1. Introduzione alla concorrenza

La concorrenza rappresenta uno dei concetti fondamentali dell'informatica moderna. Un sistema concorrente è formato da più componenti che operano simultaneamente, interagendo tra loro attraverso meccanismi di comunicazione.

1.1 Concorrenza vs Parallelismo

È importante distinguere tra:

- **Concorrenza**: principio di strutturazione che descrive sistemi composti da componenti indipendenti che possono progredire in maniera autonoma ma coordinata (concetto logico)
- **Parallelismo**: esecuzione simultanea di attività su hardware che lo permette (concetto operativo)

Come evidenziato nelle lezioni, "la concorrenza è utile per sfruttare il parallelismo, ma è molto più di questo".

1.2 Complessità dei sistemi concorrenti

I sistemi concorrenti presentano sfide specifiche:

- **Problemi classici**: deadlock, starvation, fairness
- **Nuove complessità**: connettività, guasti remoti, sicurezza, controllo delle risorse

L'esempio analizzato a lezione del problema del buffer tra produttore e consumatore evidenzia come anche problemi apparentemente semplici diventino complessi in contesto concorrente.

1.3 Approccio fondazionale

Il corso adotta un approccio rigoroso, matematicamente fondato, per lo studio della concorrenza:

- Identificazione degli operatori e costrutti fondamentali
- Comprensione della molteplicità di linguaggi, architetture e paradigmi
- Tecniche formali per progettazione, specifica e verifica

2. Il Calcolo dei Sistemi Comunicanti (CCS)

Il CCS, sviluppato da Robin Milner negli anni '80, è un calcolo di processi per modellare sistemi concorrenti. L'idea centrale è rappresentare un sistema come un insieme di processi che interagiscono tramite porte di comunicazione.

2.1 Concetti chiave del CCS

Il CCS si basa su alcuni principi fondamentali:

- Tutto è un processo
- I processi comunicano tramite interazioni sincrone
- Le interazioni avvengono su canali denominati

2.2 Sintassi informale del CCS

La sintassi informale comprende:

- **Inazione** (`0`): processo che non fa nulla
- **Prefisso d'azione** (`α.P`): esegue l'azione `α` e poi si comporta come `P`
- **Costante di processo** (`K`): definita come `K := P`
- **Scelta non deterministica** (`P + Q`): si comporta come `P` o come `Q`
- **Composizione parallela** (`P | Q`): `P` e `Q` operano in parallelo
- **Restrizione** (`P\L`): nasconde le azioni in `L`
- **Ridenominazione** (`P[f]`): rinomina le azioni secondo `f`

2.3 Esempi di base da lezione

Riprendiamo alcuni esempi significativi visti a lezione:

- **Clock**: `Clock = tick.Clock`
- **Buffer unario**: `C = in(x).C'(x)` ; `C'(x) = out(x).C`
- **Macchina del caffè**: `CM = coin.coffee.CM`
- **Macchina del caffè guasta**: `BCM = coin.BCM + coin.coffee.BCM + coin.Fail.0 + coin.coffee.Fail.0 + fail.0`

2.4 Sintassi formale

Formalmente, dato:

- Un insieme numerabile di nomi di canali `A = {a, b, c, ...}`
- Un insieme di azioni `Act = A ∪ {ā | a ∈ A} ∪ {τ}`
- Un insieme di costanti di processo `K = {K, K', K1, K2, ...}`

La sintassi del CCS è definita come:

```
P, Q ::= 0 | α.P | P + P | P | P | P\L | P[f] | K
```

2.5 Semantica operazionale

La semantica operazionale è definita da regole di inferenza che specificano come i processi evolvono attraverso le azioni. Queste regole costituiscono un sistema di transizione etichettato (LTS):

- **ACT**: $\alpha.P \to^{\alpha} P$
- **SUM**: $P_j \to^{\alpha} P'$ implica $P_1 + P_2 \to^{\alpha} P'$ per j ∈ {1,2}
- **PAR1**: $P \to^{\alpha} P'$ implica $P|Q \to^{\alpha} P'|Q$
- **PAR2**: $Q \to^{\alpha} Q'$ implica $P|Q \to^{\alpha} P|Q'$
- **COM**: $P \to^{a} P'$ e $Q \to^{\bar{a}} Q'$ implica $P|Q \to^{\tau} P'|Q'$
- **RES**: $P \to^{\alpha} P'$ e $\alpha,\bar{\alpha} \notin L$ implica $P\backslash L \to^{\alpha} P'\backslash L$
- **REL**: $P \to^{\alpha} P'$ implica $P[f] \to^{f(\alpha)} P'[f]$
- **CONST**: $P \to^{\alpha} P'$ e $K := P$ implica $K \to^{\alpha} P'$

2.6 Esempi completi di modellazione

2.6.1 Mutua esclusione con semaforo binario

```
Sem = p.v.Sem
User = p.enter.exit.v.User
Sys = (User|Sem)\{p,v}
```

Come dimostrato a lezione, questo sistema garantisce che solo un processo alla volta possa accedere alla sezione critica.

2.6.2 Problema dei filosofi a cena

Come visto nelle lezioni (LCD-2024-03-11.pdf):

```
Fork_i = take_i.leave_i.Fork_i
Phil_i = think.take_i.take_(i+1).eat.leave_i.leave_(i+1).Phil_i
System = (P₁|...|P₅|F₁|...|F₅)\{take_i,leave_i | i = 1,...,5}
```

Questo esempio illustra il classico problema di deadlock, poiché se ogni filosofo prende la forchetta alla sua sinistra, si crea una situazione di stallo.

2.6.3 Algoritmo di Peterson per mutua esclusione

Dalle lezioni (LCD-2024-03-11.pdf):

```
// Variabili condivise
b₁ = false, b₂ = false, k = 1

// Processo P₁
```

```
P₁ = while true do
  begin
    b₁ = true
    k = 2
    while (b₂ and k=2) do skip
    // sezione critica
    b₁ = false
  end

// Processo P₂
P₂ = while true do
  begin
    b₂ = true
    k = 1
    while (b₁ and k=1) do skip
    // sezione critica
    b₂ = false
  end
```

In CCS, questo viene modellato rappresentando le variabili condivise come processi:

```
B_t = get_t.B_t + set_t.B_t + set_f.B_f
B_f = get_f.B_f + set_t.B_t + set_f.B_f
K_i = get_i.K_i + set_i.K_i + set_j.K_j

P_i = set_bi_t.set_k_j.P_i1
P_i1 = get_bj_f.P_i2 + get_bj_t.(get_k_i.P_i2 + get_k_j.P_i1)
P_i2 = enter_i.exit_i.set_bi_f.P_i

System = (P₁|P₂|B₁|B₂|K)\L
```

Come dimostrato nelle lezioni, questo algoritmo garantisce la mutua esclusione.

3. Value-passing CCS

Value-passing CCS estende il CCS base con la capacità di trasmettere valori durante la comunicazione.

3.1 Sintassi estesa

```
P, Q ::= 0 | α.P | P + P | P | P | P\L | P[f] | K(e₁,...,eₙ) | if b then P
else Q
```

dove:

- α può essere $a(x)$ (input), $\bar{a}\langle e\rangle$ (output) o τ
- e è un'espressione aritmetica
- b è un'espressione booleana

3.2 Semantica operazionale con passaggio di valori

Le regole di transizione includono:

- **IN**: $a(x).P \xrightarrow{a(v)} P[v/x]$ (riceve il valore v sul canale a e sostituisce x con v in P)
- **OUT**: $\bar{a}\langle e\rangle.P \xrightarrow{\bar{a}\langle v\rangle} P$ se e valuta a v
- **COM-VAL**: $P \xrightarrow{\bar{a}\langle v\rangle} P'$ e $Q \xrightarrow{a(v)} Q'$ implica $P|Q \xrightarrow{\tau} P'|Q'$

3.3 Esempi importanti di Value-passing CCS

3.3.1 Buffer FIFO a capacità 2

Come presentato a lezione (LCD-2024-03-12.pdf):

```
F₂ = in(x).F₁(x)
F₁(x) = out(x).F₂ + in(y).F₀(x,y)
F₀(x,y) = out(x).F₁(y)
```

3.3.2 Buffer non ordinato a capacità 2

```
B₂ = in(x).B₁(x)
B₁(x) = out(x).B₂ + in(y).B₀(x,y)
B₀(x,y) = out(x).B₁(y) + out(y).B₁(x)
```

3.3.3 Contatore con incremento e decremento

```
C(x) = inc.C(x+1) + if x = 0 then dec.C(0) else dec.C(x-1)
```

3.4 Encoding in CCS base

Value-passing CCS può essere codificato in CCS base trattando ogni coppia canale-valore come un canale separato:

```
⟦a(x).P⟧ = Σ_{v ∈ V} a_v.⟦P[v/x]⟧
⟦ā⟨e⟩.P⟧ = ā_m.⟦P⟧ se e valuta a m
⟦τ.P⟧ = τ.⟦P⟧
⟦P | Q⟧ = ⟦P⟧ | ⟦Q⟧
```

```
⟦P + Q⟧ = ⟦P⟧ + ⟦Q⟧
⟦P\L⟧ = ⟦P⟧\{a_v | a ∈ L, v ∈ V}
```

4. Equivalenza comportamentale: Bisimilarità

4.1 Intuizione della bisimilarità

Due processi sono considerati equivalenti se esibiscono lo stesso comportamento osservabile. La bisimilarità è una nozione di equivalenza che cattura l'idea che due processi si comportino allo stesso modo.

4.2 Definizione formale di bisimulazione forte

Una relazione binaria `R` sui processi è una bisimulazione se per ogni coppia di processi `(P,Q) ∈ R`:

1. Se `P →ᵅ P'`, allora esiste `Q'` tale che `Q →ᵅ Q'` e `(P',Q') ∈ R`
2. Se `Q →ᵅ Q'`, allora esiste `P'` tale che `P →ᵅ P'` e `(P',Q') ∈ R`

Due processi `P` e `Q` sono bisimilari (notazione: `P ~ Q`) se esiste una bisimulazione `R` tale che `(P,Q) ∈ R`.

4.3 Caratterizzazione come gioco

La bisimilarità può essere caratterizzata come un gioco tra due giocatori:

- **Attaccante**: cerca di dimostrare che i processi non sono bisimilari
- **Difensore**: cerca di dimostrare che i processi sono bisimilari

Le regole del gioco:

1. Si parte con due processi `P` e `Q`
2. L'Attaccante sceglie un processo (P o Q) e una sua transizione `→ᵅ`
3. Il Difensore deve rispondere con una transizione corrispondente dell'altro processo
4. Il gioco continua con i processi risultanti

`P ~ Q` se e solo se il Difensore ha una strategia vincente.

4.4 Proprietà della bisimilarità

- **Relazione di equivalenza**: riflessiva, simmetrica e transitiva
- **Congruenza**: preservata da tutti gli operatori del CCS
- **Più grande bisimulazione**: contiene tutte le altre bisimulazioni

4.5 Bisimulazione up-to

La bisimulazione up-to bisimilarità (menzionata in LCD-2024-03-26.pdf) è una tecnica che semplifica le dimostrazioni di bisimilarità.

Una relazione R è una bisimulazione up-to bisimilarità se per ogni $(P,Q) \in R$:

1. Se $P \xrightarrow{\alpha} P'$, allora esiste Q' tale che $Q \xrightarrow{\alpha} Q'$ e $P' \sim R \sim Q'$
2. Se $Q \xrightarrow{\alpha} Q'$, allora esiste P' tale che $P \xrightarrow{\alpha} P'$ e $P' \sim R \sim Q'$

Dove $P' \sim R \sim Q'$ significa che esistono P'' e Q'' tali che $P' \sim P''$, $(P'',Q'') \in R$ e $Q'' \sim Q'$.

Teorema: Se R è una bisimulazione up-to bisimilarità e $(P,Q) \in R$, allora $P \sim Q$.

Esempio di applicazione

Come visto a lezione, per dimostrare che $Cell|Cell \sim F_2$, possiamo definire:

$$R = \{(Cell|Cell,\ F_2)\} \cup \{(C(m)|Cell,\ F_1(m))\} \cup \{(Cell|C(n),\ F_1(n))\} \cup \{(C(m)|C(n),\ F_0(m,n))\}$$

e verificare che R è una bisimulazione up-to.

4.6 Esempi di processi bisimilari e non bisimilari

- $a.b.0 + a.c.0 \not\sim a.(b.0 + c.0)$: nel primo processo, la scelta è esterna (visibile), nel secondo è interna (nascosta)
- $a.(b.0 + c.0) \sim a.b.0 + a.c.0 + a.(b.0 + c.0)$: il terzo termine è ridondante
- $a.0 \mid b.0 \sim a.b.0 + b.a.0$: entrambi possono eseguire a e b in qualsiasi ordine

5. Bisimilarità debole

5.1 Motivazione

La bisimilarità forte considera le azioni interne τ come qualsiasi altra azione. Tuttavia, poiché τ rappresenta comunicazioni interne non osservabili, potremmo voler astrarre da esse.

5.2 Transizioni deboli

Definiamo:

- $P \Rightarrow P'$ se $P \xrightarrow{\tau}{}^{*} P'$ (zero o più transizioni τ)
- $P \stackrel{\alpha}{\Longrightarrow} P'$ se $P \Rightarrow \xrightarrow{\alpha} \Rightarrow P'$ per $\alpha \neq \tau$
- $P \stackrel{\tau}{\Longrightarrow} P'$ se $P \Rightarrow P'$

5.3 Definizione formale di bisimulazione debole

Una relazione binaria R sui processi è una bisimulazione debole se per ogni $(P,Q) \in R$:

1. Se $P \rightarrow^{\alpha} P'$, allora esiste Q' tale che $Q \Rightarrow^{\alpha} Q'$ e $(P',Q') \in R$
2. Se $Q \rightarrow^{\alpha} Q'$, allora esiste P' tale che $P \Rightarrow^{\alpha} P'$ e $(P',Q') \in R$

Due processi P e Q sono debolmente bisimilari (notazione: $P \approx Q$) se esiste una bisimulazione debole R tale che $(P,Q) \in R$.

5.4 Esempi di processi debolmente bisimilari

- $\tau.a.0 \approx a.0$: l'azione interna τ è ignorata
- $a.(\tau.b.0 + \tau.c.0) \approx a.b.0 + a.c.0$: la scelta interna dopo a è equivalente a una scelta esterna prima di a
- Sistema di trasmissione affidabile su canale inaffidabile (esempio dalla lezione)

5.5 Proprietà della bisimilarità debole

- **Relazione di equivalenza**: riflessiva, simmetrica e transitiva
- **Non è una congruenza completa**: non è preservata dall'operatore di scelta (+)
- **La bisimilarità forte implica la bisimilarità debole**: $P \sim Q$ implica $P \approx Q$

5.6 Congruenza osservazionale

Per ovviare al problema della mancata congruenza, si definisce la congruenza osservazionale:

$P \simeq Q$ se:

1. Se $P \rightarrow^{t} P'$, allora esiste Q' tale che $Q \rightarrow^{t}\Rightarrow Q'$ e $P' \approx Q'$
2. Se $Q \rightarrow^{t} Q'$, allora esiste P' tale che $P \rightarrow^{t}\Rightarrow P'$ e $P' \approx Q'$
3. Se $P \rightarrow^{\alpha} P'$ con $\alpha \neq \tau$, allora esiste Q' tale che $Q \Rightarrow^{\alpha} Q'$ e $P' \approx Q'$
4. Se $Q \rightarrow^{\alpha} Q'$ con $\alpha \neq \tau$, allora esiste P' tale che $P \Rightarrow^{\alpha} P'$ e $P' \approx Q'$

Teorema: $\simeq$ è una congruenza per tutti gli operatori del CCS, incluso l'operatore di scelta.

6. Teoria dei punti fissi e Bisimilarità

6.1 Teoria dei punti fissi

Un punto fisso di una funzione $f: D \to D$ è un elemento $x \in D$ tale che $f(x) = x$.

6.1.1 Ordini parziali completi (CPO)

Un insieme parzialmente ordinato $(D, \sqsubseteq)$ è un CPO se:

- Ha un elemento minimo $\perp$ (bottom)

- Ogni sottoinsieme diretto ha un limite superiore (lub)

6.1.2 Funzioni continue

Una funzione `f: D → E` tra CPO è continua se:

- È monotona: se `x ⊑ y`, allora `f(x) ⊑ f(y)`
- Preserva i lub di insiemi diretti: `f(⊔S) = ⊔f(S)` per ogni insieme diretto `S`

6.1.3 Teorema del punto fisso di Kleene

Ogni funzione continua `f: D → D` su un CPO `D` ha un punto fisso minimo, dato da:

```
fix(f) = ⊔_{n ≥ 0} f^n(⊥)
```

6.2 Bisimilarità come punto fisso

La bisimilarità può essere caratterizzata come punto fisso di un operatore adeguato.

Sia `F: 2^(Proc×Proc) → 2^(Proc×Proc)` definita come:

```
F(R) = {(P,Q) | se P →ᵅ P' allora esiste Q' tale che Q →ᵅ Q' e (P',Q') ∈ R,
         e se Q →ᵅ Q' allora esiste P' tale che P →ᵅ P' e (P',Q') ∈ R}
```

Teorema: La bisimilarità `~` è il più grande punto fisso di `F`.

6.3 Algoritmo per sistemi finiti

Per sistemi di transizione etichettati finiti, la bisimilarità può essere calcolata utilizzando l'algoritmo di raffinamento delle partizioni:

1. Si parte con una singola partizione contenente tutti gli stati
2. Iterativamente si raffina la partizione in base alle transizioni
3. L'algoritmo termina quando non è più possibile alcun raffinamento
4. Gli stati nella stessa partizione finale sono bisimilari

Questo algoritmo ha complessità temporale `O(mn log n)`, dove `m` è il numero di transizioni e `n` è il numero di stati.

7. Logica di Hennessy-Milner

7.1 Sintassi

La logica di Hennessy-Milner (HML) è una logica modale per specificare proprietà dei processi:

```
ϕ ::= true | ¬ϕ | ϕ ∧ ϕ | ⟨α⟩ϕ
```

Dove:

- `true` è sempre soddisfatto
- `¬ϕ` è soddisfatto se `ϕ` non è soddisfatto
- `ϕ ∧ ψ` è soddisfatto se sia `ϕ` che `ψ` sono soddisfatti
- `⟨α⟩ϕ` è soddisfatto se c'è una transizione `α` verso uno stato che soddisfa `ϕ`

Altri operatori possono essere definiti come abbreviazioni:

- `false ≡ ¬true`
- `ϕ ∨ ψ ≡ ¬(¬ϕ ∧ ¬ψ)`
- `[α]ϕ ≡ ¬⟨α⟩¬ϕ` (tutte le transizioni `α` portano a stati che soddisfano `ϕ`)

7.2 Semantica

La relazione di soddisfacimento `⊨` è definita induttivamente:

```
P ⊨ true per tutti i P
P ⊨ ¬ϕ sse P ⊭ ϕ
P ⊨ ϕ ∧ ψ sse P ⊨ ϕ e P ⊨ ψ
P ⊨ ⟨α⟩ϕ sse esiste P' tale che P →ᵅ P' e P' ⊨ ϕ
```

7.3 Esempi di formule

- "Può eseguire un'azione coffee": `⟨coffee⟩true`
- "Non può eseguire un'azione coffee": `¬⟨coffee⟩true` o equivalentemente `[coffee]false`
- "Dopo un'azione coffee, può eseguire un'azione tea": `⟨coffee⟩⟨tea⟩true`
- "Dopo qualsiasi azione coffee, deve poter eseguire un'azione tea": `[coffee]⟨tea⟩true`

7.4 Teorema di Hennessy-Milner

Il teorema di Hennessy-Milner stabilisce una connessione profonda tra bisimilarità e equivalenza logica:

Teorema: Per processi a immagine finita (processi con un numero finito di derivati `α` per ogni azione `α`):

```
P ~ Q sse per tutte le formule ϕ di HML: P ⊨ ϕ sse Q ⊨ ϕ
```

Questo teorema stabilisce che due processi sono bisimilari se e solo se soddisfano esattamente le stesse formule HML.

7.5 Logica con ricorsione (μ-calculus)

La logica di Hennessy-Milner può essere estesa con operatori di punto fisso per esprimere proprietà temporali:

```
ϕ ::= true | ¬ϕ | ϕ ∧ ϕ | ⟨α⟩ϕ | X | μX.ϕ
```

Dove:

- `X` è una variabile
- `μX.ϕ` è il punto fisso minimo di `λX.ϕ`

L'operatore di punto fisso massimo `νX.ϕ` può essere definito come `¬μX.¬ϕ[¬X/X]`.

7.5.1 Esempi di proprietà nel μ-calculus

- "Eventualmente a": `μX.(⟨a⟩true ∨ ⟨τ⟩X)`
- "Sempre non a": `νX.([a]false ∧ [τ]X)`
- "Libertà da deadlock": `νX.(⟨-⟩true ∧ [-]X)`

8. π-calculus

8.1 Da CCS a π-calculus

Il π-calculus, sviluppato da Robin Milner, Joachim Parrow e David Walker, estende CCS con la capacità di comunicare nomi di canali. Questo permette di modellare sistemi con topologia di comunicazione dinamica.

8.2 Sintassi

```
P ::= 0 | π.P | P + P | P | P | νx.P | !P
```

Dove:

- `π` è un prefisso, che può essere:
  - `x(y)` : riceve un nome sul canale `x` e lo lega a `y`
  - `x̄⟨y⟩` : invia il nome `y` sul canale `x`
  - `τ` : azione interna
- `νx.P` : crea un nuovo nome `x` con scope `P`
- `!P` : replicazione (infinite copie parallele di `P`)

8.3 Semantica operazionale

Le principali regole di transizione includono:

- **PREFIX**: $\pi.P \xrightarrow{p} P$
- **SUM**: $P \xrightarrow{p} P'$ implica $P + Q \xrightarrow{p} P'$
- **PAR**: $P \xrightarrow{p} P'$ implica $P|Q \xrightarrow{p} P'|Q$
- **COM**: $P \xrightarrow{x(z)} P'$ e $Q \xrightarrow{\bar{x}\langle y\rangle} Q'$ implica $P|Q \xrightarrow{t} P'[y/z]|Q'$
- **OPEN**: $P \xrightarrow{\bar{y}\langle z\rangle} P'$, $x \neq y$ e $x = z$ o $x \in fn(P')$ implica $\nu x.P \xrightarrow{\bar{y}\langle \nu x\rangle} P'$
- **RES**: $P \xrightarrow{p} P'$ e $x \notin fn(\pi)$ implica $\nu x.P \xrightarrow{p} \nu x.P'$
- **REP**: $P|!P \xrightarrow{p} P'$ implica $!P \xrightarrow{p} P'$

8.4 Esempio: Telefoni mobili

Un esempio importante visto a lezione è il modello di una rete telefonica mobile, dove un telefono può spostarsi tra diverse stazioni base:

```
Telephone(id, base) = basē⟨id⟩.base(new_base).Telephone(id, new_base)
BaseStation(i) = id(tel_id).tel_id̄⟨base_j⟩.BaseStation(i)
Network = ν base₁, ..., base_n.(Telephone(id₁, base₁) | ... | BaseStation(1)
| ...)
```

9. Linguaggi di programmazione concorrenti

I calcoli di processo come CCS e π-calculus hanno influenzato la progettazione di linguaggi di programmazione con caratteristiche di concorrenza integrate.

9.1 Google Go

Go è un linguaggio di programmazione compilato con funzionalità di concorrenza integrate, progettato da Google.

9.1.1 Goroutine

Le goroutine sono thread leggeri gestiti dal runtime di Go:

```
func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
```

```go
    say("hello")
}
```

9.1.2 Canali

I canali sono condotti tipizzati per l'invio e la ricezione di valori:

```go
func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // Invia sum al canale c
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)

    x, y := <-c, <-c // Riceve dal canale c

    fmt.Println(x, y, x+y)
}
```

9.1.3 Select

Il costrutto select permette di attendere su più operazioni di comunicazione:

```go
func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}
```

9.1.4 Modelli di concorrenza

Go implementa la filosofia "Do not communicate by sharing memory; instead, share memory by communicating" ("Non comunicare condividendo memoria; invece, condividi memoria comunicando").

9.2 Erlang

Erlang è un linguaggio di programmazione funzionale con supporto integrato per concorrenza, distribuzione e tolleranza ai guasti.

9.2.1 Modello attore

La concorrenza in Erlang si basa sul modello attore:

```
% Server echo
echo() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            echo();
        stop ->
            ok
    end.

% Utilizzo
Server = spawn(fun echo/0).
Server ! {self(), "Hello"}.
receive
    {Server, Msg} ->
        io:format("Echo: ~p~n", [Msg])
end.
Server ! stop.
```

9.2.2 Robustezza

Erlang implementa la filosofia "let it crash" ("lascia che si blocchi") supportata dal collegamento e dal monitoraggio dei processi:

```
% Collegamento di processi
spawn_link(fun() ->
    % Questo causerà anche il crash del processo genitore
    1 = 2  % Errore deliberato
end).

% Monitoraggio dei processi
Pid = spawn(fun() -> timer:sleep(1000) end).
Ref = monitor(process, Pid).
receive
```

```
    {'DOWN', Ref, process, Pid, Reason} ->
        io:format("Il processo ~p è terminato: ~p~n", [Pid, Reason])
end.
```

9.2.3 Distribuzione

I processi Erlang possono comunicare tra diverse macchine in modo trasparente.

9.3 Clojure

Clojure è un dialetto di Lisp che gira sulla JVM e enfatizza la programmazione funzionale con strutture dati immutabili.

9.3.1 Memoria transazionale software

Clojure fornisce diversi tipi di riferimento per gestire lo stato mutabile condiviso:

```
;; Atom: riferimento sincrono, non coordinato
(def counter (atom 0))
(swap! counter inc)  ; Incrementa atomicamente il contatore
(reset! counter 0)   ; Imposta il contatore a 0

;; Ref: riferimento sincrono, coordinato
(def account1 (ref 1000))
(def account2 (ref 500))
(dosync
  (alter account1 - 100)
  (alter account2 + 100))
```

9.3.2 Futures e promesse

Futures e promesse forniscono un modo per lavorare con valori che potrebbero non essere ancora disponibili:

```
;; Future: esegue un task in un thread separato
(def f (future
         (Thread/sleep 1000)
         (+ 1 2)))
;; Blocca fino a quando il risultato non è disponibile
(deref f)  ; o @f, ritorna 3

;; Promise: un segnaposto per un valore da consegnare più tardi
(def p (promise))
(def t (future
         (Thread/sleep 1000)
         (deliver p 42)))
```

```
;; Blocca fino a quando un valore viene consegnato
(deref p 2000 :timeout)  ; Ritorna 42, o :timeout se ci vuole > 2 secondi
```

9.4 Modelli di concorrenza a confronto

- **Go**: Communicating Sequential Processes (CSP) con canali e goroutine
- **Erlang**: Modello attore con processi e passaggio di messaggi
- **Clojure**: Memoria transazionale software con programmazione funzionale

Ogni modello ha i suoi punti di forza:

- CSP è buono per task ad alto throughput, vincolati dalla CPU
- Il modello attore eccelle in sistemi distribuiti, tolleranti ai guasti
- STM funziona bene con lo stato condiviso in un contesto funzionale

10. Conclusioni

Questo corso ha coperto sia i fondamenti teorici della concorrenza che i suoi aspetti pratici nei linguaggi di programmazione moderni:

- **Teoria**: Calcoli di processo (CCS, π-calculus), bisimulazione, teoria dei punti fissi, logiche modali
- **Pratica**: Linguaggi di programmazione (Go, Erlang, Clojure) con funzionalità di concorrenza integrate

La connessione tra teoria e pratica è evidente in come i calcoli di processo hanno influenzato la progettazione dei linguaggi di programmazione concorrenti:

- CSP → Go
- Modello attore → Erlang
- Approcci funzionali → Clojure

Comprendere i fondamenti teorici aiuta a ragionare sui programmi concorrenti e a evitare problemi comuni come race condition, deadlock e livelock. I linguaggi pratici forniscono meccanismi efficienti ed espressivi per implementare sistemi concorrenti nelle applicazioni del mondo reale.