PCD Module 3.2

Attori

Attori

- Introdotto da Carl Hewitt (70s), sviluppato da Gul Agha, Akinori Yonezawa (80s-90s) come **unione tra OOP e concorrenza**
- Modello alternativo alla programmazione multithreaded.

Modello

Teoria matematica che tratta gli attori come primitive universali di computazioni concorrenti digitali.

Molte librerie basate sui linguaggi più popolari (Broadway, Actalk, AALR, Akka, Oscar...)

Idea di base

Scambio di messaggi asincrono tra oggetti autonomi puramente reattivi chiamati **attori**. Ogni attore ha un identifiatore univoco e un'unica mailbox dove i messaggi sono accodati. Ogni interazione ha luogo come scambio di messaggi asincrono.

Attori come oggetti autonomi reattivi

Astrazione degli attori: entità computazionali che incapsulano uno stato, il comportamento e il **flusso logico di controllo** (non incorporato in OOP). Gli attori non possiedono necessariamente un thread fisico di controllo (OS): si possono avere sistemi di N attori in esecuzione su sistemi con M core/thread.

Primitive

- **send**: invio asincrono di messaggi ad un attore *(invocazione di funzioni nella prog. sequenziale)*
- **create**: creazione di un attore con il comportamento specificato *(astrazione delle funzioni nella prog. sequenziale)*
- **become**: specifica un nuovo comportamento (stato) usabile dall'attore per rispondere ad un messaggio. Da all'attore un comportamento sensibile alla cronologia, necessario per oggetti dati mutabili e condivisi.

Un attore può comunicare con attori ai quali è connesso, ovvero dei quali conosce l'identificatore. L'ID può essere scambiato nei messaggi.

Aspetti chiave della semantica

- **comportamento puramente reattivo**: un attore lavora solo se riceve messaggi, altrimenti è bloccato
- **incapsulamento degli stati**: un attore non può accedere allo stato interno di altri attori
- **semantica a macro-step** (run-to-completion): una volta ricevuto un messaggio, la relativa computazione è eseguita dopo aver ricevuto altri messaggi
- **imparzialità nell'invio e lavorazione dei messaggi**: un messaggio inviato ad un attore è eventualmente spedito a destinazione e processato dall'attore
- **trasparenza nella locazione**: per inviare un messaggio non serve sapere dov'è posizionato, solamente l'identificatore.

Osservazioni sui messaggi

- *semantica della send*: nel modello base, i messaggi inviati sono *eventualmente* recapitati all'attore destinatario, ma non si può assumere quanto possa impiegaci.
- *nessun ordinamento e pattern di scambio dei messaggi*: non si può assumere l'ordine di ricezione dei messaggi, anche se le send sono effettuate dallo stesso attore. Qualsiasi tipo di ordinamento deve essere effettuato tramite pattern di scambio di messaggi.

Implementazioni

Implementato in diversi linguaggi e framework recenti (Erlang, Scala Actors, HTML5 Web Workers, Google Dart...), con diverse funzionalità e semantiche. Alcune differenze importanti: **loop dei messaggi** esplicito vs implicito, ricezione guidata dai pattern, ecc...

Loop degli eventi incapsulato e ricezione implicita

Comportamento dell'attore governato da un loop degli eventi:

```
loop {
  msg <- waitForMsg()
  handler <- selectHandler(msg)
  execute (handler)
}
```

- Il comportamento degli attori è definito dagli handler (body). Ogni handler è una sequenza di azioni che modificano lo stato interno dell'attore e possibilmente manda messaggi o crea altri attori
- Architettura di controllo dell'attore esplicita: loop embeddato all'infuori del codice del programmatore
- **Semantica a macro-step**: un handler è eseguito completamente prima di ricevere il prossimo messaggio. Gli handler dovrebbero avere un comportamento non bloccante - l'unico puntio bloccante è gestito dal loop degli eventi.
- Stessa semantica delle *architetture guidate dagli attori*, es. OS guidati dagli eventi e programmazione

web moderna.

Esempi concreti: ActorFoundry e Akka

- **ActorFoundry**: framework a livello accademico basato su Java e sviluppato dal gruppo di Agha
- **Akka**: framework ad attori a livello industriale per ambienti Java e Scala

Esempi di codice sulle slide.

Osservazioni

- Consequenze della semantica a macro step
  - elimina le race conditions e simplifica il ragionamento sui programmi
  - ma complica pesantemente la programmazione: **spaghetti asincroni**
    - problema che affligge le architetture guidate dagli eventi (callback e loop degli eventi), es. Javascript
    - logica dell'applicazione suddivisa in un insieme non strutturato di handlers, ognuno che reagisce ad un evento
    - soluzioni: **promises**, promise pipelines

Sincronizzazione e ordinamento dei messaggi

- Sincronizzazione negli attori ottenuta tramite comunicazione
- Due pattern usati:
  - **Remote Procedure Call (RPC)-like messaging**: invio della richiesta e attesa della ricezione prima di procedere
  - **Vincoli locali di sincronizzazione**: scegliendo quando si vogliono ricevere messaggi
- I costrutti sono tipicamente forniti per specificare tali pattern, definibili in termini di costrutti primitivi degli attori, ma fornendoli come oggetti linguistici di prima classe che semplificano la programmazione.

Scambio di messaggi RPC-like

- Può essere implementato come un protocollo:
  - l'attore client invia la richiesta
  - il client controlla i messaggi inarrivo
  - se il messaggio in arrivo corrisponde alla risposta alla sua richiesta, il client prende l'azione appropriata
  - se un messaggio in arrivo non corrisponde alla risposta alla sua richiesta, il messaggio deve essere gestito
  - il client continua a controllare i messaggi per replicare

- Supporto diretto fornito dai framework: primitiva `call` in ActorFoundry

Il problema dell'ordinamento dei messaggi

Con l'asincronismo, il numero di possibili ordinamenti in cui i messaggi possono arrivare è esponenziale al numero di messaggi in attesa (es. messaggi inviati ma non ricevuti).

Un mittente può non essere al corrente dello stato dell'attore ricevente del messaggio: il destinatario potrebbe non essere in uno stato dove può processare il messaggio che riceve.

La necessità di tale ordinamento porta ad una complessità considerabile nei programmi concorrenti, alcune volte introducendo bug o inefficienze a causa di strategie di implementazione subottimali.

Esempio: processo di stampa

Suppose a ‘get’ message from an idle printer to its spooler may arrive when the spooler has no jobs to return the printer. One way to address this problem is for the spooler to refuse the request.

- now the printer needs to repeatedly poll the spooler until the latter has a job.
- busy waiting => can be expensive–preventing the waiting actor from possibly doing other work while it “waits”, and it results in unnecessary message traffic. An alternate is to the spooler buffer the ‘get’ message for deferred processing
- the effect of such buffering is to change the order in which the messages are processed in a way that guarantees that the number of messages put messages to the spooler is always greater than the number of get messages processed by the spooler

Vincoli di sincronizzazione locale

- Se i messaggi in attesa sono processati esplicitamente all'interno del corpo di un attore, il codice che specifica la funzionalità dell'attore è mischiato con la logica determinando l'ordine in cui l'attore processa i messaggi. Tale mix viola il principio di separazione dei compiti.
- Proposte di vari costrutti per permettere ai programmatori di specificare l'ordinamento corretto in modo astratto e modulare
- Diversi linguaggi e framework ad attori forniscono tali costrutti

Soluzione in ActorFoundry: L.S.C.

Abilitando e disabilitando condizioni specificati dagli handler tramite annotazioni:

```
@Disable(messageName = "put")
public Boolean disablePut(Integer x) {
    if (bufferReady) {
        return (tail == bufferSize);
    } else return true;
}
```

(Quasi) soluzione in Akka

Utilizzando una coda implicita separata:

- `stash` per memorizzare messaggi ricevuti per una gestione futura
- `unstash` per riportare i messaggi impilati sulla coda principale

```
import akka.actor.Stash
class ActorWithProtocol extends Actor with Stash {
  def receive = {
    case "open" =>
      unstashAll()
      context.become({
        case "write" => // do writing...
        case "close" =>
          unstashAll()
          context.unbecome()
        case msg => stash()
      }, discardOld = false) // stack on top instead of replacing
    case msg => stash()
    }
}
```

Futures

Schema di progettazione ricorrente che coinvolge più attori interagenti: Future, Serializer, Fork-Join, ecc.

Future/Promises si riferiscono ad oggetti che agiscono come proxy per un risultato che inizialmente è sconosciuto a causa della computazione incompleta dei suoi valori.

Impostare il valore in una future è anche chiamato resolving, fulfilling o binding. La Promise è il meccanismo che imposta la Future.

Nel modello ad attori, una Future è rappresentata da un attore con tre stati:

- la future non ha valore e nessun cliente attende il valore
- se dei clienti vogliono leggere il valore, sono messi in coda finchè il valore è disponibile

- quando il valore è disponibile, tutti i clienti in attesa sono notificati, e i susseguenti

L'identificatore dell'attore futuro è rilasciato immediatamente ad un cliente.

Punti chiave

- migliora il parallelismo
- implementa forme sincrone di comunicazione usando lo scambio di messaggi asincrono, massimizzando la concorrenza: la future può essere passata ad altri attori prima di essere risolta
- i linguaggi ad attori forniscono un supporto diretto a future/promises

Problema della proattività

Gli attori sono entità puramente reattive. Un problema principale è: come modellare comportamenti proattivi, ovvero comportamenti orientati a completare certi task per mezzo di un qualche piano di azione? Come integrare propriamente comportamenti proattivi e reattivi?

Soluzioni parziali: dividendo l'attore in più attori interagenti o tramite **autoinvio dei messaggi**.

Integrazione tra oggetti ed attori

Modelli di comunicazione del loop ad eventi

Un modello che integra oggetti passivi ed attori, introdotto in E e AmbientTalk. Gli attori sono contenitori (VAT) di oggetti passivi. Lo scambio di messaggi avviene tra oggetti che possono appartenere allo stesso contenitore (sincrono) o altri contenitori (asincrono). Ogni contenitore è basato sul modello del loop ad eventi.

Loop esplicito / Receive esplicita

Nessuna architettura di controllo esplicita e loop ad eventi impliciti. I loop di ricezione devono essere esplicitamente gestiti dai programmatori.

Primitiva **receive**, tipicamente fornita di qualche supporto alla selezione di receive con guards. Approcci concreti: Erlang, Akka.

Attori in Scala Actors Library e Erlang: vedi slide.

Erlang

Linguaggio funzionale che fornisce supporto nativo alla programmazione concorrente, basato su **processi** che processano comunicazioni asincrone tramite **scambio di messaggi**.

Macchina virtuale concorrente BEAM, con concetto astratti di processo. Gestione dei processi estremamente efficiente.

Erlang come linguaggio funzionale

Un programma descrive una serie di funzioni. Gli operatori sono un tipo speciale di funzioni. Ogni funzione usa il *pattern matching* per determinare quale funzione eseguire (variabili iniziano con le maiuscole). Non esistono variabili globali. Esistono moduli per "impacchettare" le funzioni.

```
fact(0) -> 1;
fact(N) -> N * fact
fib(1) -> 1;
fib(2) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

Chiamata a funzione: `X = math:fact(100).`

Strutture dati

- Tipi di dato
  - base (atomi): simboli, costanti, numeri, stringhe
  - strutturato:
    - **tuple**: struttura ordinata di record con numero fisso di elementi. Supporta pattern matching. (come array)
    - **liste**: per memorizzare numero variabile di oggetti. Notazione Head e Tail (H, T)

Definire i processi (attori)

Un processo (attore) è un'attività computazionale il cui comportamento è dato da qualche funzione. La primitiva `spawn` lancia un processo e ne ricava il PID, es.: `Pid = spawn(math, fact, [999])`.

I processi sono entità logiche: BEAM mappa i processi logici ai thread fisici. I programmi possono avere migliaia di processi, la cui creazione è molto economica.

Scambio di messaggi asincrono

- I processi possono comunicare solo tramite scambio di messaggi: ogni processo ha una mailbox dove i messaggi sono accodati.
- **inviare un messaggio**: si usa l'operatore `!`, esempio: `Pid ! message`.
- **ricevere un messaggio**: si usa il costrutto `receive`. Semantica: bloccato finchè un messaggio nella coda corrisponde a un pattern e la guard corrispondente è vera.

```
receive
Pattern1 [when Guard1 ] -> Expression1;
Pattern2 [when Guard2 ] -> Expression2; ...
end
```

Esempi di codice sulle slide.

Sincronizzazione e ordinamento di messaggi tramite receive esplicite

La sincronizzazione può essere implementata tramite pattern di comunicaizone basati anche sulle receive. L'ordinamento dei messaggi è risolto dal comportamento di default delle receive selettive:

- i messaggi che non corrispondo a nessun braccio della receive selettiva sono rimossi e piazzati nella coda differita
- quando un messaggio che corrisponde a un braccio è trovato e la receive è sbloccata, tutti i messaggi nella coda differita sono rimessi sulla coda principale.