto ai torna al C*onfinamento per Sessione*
- Anche i TSD possono farsi rientrare in quest'ambito
  - tramite `java.lang.ThreadLocal`
  - campi privati di una classe che deriva da `java.lang.Thread`
- Questa tecnica ha avuto un grande successo, ed è quella predominante per i framework a supporto della programmazione di applicazioni grafiche>>

Confinamento per Thread (2)

- Usata da Java FX 2, SWT e praticamente da quasi tutti i toolkit a supporto dello sviluppo di applicazioni grafiche, di qualsiasi genere (mobile, desktop)
- Usata da `javax.swing`
  - Il codice di risposta agli eventi grafici viene eseguito da un thread dedicato che li processa sequenzialmente
  - Per eseguire del codice che cambia l'aspetto grafico (ad es. quello in risposta agli eventi sui widget), il *main* thread deve mandargli una *delega*
- Principale beneficio e motivo del successo:
  - consente la scrittura di codice thread-safe anche a programmatori senza conoscenze avanzate di programmazione concorrente
  - basta rispettare alcune semplici regole senza necessariamente comprenderne la vera natura (vedi **`javax.swing.SwingWorker`**)
  - le alternative sono decisamente più complesse!

Confinamento per Oggetto

- Un oggetto *guest* è confinato in un oggetto *host:* quando gli accessi al guest sono disciplinati dalle limitazioni imposte dall'host



Confinamento per Oggetto: Classi Wrapper

Method Summary	
static Collection	**synchronizedCollection**(Collection c) Returns a synchronized (thread-safe) collection backed by the specified collection.
static List	**synchronizedList**(List list)
static Map	**synchronizedMap**(Map m)
static Set	**synchronizedSet**(Set s)
static Collection	**unmodifiableCollection**(Collection c) Returns an unmodifiable view of the specified collection.
static List	**unmodifiableList**(List list)
static Map	**unmodifiableMap**(Map m)
static Set	**unmodifiableSet**(Set s)

Confinamento per Gruppo

- Un oggetto è confinato in un gruppo di thread quando tutti i thread del gruppo vi possono accedere ma l'accesso è disciplinato di modo che solo uno di questi alla volta ne ha diritto

- *Token Ring*
  - un token viene fatto girare per un anello di thread
  - solo il possessore del token può accedere alla risorsa/oggetto condivisa

Esercizi

Esercizio: scrivere un programma java thread-safe per modellare un ascensore che si muove su N piani. La politica adottata dall'ascensore è quella di non cambiare direzione di marcia sino all'esaurimento delle richieste di servizio in direzione.

Inquadrare le soluzioni adottate in tale programma nella classificazione delle tecniche per la scrittura di programmi thread-safe.

Modello ad Attori e le Tecniche di Programmazione Thread-Safe

- Il modello ad attori (>>) può essere interpretato come l'applicazione congiunta di due tecniche
  - Tecnica di confinamento per thread
  - Tecnica della immutabilità degli oggetti

- Impone un cambio di paradigma rispetto al modo "tradizionale" di scrivere programmi
- In cambio si affrontano alla radice alcuni problemi intrinseci della PC

Modello ad Attori

- Paradossalmente spinge a programmare applicazioni concorrenti ma locali come fossero distribute
- E' un modello di programmazione semplice, ma “sorprendente” quando utilizzato per lo sviluppo di codice concorrente ma *NON* distribuito

✔ In passato l'approccio opposto è stato ripetutamente tentato

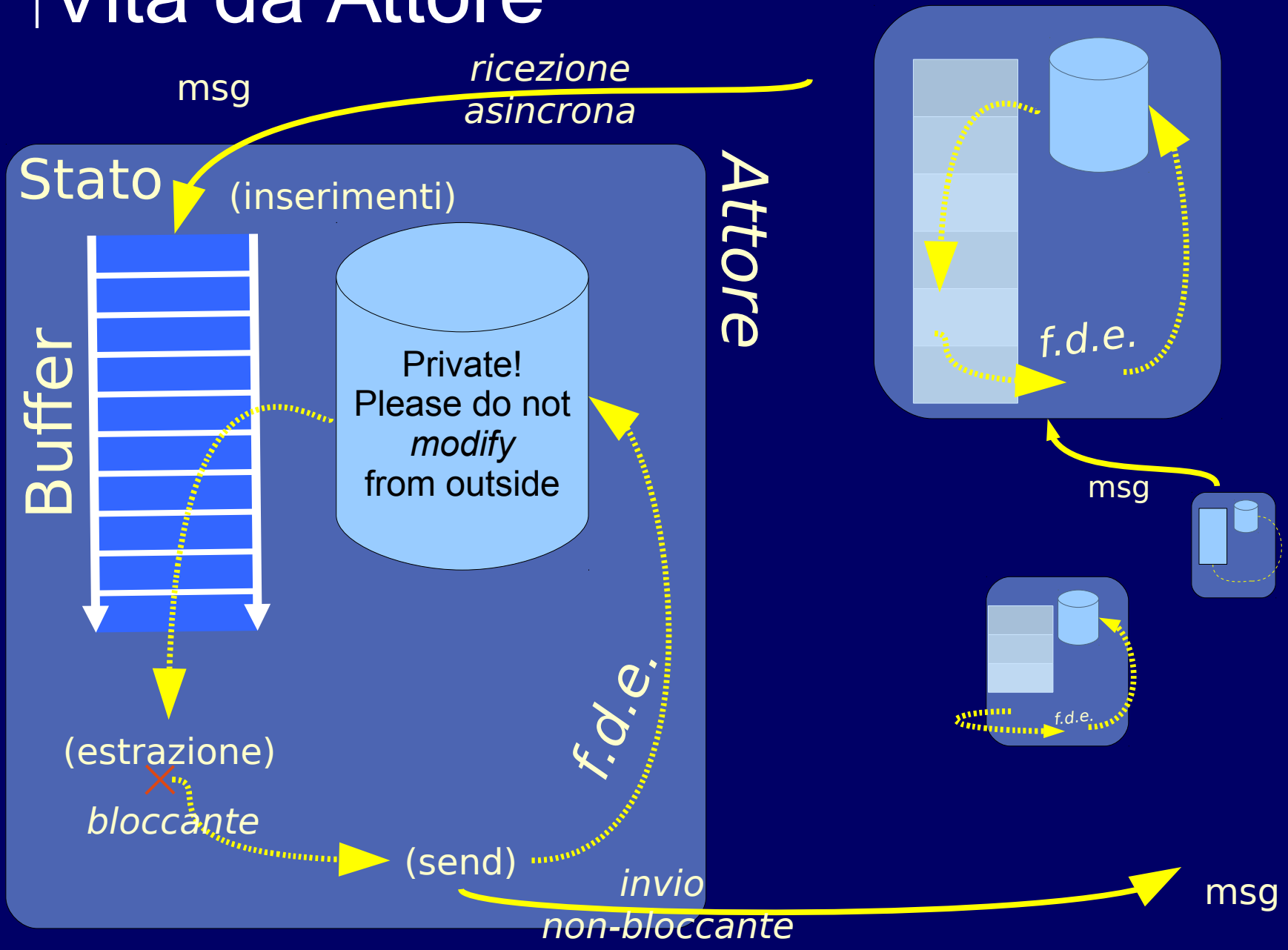
- ovvero, molte proposte hanno tentato di rendere trasparente la distribuzione dell'esecuzione (ad es. *RPC*)
  - perché facevano leva sulle competenze più diffuse
  - in futuro avrà senso richiedere che la formazione sia orientata alla programmazione di attività concorrenti ben prima di quanto accade ora?
  - secondo paradigmi più semplici di quelli attualmente ritenuti fondamentali, ma che non tengono dei problemi di PC?

Il Modello ad Attori

- Le applicazioni sviluppate secondo il modello ad attori sono organizzate attorno a delle unità di processamento chiamate *Attori*
- Una fusione piuttosto naturale del concetto di
  - Oggetto (tipica della POO)
  - f.d.e. (tipica della PC)

  (vedi anche “Active Objects”: *https://en.wikipedia.org/wiki/Active_object*)
- Un attore possiede un unico f.d.e. con accesso *esclusivo* allo suo stato (reso con un singolo oggetto)
- Un attore comunica con gli altri attori tramite messaggi
  - inviandoli (in maniera *non-bloccante*)
  - ricevendoli in un buffer locale di processamento da cui effettua estrazioni (in maniera *bloccante*)
- L'elaborazione di un attore è descrivibile come un ciclo indefinito di ricezione e processamento di nuovi messaggi…

Vita da Attore



Modello ad Attori e le Tecniche di Programmazione Thread-Safe

- Il modello ad attori può essere interpretato come l'applicazione congiunta di due tecniche
- Tecnica di confinamento per thread
  - Un attore è associato ad un singolo f.d.e.
- Tecnica della immutabilità degli oggetti
  - I messaggi scambiati sono immutabili
  - (Vedi HWA>>)

La Programmazione e la Thread-Safeness