PCD Module 1.1

Concurrent Programming - Overview

Concurrent programming

- **Concorrenza**: proprietà dei sistemi in cui diversi processi computazionali vengono **eseguiti nello stesso istante**, potenzialmente **interagendo** tra di loro.
- **Programmazione concorrente**: costruire programmi in cui più attività si *sovrappongono* nel tempo e *interagiscono* in qualche modo.
- **Programmi concorrenti**: insieme finito di programmi sequenziali (**processi**) eseguibili in parallelo (*processi paralleli*)
- **Tipologie di programmazione**
    - *Concorrente*: v. sopra. Non necessariamente i programmi vengono eseguiti in processori fisici separati. Focus sull'organizzazione, livello logico/astratto/programmazione.
    - *Parallela*: esecuzione sovrapposta su processori fisici separati. Focus sulle performance, livello fisico.
    - *Distribuita*: processori distribuiti su una rete, nessuna memoria condivisa.
- Concurrent vs. Parallel, pg. 6
- **Paradigmi**
    - *Programmazione multi-threaded*: stato condiviso, meccanismi di sync (sem_t, monitor)
    - *Programmazione basata sui messaggi*: no stati condivisi, interazione tramite scambio messaggi, sync/async
    - *Programmazione guidata dagli eventi*: flusso del programma determinato dagli eventi (azioni, sensori, messaggi da altre app)
    - *Programmazione asincrona*: progettare programmi con azioni e richieste async (never blocking dogma, future mechanisms & callbacks)
    - *Programmazione reattiva*: il flusso è progettato attorno a flussi di dati e alla propagazione dei cambiamenti

Architetture parallele

- **Multi-core**: più core sullo stesso chip, condividendo RAM ed eventualmente livelli di cache
- **Core eterogenei**: processore standard + processori *attached* specializzati (es. CPU e GPU, CUDA)
- **Supercomputer**: labs + grandi aziende, grande numero di processori e architetture diverse.
- **Clusters/grid**: composto da più computer interconnessi (es. server Apple). Memoria e risorse non condivise, comunicazione tramite scambio messaggi.
- **Cloud computing**: potenza di calcolo fornita come servizio in rete: risorse, software e informazione condivise. SAAS, PAAS e IAAS; cloud private/pubbliche, esempi (EC2, Azure, AppEngine).

Tassonomia di Flynn

Categorizzazione dei sistemi basata sul numero dei flussi di informazioni e dei flussi dati:

- **SISD**: modello Von Neumann, singolo processore.
- **SIMD**: singolo flusso di istruzioni trasmesse concorrentemente a più processori, ognuno con il proprio flusso dati.
- *MISD*: teorico.
- **MIMD**: ogni processore ha il suo flusso di informazioni operante su un flusso dati.
    - *a memoria condivisa*: tutti i processi condividono un singolo spazio degli indirizzi e comunicano su variabili condivise
        - *SMP - architettura simmetrica multiprocesso*: condivisione connessione a memoria condivisa + accesso alla stessa velocità
        - *NUMA - accesso alla memoria non uniforme*: memoria condivisa, alcuni blocchi di memoria possono essere fisicamente più vicinamente associati a certi processori
    - *a memoria distribuita*: ogni processo ha il proprio spazio degli indirizzi e comunica con gli altri tramite scambio messaggi
        - *MPP - processori massivamente paralleli*: infrastrutture altamente specializzate e accoppiate, scalabili (HPC)
        - *Clusters*
        - *Grid*: sistemi che usano risorse eterogenee distribuite tramite LAN/WAN senza un punto comune di amministrazione.

Motivazioni

- *Miglioramento delle performance*: aumento produttività e reattività. Misura: **speedup** ($S = \frac{T_1}{T_N}$)
    - **Legge di Amdahl**: massimo speedup nella parallelizzazione di un sistema $S = \frac{1}{1-P+\frac{P}{N}}$.
    - significa che serializzazione/sequenzializzazione sono un danno alle performance, ma a volte necessarie per la correttezza --> sforzi e compromessi
    - senza dimenticarsi dell'efficienza, $E = \frac{S}{P}$
    - *collo di bottiglia*: la memoria. Solo 1 operazione di memoria per volta --> importanza della cache
- *Progettazione e astrazione*: definizione livello di astrazione dei programmi che interagiscono con l'ambiente.
    - Concorrenza come strumento per la progettazione e la costruzione.
    - Entrambi influenzano l'intero spettro ingegneristico (modellazione, progettazione, implementazione, verifica, testing)

Gergo

- **Processo**: un programma sequenziale in esecuzione, unità base di un sistema concorrente, singolo thread logico. Unità di lavoro, task, concetto astratto/generale
    - *indipendenza dalla velocità*: l'esecuzione è completamente asincrona, non si possono fare previsioni sulla velocità --> **nondeterminismo**.
- **Interazione**: ogni programma concorrente è basato su processi multipli che hanno bisogno di interagire, in qualche modo

- *Cooperazione*: interazioni previste e ricercate, parte della semantica del programma concorrente
    - *Comunicazione*: realizzazione flusso di informazioni tra processi, tipicamente realizzato in termini di messaggi
    - *Sincronizzazione*: definizione esplicita di relazioni temporali/dipendenze tra processi e azioni. Introduzione supporto allo scambio di segnali temporali.
- *Competizione*: interazioni previste e necessarie, ma non ricercate. Necessità di coordinamento dell'accesso da processi multipli a risorse condivise.
    - *Mutex*: regolazione accesso a risorse condivise da processi distinti
    - *Sezioni critiche* : regolazione esecuzione concorrente di blocchi di azioni da processi distinti
        - Differenze
        - Esempio del latte + soluzione "note nel frigo"
- *Interferenza*: interazione inaspettate e non necessarie, producono cattivi effetti solo con determinati valori. **heisenbugs**
    - *Race conditions*: 2+ processi accedono e aggiornano concorrentemente risorse condivise. Il risultato del singolo aggiornamento dipende dall'ordine di accesso. Causato da cattiva gestione delle interazioni previste e presenza di interazioni spurie disattese nel problema.
- *Situazioni critiche*
    - *Deadlock*: 2+ azioni aspettano l'altra per finire, e nessuno lo fa (rilascio di risorse condivise bloccate)
    - *Starvation*: processo bloccato in attesa infinita
    - *Livelock* : simile a deadlock, ma lo stato cambia costantemente in accordo con l'altro

Storia

- Motivazione originaria: *sviluppare OS affidabili*
- 1961: nascita multiprogrammazione (interrupt), ASM
- 1965-1979: concetti fondamentali per PCD
- 1970s: primi libri su PCD

Linguaggi, meccanismi e astrazioni concorrenti

- Per eseguire un programma concorrente, serve una **macchina concorrente**
    - che fornisce:
        - supporto all'esecuzione di programmi concorrenti
        - processori virtuali
        - **meccanismi base**
- Meccanismi base:
    - **multiprogrammazione**: meccanismi che rendono possibile creare nuovi processori virtuali e allocarci processori fisici a basso livello tramite scheduling.

    - **sincronizzazione e comunicazione**
        - *modello a condivisione di memoria* tra processori virtuali (es. multithreading)
        - *modello a scambio di messaggi*: nessuna memoria fisica, scambio di messaggi tra processori

- Per descrivere un programma concorrente, serve un **linguaggio concorrente**
    - programmi organizzati come set di processi sequenziali eseguiti concorrentemente sui processori virtuali
    - costrutti base per specificare
        - **concorrenza** (processi multipli)
        - **interazione dei processi** (sync, comunicazione, mutex)

Linguaggi: approcci progettuali

- linguaggio sequenziale + libreria con primitive concorrenti (es. C + PThread)
- linguaggio progettato per la concorrenza (es. OCCAM, ADA, Erlang)
- approccio ibrido
    - paradigmi sequenziali estesi con supporto nativo per la concorrenza (es. Java, Scala)
    - librerie e pattern basati su meccanismi base (es. `java.util.concurrent`)

Notazioni base e costrutti

fork/join ('60/'70)

- **fork**: comportamento simile a invocazione procedure, con differenza che un nuovo processo è creato e attivato per eseguire la procedura. Input: procedura da eseguire, output: ID del processo creato. Biforcazione del flusso del programma: il processo figlio è eseguito async.
- **join**: intercetta processi creati con fork che sono terminati e sincronizza il flusso di controllo.
- pro: generale e flessibile
- contro: basso livello di astrazione, programmi difficili da leggere, nessuna rappresentazione esplicita del processo di astrazione

cobegin/coend ('60/'70)

Concorrenza "a blocchi": le istruzioni all'interno di un blocco **cobegin/coend** sono eseguite in parallelo, terminando solo quando tutti i componenti (processi) hanno finito.

- pro: disciplina più forte nello strutturare un programma concorrente rispetto a fork/join. Programmi più leggibili.
- contro: meno flessibilità di fork/join. Non c'è astrazione esplicita che incapsula il processo.

Linguaggi con supporto di prima classe per i processi

Processi come "entità di prima classe" del linguaggio, "moduli" per organizzare il programma e il sistema. Incapsulazione esplicita del flusso di controllo. Esempi: Concurrent Pascal, OCCAM, ADA, Erlang

Concorrenza nei linguaggi popolari

- I linguaggi popolari (C, Java, ecc.) supportano la creazione e l'esecuzione dei processsi tramite librerie.
- Supporto per *programmazione multithreading*: thread come implementazione di processo lightweight - non da

confondere con i processi di sistema.
- Esempi: multithreaded programming in Java, Pthread in C/C++

Programmazione multithreaded in Java

- Primo linguaggio popolare a fornire supporto nativo a programmazione concorrente, approccio conservativo.
- Processo implementato come **thread**, con mapping diretto sui thread dell'OS.
- Supporto esteso dal 2005 tramite *Java Concurrency Utility*

Threads in Java

- Modello
  - singolo flusso di controllo che condivide la memoria con tutti gli altri thread (stack privato, heap in comune)
  - ogni programma Java contiene almeno 1 thread
  - altri thread possono essere creati e attivati dinamicamente a runtime

- Definizione
  - oggetti che estendono la classe Thread fornita in `java.lang.Thread`

- Esecuzione
  - gli oggetti possono essere istanziati e generati invocando il metodo `start()`

Programmazione multithreaded in C/C++ + Pthreads

- Fornisce primitive nbasilari per programmazione multithreaded in C/C++
- Processo implementato come thread; corpo del processo specificato per mezzo di procedure
- Lo standard definisce interfacce e specifiche, non l'implementazione - la quale dipende dagli OS.

Funzioni

- Interfaccia definita in `pthread.h`
- Tipi di dato:
  - `pthread_t` - thread identifier data type
  - `pthread_attr_t` - data structure for thread attributes

- Funzioni principali:
  - creazione del thread (fork): `pthread_create(pthread_t* tid, pthread_attr_t* attr, void* (*func)(void*), void* arg)` e `pthread_attr_init(pthread_attr_t*)`
  - terminazione del thread: `pthread_exit(int)`
  - congiunzione: `int pthread_join(pthread_t thread, void ...)`

Oltre ai thread: uno scenario ricco

- Framework basati sui **task** (es. Java Executor)
- **Programmazione asincrona** (modelli event-loop, AsyncTasks)
- **Attori** (OOP concorrente, variante ad oggetti attivi)
- **Scambio di messaggi sincrono** (processi + s.m.s.)

- **Programmazione reattiva** (es. programmazione reattiva funzionale, estensioni reattive - Rx)
- **Calcolo parallelo** (GPGPU Programming - OpenCL, Lambda Architecture/Paralellism for BigData - MapReduce)

Modello ad attori

- Scambio messaggi asincrono tra oggetti reattivi chiamati attori
  - tutto è un attore, con ID unico e unica mailbox
  - ogni interazione ha luogo come scambio messaggi asincrono

- Poche primitive: **send**, **create**, **become**
- Sviluppato tra gli anni 70 e 90
- Diversi framework: Akka, Erlang, HTML5 Web Workers, DART Isolates

Concorrenza logica vs. fisica

I paradigmi di programmazione concorrenti moderni promuovono una visuale logica sulla concorrenza, ad alto livello. Si possono così avere migliaia di attori in esecuzione sulla stessa macchina.

Oggetti attivi

Concorrenza + OO. Oggetti attivi + passivi, creazione implicita dei thread + meccanismi di sincronizzazione.

PCD Module 2.1

Modellare l'esecuzione di un programma concorrente

Creazione dei modelli di programmi concorrenti

Dai programmi ai modelli e ritorno

- Importanza dei modelli e dell'astrazione:
  - modello: descrizione rigorosa / rappresentazione della struttura del programma ad un certo livello di astrazione
  - rappresentazioni diagrammatiche per la progettazione di programmi
  - modelli formali per l'analisi e la verifica dei programmi

- Definire modelli appropriati per programmi concorrenti
  - astraendo dai dettagli di basso livello nell'attuale implementazione e realizzazione
  - abilitando la possibilità di ragionare sul comportamento dinamico dei programmi concorrenti

Un modello per l'esecuzione di programmi concorrenti

- Modellare ogni processo come una sequenza di azioni **atomiche**: 1 azione = 1 istruzione
- Presupposto di indipendenza della velocità: modellare esecuzione concorrente come sequenza di azioni ottenuta sovrapponendo arbitrariamente le azioni atomichedai processi.
  - Un singolo processore globale astratto che esegue tutte le azioni
  - Istruzioni atomiche => eseguite al completamento senza possibilità di sovrapposizione
  - Durante la computazione del control pointer o istruzione che indica la prossima istruzione che può essere eseguita da quel processo

- una **computazione/scenario** è un'esecuzione di una sequenza che può accadere come risultato della sovrapposizione.

Slide pg. 5: esempio di azioni atomiche

Diagramma a stati e scenari di esecuzione

Dato un modello, l'esecuzione di un programma concorrente può essere formalmente rappresentata da:

- **stati**: tupla (1 elemento di ogni processo che è etichetta di tale processo)
- **transizioni** tra stati (se l'esecuzione in s1 cambia lo stato in s2>

- in un **diagramma a stati**: grafo contenente tutti gli stati raggiungibili dal programma. Gli scenari sono rappresentati da percorsi direzionati nel diagramma a stati. I cicli rappresentano possibili computazioni infinite.

Esempio di diagramma a stati pg. 7-15

Numero di scenari prodotti da n processi, ognuno con m_i azioni: $ns = \frac{(\sum_{i=1}^{n} m_i)!}{(\prod_{i=1}^{n} m_i)!}$

L'importanza di essere atomici

Assegnazioni e incrementi a livello codice macchina

Strutture non atomiche

"Atomiche" possono essere anche le strutture dati, se esse possono essere in un numero finito di stati uguali al numero di valori che possono assumere, tipicamente tipi primitivi in linguaggi concorrenti (non lo è double in java, e.g.). ADT composti da oggetti dati semplici multipli sono tipicamente nonatomici (es. classi OO).

In tal caso, negli ADT si possono verificare due tipi di stati:

- **internal**: significativo per chi definisce l'oggetto (classe)
- **external**: significativo per chi usa i dati

La corrispondenza è parziale: esistono stati interni che non hanno corrispondenze esterne. Stati interni con corrispondenze esterne sono definiti **consistenti**.

L'esecuzione di una operazione su un ADT non atomico può passare tramite stati non consistenti. Ciò non è problema in caso di programmi sequenziali, ma lo è nei programmi concorrenti. È quindi necessario introdurre meccanismi adatti che garantiscano che i processi lavorino su oggetti già in stato consistente.

Sovrapposizione arbitraria

L'intreccio arbitrario significa che ignoriamo il tempo nella nostra analisi dei programmi concorrenti, concentrandoci solo su

- ordini parziali relativi alle sequenze di azioni a1, a2, ...
- l'atomismo dell'azione individuale: scegliere ciò che è atomico è fondamentale

Robustezza rispetto all'hardware (processore) e alle modifiche del software, indipendente dalle variazioni dei tempi / prestazioni. Questo rende i programmi concorrenti suscettibili di **analisi formale**, che è necessaria per garantire la correttezza dei programmi concorrenti. - dimostrare la correttezza oltre al tempo effettivo di esecuzione, che è tipicamente strettamente dipendente dalla velocità del processore e dai tempi di gestione del sistema

Correttezza

Correttezza dei programmi

- Verifica della correttezza dei programmi sequenziali: **unit tests** basati su input specificato e aspettando qualche output specificato. Diagnosticare, correggere, ripetere il ciclo... ma rieseguire un programma con lo stesso ingresso dà sempre lo stesso risultato
- Nuova prospettiva impegnativa nella Programmazione concorrente: lo stesso ingresso può fornire diverse uscite, a seconda dello scenario. Alcuni scenari possono dare un'uscita corretta mentre altri non possiedono un debug di un programma concorrente nel modo normale. Ogni volta che si esegue il programma probabilmente avremo un diverso scenario.
- Necessità di diversi approcci: analisi formale, controllo dei modelli. Basati su modelli astratti

Correttezza dei programmi concorrenti

La correttezza di programmi (eventualmente non terminanti) concorrenti è definiti in termini di proprietà dei calcoli - condizioni che devono essere verificate in tutti i possibili scenari. Due tipi di proprietà di correttezza:

Proprietà di sicurezza

La proprietà deve essere sempre vera, cioè per una proprietà di sicurezza P da tenere, deve essere vero in ogni stato di ogni calcolo. Espresse come invarianti di un calcolo. Usato tipicamente per specificare che le "cose cattive" non dovrebbero mai accadere:

- mutex
- no deadlock
- ...

Proprietà di vivacità

La proprietà deve eventualmente diventare vera, cioè per una proprietà di vivacità P da tenere, deve essere vero che in ogni calcolo esiste in uno stato in cui P è vero. Usato tipicamente per specificare che "cose positive" accadranno

- no starvation
- no dormancy
- comunicazione affidabile

Imparzialità

Una proprietà di vivacità che detiene che qualcosa di buono accade infinitamente spesso. Esempio

principale: un processo attivato infinitamente spesso durante un'esecuzione dell'applicazione, ogni processo ottenendo un giusto andamento. Cioè un'azione che può essere eseguita, alla fine verrà eseguita.

Quindi i programmi possono avere diversi comportamenti di abilità a seconda di come vengono interlevate le loro istruzioni. Come le istruzioni sono interlevate è il risultato di una politica di pianificazione: **unconditional / weak / strong fairness**.

PCD Module 2.2

Il problema della sezione critica

N processi, ognuno esegue in un loop infinito una sequenza di istruzioni che può essere divisa in 2 sottosequenze, CS e non-CS (NCS):

```
loop {
  NCS
  entry protocol
  CS
  exit protocol
  NCS
}
```

Ogni sezione critica è una sequenza di istruzioni che accede ad un oggetto condiviso.

Proprietà delle CS

Obiettivo: progettare *entry* e *exit protocol* che soddisfino:

- **mutex**: le istruzioni dai 2+ processi non devono essere sovrapposte
- **libero da deadlock**: se un processo prova di entrare in CS, un processo deve per forza avere successo
- **libero da starvation individuale**: se un qualsiasi processo prova ad entrare in CS, quel processo deve aver successo

Ogni soluzione proposta deve soddisfare anche la **proprietà progressa** delle CS: una volta che un processo inizia l'esecuzione di istruzioni nella CS, deve finire. Ciò non è richiesto nella NCS.

Problemi concreti basati sulle CS

Il problema della CS serve per sviluppare modelli che eseguono computazioni complesse, ma occasionalmente hanno bisogno di accedere a dati o risorse condivise da più processi (es. chiosco check-in aeroporto).

Algoritmi vs. Meccanismi

CS può essere risolto tramite:

- **meccanismi**, es. semafori o blocchi (piu avanti)
- **algoritmi** (questo modulo), utilizzando solo istruzioni base atomiche (load, store)

CS di due processi: tentativi

- 1° tentativo: `await turn`, istruzione che attende la fine dell'istruzione `turn`. Implementabile tramite busy-wait loop.
    - Studio della correttezza dell'algoritmo, riduzione stati
    - Starvation non soddisfatta
- 2° tentativo: variabili globali che indicano quando un processo è nella sua CS
    - mutex non soddisfatta, raggiungibile a causa della nonatomicità di pre e post protocol.
- 3° tentativo: si considera `await` parte della CS, si spostano gli assignment prima dell' `await`
    - mutex risolta ma possono accadere deadlock (livelock)
- 4° tentativo: richiedere che un processo si arrenda ad entrare nella CS se vede che è contesa da altri processi
    - deadlock risolta, ma c'è possibilità di starvation in caso di perfetta sovrapposizione

Algoritmi

Algoritmo di Dekker

Combo 1° + 4° tentativo. Si richiede che il *diritto di insistere ad entrare* sia esplicitamente passato tra processi per mezzo della variabile `turn`. È corretto: soddisfa mutex ed è libero da deadlock e starvation.

Algoritmo di Peterson

Più conciso, collassa le due istruzioni `await` in una con una condizione composta.

Istruzioni atomiche composte

Il problema della CS può essere semplificato se si sfruttano istruzioni atomiche più complesse fornite dalla macchina concorrente, es. *test-and-set*, *exchange*, *fetch-and-add*, *compare-and-swap*. Ad esempio, *test-and-set* è un'istruzione atomica definita come l'esecuzione di `test-and-set(x,r): <r:=x, x:=1>` senza possibilità di sovrapposizione. Notazione tipica: `< ... >` tra gruppi di istruzioni.

Blocchi

Sfruttando istruzioni atomiche composte si può facilmente realizzare un meccanismo di blocco basilare.

Due fasi: acquisizione e rilascio del blocco. Pg. 21

Istruzioni atomiche in Java: `synchronized`

Istruzioni atomiche possono essere implementate in Java mediante l'uso di `synchronized`. In questo caso le sequenze non si sovrappongono.

Soluzione con ticket: l'Algoritmo di Bakery (tie-breaker)

Introduzione di un ticket per stabilire il turno di un processo. Possibili overflow aritmetici di `turn` e `num`.

PCD Module 2.3

Verifica di programmi concorrenti - intro

Verifica e testing

Terminologia

- **falla**: manifestazione di un errore (azione che produce un risultato errato)
- **guasto**: quando una falla si verifica, può causare un guasto. Un guasto può essere causato da molti errori, una falla può causare molti guasti. I guasti sono relativi alle specifiche.
- **testing**: attività volta alla ricerca di guasti (e falle rilevanti)
- **verifica**: controllare se il sistema soddisfa le specifiche
- **validazione**: controllare se il sistema soddisfa le aspettative del cliente

Tipologia di test

- **test di sicurezza**: verificano che un comportamento di una classe o del sistema sia conforme alla sua specifica, di solito assumendo la forma di invarianti di test, in genere utilizzando le assertion.
- **test di vitalità**: test di progresso / non-progresso, molto difficile da impostare

Problemi e limiti di test:

- in genere controlla solo un sottoinsieme degli scenari
- fenomeno "heisenbugs": il codice di prova può introdurre manufatti temporali o sincronizzazione che possono mascherare gli errori

Testing vs verifiche

- Il *testing* è volto a verificare che una proprietà (correttezza) vale per un certo scenario selezionato: il test rivela la presenza di errori, non la loro assenza
- La *verifica* è volta a verificare che una proprietà vale per tutti i possibili scenari (tracce). Necessita di tecniche formali (es. model checking)

Metodi formali

- **model checking**: la verifica è eseguita generando uno per uno gli stati del sistema e controllando le proprietà stato per stato. Possono essere automatizzati tramite appositi strumenti.
- **prove di invarianza induttive**: proprietà invarianti sono provate per induzione sugli stati del sistema. Possono essere automatizzate tramite strumenti chiamati sistemi deduttivi.

Entrambe le tecniche si appoggiano su un qualche tipo di linguaggio formale/calcolo per specificare la correttezza delle proprietà.

Proprietà di correttezza nel calcolo delle proposizioni

Con il calcolo proposizionale, le proprietà di correttezza sono espresse come *formule logiche* che devono essere vere per verificare la proprietà in un qualche stato del sistema. Tali formule sono asserzioni ottenute componendo proposizioni tramite connettori logici (and/or/not/implicazioni/equivalenze).

Nel nostro caso, le proposizioni riguardano i *valori delle variabili* e i *puntatori di controllo* durante l'esecuzione di un programma concorrente. Ad esempio, data una variabile booleana `wantp`, una proposizione atomica (*asserzione*) `wantp` è vera in un certo stato se e solo se il valore dell variabile `wantp` è vera in tale stato.

Ogni etichetta di un'istruzione del processo sarà usata come proposizione atomica la cui interpretazione è "il puntatore di controllo di tale processo è attualmente a quella etichetta".

Esempio: mutua esclusione

```
bool wantp = false
bool wantq = false

p                       | q
----------------------------------------
loop forever            loop forever
p1: non-CS          q1: non-CS
p1: wantp = true        q2: wantq = true
p3: await !wantq        q3: await !wantp
p4: CS                  q4: CS
p5: wantp = false   q5: wantq = false
```

La formula $p_4 \wedge q_4$ è vera se entrambi i puntatori di controllo del processo sono all'interno della CS. Se esiste qualche stato in cui questa formula è vera, allora significa che la proprietà di mutex non è soddisfatta. Allo stesso modo, un programma soddisfa la proprietà mutex se la formula $\neg(p_4 \wedge q_4)$ è vera per ogni possibile stato di ogni scenario.

Logica temporale

Processi e sistemi cambiano il loro stato nel tempo, e quindi anche l'interpretazione delle formule riguardanti i relativi stati possono cambiare nel tempo. C'è bisogno di un linguaggio formale/calcolo che può prendere in considerazione tale aspetto. La *logica temporale* è uno dei principali e popolari.

La **logica temporale** è una logica formale ottenuta aggiungendo operatori temporali alla logica proposizionale o dei predicati:

- **Linear Temporal Logic (LTL)**: per esprimere proprietà che devono essere vere in un certo stato per ogni possibile scenario.
- **Branching Temporal Logics**: per esprimere proprietà che devono essere vere in alcuni o tutti gli scenari partendo da uno stato, esempio **CTL (computational tree logic)**

LTL: operatori temporali

LTL è basato su due operatori temporali di base:

- $\Box$ **sempre**: la formula è vera in uno stato s_i di una computazione se e solo se è vera in *tutti* gli stati $s_j, j \geq i$. Sinonimo: G (globally). Può essere usato per specificare *proprietà di sicurezza* (sempre vere).
- $\Diamond$ **eventualmente**: la formula è vera in uno stato s_i di una computazione se e solo se è vera in *alcuni* stati $s_j, j \geq i$. Sinonimo: F (finally). È usato per specificare *proprietà di vitalità* (qualcosa che potrebbe essere vero).

Proprietà base

- **Riflessività**: $\Box A \rightarrow A, A \rightarrow \Diamond A$
- **Dualità**: $\neg\Box A = \Diamond\neg A, \neg\Diamond A = \Box\neg A$
- **Sequenze di operatori**: $\Diamond\Box A, \Box\Diamond A$

Deduzioni con logiche temporali

La LT è un sistema formale di logica deduttiva con suoi assiomi e regole di inferenza. Può essere usato per formalizzare la semantica di programmi concorrenti e usata per provare rigorosamente le proprietà di correttezza dei programmi.

- Esempi:
  - $(\Diamond\Box A1 \wedge \Diamond\Box A2) \rightarrow \Diamond\Box(A1 \wedge A2)$ è vera
  - $(\Box\Diamond A1 \wedge \Box\Diamond A2) \rightarrow \Box\Diamond(A1 \wedge A2)$ è falsa

Specificare proprietà di sicurezza

$\Box P, P = \neg$, Q è la descrizione di uno stato incorretto.

Esempio: proprietà di mutua esclusione ($\Box\neg(p_3 \wedge q_3)$)

```
int turn = 1

p                       q
---------------------------------
loop forever            loop forever
p1: non-CS          q1: non-CS
p2: await turn = 1  q2: await turn = 2
p3: CS                  q3: CS
p4: turn = 2            q4: turn = 1
```

Specificare proprietà di vitalità

$\Diamond P$, dove P è la descrizione di un caso buono.

Esempio: proprietà di progresso (assenza di *starvation*) -- codice uguale a prima

- Proprietà di progresso per lo stato attuale: $p_2 \rightarrow \Diamond p_3$
- Proprietà di progresso per tutti gli stati: $\Box(p_2 \rightarrow \Diamond p_3)$

Operatore prossimo (next)

$\circle\phi$ è vero se la proprietà ϕ rimane nel prossimo stato, in tutti i casi possibili. Sinonimo: X (neXt)

Operatori binari

- $A \union B$ **fino a** (until): la formula è vera in uno stato s_i se e solo se
  - B è vero in *qualche* stato $s_j, j \geq i$, e
  - A è vero in *tutti* gli stati $s_k, i \leq k < j$
  - ovvero: finchè B è vera, A deve essere vera fino a che ciò succede ??
- AWB **fino a-debole**: come Until, ma non è richiesto che la formula B diventi eventualmente vera. Se non lo fa, A rimane vera indefinitamente (finchè A è falso, B deve essere vero).

sempre ed **eventualmente** possono essere definiti in termini **fino a**:

- $\Diamond A = true \union A$
- $\Box A = \neg\Diamond(\neg A)$

Sorpasso (overtaking)

Esempio: `try-p,{try-q,CSq,try-q,CSq,...,CSq} (x1000),CSp`

- Non è un esempio di starvation, ma è vero che $\Diamond CSp$ ed è evidente che la libertà da starvation può essere una proprietà molto debole
- In alcuni casi abbiamo bisogno di assicurarci che un processo possa entrare nella sua CS in una quantità di tempo ragionevole.

Proprietà di sorpasso k-delimitata

Dal momento in cui un processo p prova ad entrare nella sua CS, un altro processo può entrare al massimo k volte prima che p lo faccia.

Esempio: 3-overtaking `try-p,try-q,CSq,try-q,CSq,try-q,CSq,CSp`

La proprietà può essere espressa dall'operatore **fino a-debole**, es. 1-bounded overtaking:

`try_cs(P) => not cs(Q) W cs(Q) W not cs(Q) W cs(P) (with pars: try_cs(P) => not cs(Q) W (cs(Q) W (not cs(Q) W cs(P)))`

Tecniche di verifica

Model-checking

- È la tecnica più importante ed usata per controllare automaticamente la correttezza delle proprietà di un sistema concorrente.
- Strategia basata sulla ricerca esaustiva nell'intero spazio degli stati di un sistema e verificare se una certa proprietà è soddisfatta

    - Proprietà come predicati di uno stato del sistema o di stati, espressi come specifiche logiche ad esempio le formule di logica temporale proposizionale
    - Se il sistema soddisfa le proprietà, il model checker genera una risposta di **conferma**, altrimenti produce una **traccia** (*trace*, controesempio)

- MC può essere applicato anche all'hardware, es. NASA

Affrontare il problema dell'esplosione dello stato degli spazi

Il grande problema con le tecniche di model checking è la dimensione dello spazio degli stati. Tecniche principali:

- applicando regole di riduzione del numero di stati
- costruzione incrementale del grafo
- model checking simbolico

Model checker: SPIN + linguaggio PROMELA

Java Path Finder

MC specializzato nella verifica di programmi scritti in Java. Scritto dalla Nasa. È una JVM che esegue programmi in tutti gli scenari possibili (percorsi di esecuzione), controllando deadlock, eccezioni non gestite, errori...

Prove induttive di invarianza

- **Invarianza**: formula che deve essere invariabilmente vera in un qualsiasi punto di una qualsiasi computazione (es. $\neg(p_4 \wedge q_4)$)
- Provate tramite **induzione** su tutti gli stati di tutte le computazioni. Per provare che A è un'invariante, si prova che A è vero nello stato iniziale e si assume che A sia vero in uno stato generico S e si prova che A è vera in tutti i possibili stati prossimi a S.
- Sistemi deduttivi

Verifica delle proprietà di sicurezza e vitalità

- Proprietà di sicurezza: facile da verificare, deve essere vera in tutti gli stati. È sufficiente trovare uno stato che non verifica la proprietà per completare la verifica.
- Proprietà di vitalità: non è sufficiente controllare gli stati uno per uno, ma tutti i possibili sceenari. è perciò piu complesso.

Logica temporale delle azioni (TLA/TLA+)

Usato per specificare e verificare sistemi complessi nel mondo reale (es. Amazon).

Una specifica TLA+ descrive l'insieme di tutti i possibili comportamenti legali di un sistema. Può essere usato per descrivere le proprietà di correttezza desiderate nel sistema e il design del sistema.

Obiettivo: rendere possibile dimostrare che la progettazione corretta di un sistema implementa le proprietà di correttezza desiderate, tramite strumenti come TLC model checker.

PlusCal, linguaggio algoritmico basato su TLA+ per scrivere algoritmi.

PCD Module 2.4

Coordinamento dei processi: meccanismi base

Semafori

- Funziona come un normale semaforo: blocca e sblocca l'esecuzione di processi in base alle necessità.
- Tipo di dato primitivo fornito dalla macchina concorrente

Tipo di dato

- tipo di dato composto con due campi:
  - **S.V**, int >= 0 (inizializzato con k >= 0)
  - **S.L**, set di processi (id) (inizializzato con set vuoto)
- operazioni atomiche:
  - **wait(S)** (P(S)). Definizione, con p=processo che esegue wait:
    - se S.V > 0, S.V-- (semaforo verde)
    - altrimenti, S.L aggiunge p. p.state = blocked (semaforo rosso, processo bloccato)
  - **signal(S)** (V(S)). Definizione:
    - se S.L vuoto (nessun processo in attesa), S.V += 1
    - altrimenti, rimuove dalla lista e imposta a ready (sbloccato) un elemento arbitrario di S.L

Invarianti del semaforo

k = valore iniziale di S.V, `#signal(S)` n° di signal(S) eseguiti e `#wait(S)` n° di wait eseguite.

Un processo che è bloccato mentre esegue wait(s) non esegue con successo l'istruzione.

Teorema: un semaforo S soddisfa le seguenti invarianti: - `S.V >= 0` - `S.V = k + #signal(S) – #wait(S)`

Tipi di semaforo

- **Mutex** (binari), con S.V = {0,1}
- **General** (contatori), con S.V di valore qualsiasi >= 0
- **Event**, inizializzati a 0, con scopo di sincronizzazione

Definizioni

- **strong**: S.L non è un set ma una **coda**
  - non c'è starvation (impossibile)
- **weak**: S.L è un **set**
- **busy-wait**: non c'è S.L. Sono ancora atomici, non c'è interleaving tra le due istruzioni: `wait(S) = await(S.V > 0); S.V -= 1` e `signal(S) = S.V += 1`.
  - Può esserci starvation: non è scontato che che un processo riesca ad entrare nella sezione critica.
  - Appropriati in sistemi multiprocessore, quando il processo che attende ha il suo processore e non consuma tempo di CPU che può essere usat per altri calcoli.

Utilizzo

I semafori possono essere usati come blocchi a basso livello per risolvere quasi ogni problema riguardante l'interazione tra processi, tra cui **mutex** (critical section) e **sincronizzazione** (event semaphore/barriers).

CS con semafori

Usando un semaforo, la soluzione al problema della CS con 2 processi è triviale, usando un semaforo come **lock**.

```
CS con semafori: 2 processi
sem_bin S = (1, {})
{P, Q}
loop forever
1: NCS
2: wait(S)
3: CS
4: signal(S)
```

Correttezza: costruendo il diagramma a stati ridotto e controllando le proprietà può essere verificato che la soluzione del semaforo al problema della CS è corretto: non c'è mutex ed è libero da deadlock e starvation. Diagramma a stati pg. 15

```
CS con semafori: 2 processi (abbreviato)
sem_bin S = (1, {})
{P, Q}
loop forever
1: wait(S)
2: signal(S)
```

CS con N processi: stessa soluzione applicata a N processi, non c'è più libertà da starvation.

Utilizzo di semafori per la sincronizzazione

I semafori forniscono meccanismi base per sincronizzare i processi, risolvendo così problemi relativi all'ordine di esecuzione.

Semafori ad eventi: usati per inviare/ricevere un segnale temporale. Inizializzati a (0, {}). Esempio merge sort:

```
sem_bin S1 = (0,{})
sem_bin S2 = (0,{})
int[] A = []

sort1                    sort2                    merge
p1: sort 1st half of A   q1: sort 2nd half of A   r1: wait(S1)
p2: signal(S1)           q2: signal(S2)           r2: wait(S2)
                                                  r3: merge halves of A
```

Problemi comuni

Producer/consumer

- **producer**: esegue `produce()` per creare un elemento `data` che viene poi inviato al processo consumer
- **consumer**: al ricevimento di un elemento dal producer, esegue `consume(data)`.

Quando un elemento viene inviato da un processo a un altro, la comunicazione può essere:

- *sincrona*: la comunicazione non avviene finchè P e C non sono pronti
- *asincrona*: il canale di comunicazione ha della capacità di archiviare dei dati.
    - La separazione è molto utile per sistemi dinamici/aperte (separazione temporale tra partecipanti, set dinamico di processi)
    - È utile anche quando P e C hanno velocità diverse
    - Ha bisogno di un **buffer** per archiviare e ritrovare dati, una struttura dati con stato mutabile, letta dai consumatori e scritta dai produttori.

Buffer infinito: una sola interazione che dev'essere sincronizzata - C non deve provare a eseguire una operazione da buffer vuoto.

Invariante: nAvailItems.V = #buffer

```
buffer_illimitato buffer = {}
sem nAvailItems = (0,{}) (resource semaphore)

producer                 consumer
loop forever             loop forever
p1: item el = produce()  q1: wait(nAvailItems)
p2: buffer.append(el)    q2: item el = buffer.take()
p3: signal(nAvailItems)  q3: consume(el)
```

Buffer finito: c'è un'altra interazione che dev'essere sincronizzata: il produttore non deve provare ad aggiungere operazioni in un buffer pieno.

Invariante: nAvailItems + nAvailPlaces = N

```
buffer_limitato buffer = {}
sem nAvailItems = (0,{}) (resource semaphore) |
sem nAvailPlaces = (N,{})                     |--> semafori separati

producer                 consumer
loop forever             loop forever
p1: item el = produce()  q1: wait(nAvailItems)
p2: wait(nAvailPlaces)   q2: item el = buffer.take()
p2: buffer.append(el)    q3: signal(nAvailPlaces)
p3: signal(nAvailItems)  q4: consume(el)
```

Combinare mutex e semafori sincronizzati: come generalizzazione dei casi precedenti, consideriamo l'uso condiviso di una struttura dati nonatomica (buffer), quindi con operazioni nonatomiche. Introduciamo una mutex per garantire anche la mutex.

```
buffer_limitato buffer = {}
sem nAvailItems = (0,{}) (resource semaphore) |
sem nAvailPlaces = (N,{})                     |--> semafori separati
sem_bin mutex = (1,{})

producer                 consumer
loop forever             loop forever
p1: item el = produce()  q1: wait(nAvailItems)
p2: wait(nAvailPlaces)   q2: wait(mutex)
p3: wait(mutex)          q3: item el = buffer.take()
p4: buffer.append(el)    q4: signal(mutex)
p5: signal(mutex)        q4: signal(nAvailPlaces)
p3: signal(nAvailItems)  p4: consume(el)
```

Filosofi a cena

Problema relativo a 5 computer che competono per accedere 5 nastri condivisi. 5 filosofi eseguono due operazioni: *pensano* e *mangiano*. I piatti sono presi in comune ad un tavolo con 5 piatti e 5 forchette. Al centro del tavolo c'è una ciotola di spaghetti che viene riempita all'infinito. Gli spaghetti sono ingrovigliati, perciò ogni filosofo ha bisogno di 2 forchette per mangiare. Ogni filosofo può prendere le forchetta a destra o sinistra, ma una sola forchetta per volta.

Proprietà

```
loop forever
p1: think
p2: <pre-protocol>
p3: eat
p4: <post-protocol>
```

- un filosofo può mangiare solo se ha 2 forchette
- **mutex**: due filosofi non possono tenere la stessa forchetta contemporaneamente
- **libertà da deadlock**
- **libertà da starvation**
- comportamento efficiente in assenza di collisioni

Primo tentativo

```
sem_array[5] fork = [1,1,1,1,1]
loop forever
p1: think
p2: wait(fork[i])
p3: wait(fork[(i+1)%N])
p3: eat
p4: signal(fork[i])
p5: signal(fork[(i+1)%N])
```

- Ogni forchetta è un semaforo: con wait prende la forchetta, con signal la rilascia
- Può essere provato che nessuna forchetta è mai tenuta da due filosofi
- Crea **deadlock** (v. piu avanti): se tutti i filosofi prendono la forchetta a sx, uno prova a prendere la forchetta a dx (non disponibile)

Soluzione 1: tickets

Limitare il numero di filosofi che mangiano simultaneamente, introducendo il ticket "pasto": N-1 ticket per N filosofi.

```
sem_array[5] fork = [1,1,1,1,1]
sem ticket = (4,{})
loop forever
p1: think
p2: wait(ticket)
p3: wait(fork[i])
p4: wait(fork[(i+1)%N])
p5: eat
p6: signal(fork[i])
p7: signal(fork[(i+1)%N])
p8: signal(ticket)
```

Soluzione 2: rompere la catena wait-for

- Non puoi avere deadlock se l'ultimo filosofo prende prima la forchetta a dx e poi quella a sx, rompendo così la catena wait-for.
- Dato l'ordine totale degli identificatori delle forchette, l'ultimo filosofo prende la forchetta in ordine opposito rispetto all'ordine dei filosofi: prima quella a N-1 e poi quella a 0
- La soluzione riguarda scegliere la forchetta sempre nello stesso ordine

```
sem_array[5] fork = [1,1,1,1,1]
int first = min(i, (i+1)%N)
int second = max(i, (i+1)%N)
loop forever
p1: think
p3: wait(fork[first])
p4: wait(fork[second])
p5: eat
p6: signal(fork[first])
p7: signal(fork[second])
```

Lettori/scrittori

Astrazione del problema di accesso a un db. Divisione dei processi in due classi:

- **lettori**, escludono gli scrittori ma non altri lettori
- **scrittori**, escludono scrittori e lettori

Non c'è quindi problema ad avere processi che leggono dati concorrentemente, ma la scrittura deve avvenire in mutua esclusione.

La soluzione deve soddisfare queste invarianti (nR = n° lettori, nW = n° scrittori)

```
nR >= 0
nW = 0 || nW = 1
(nR > 0 -> nW = 0) && (nW = 1 -> nR = 0)
```

Soluzione sovravincolata tramite singolo semaforo che funziona come blocco:

```
sem_bin rw = (1,{})
database db;

reader                  writer
loop forever            loop forever
p1: wait(rw)            q1: wait(rw)
p2: item el = db.read() q2: item el = create()
p3: signal(rw)          q3: db.write(el)
                        q4: signal(rw)
```

Soluzione: i lettori non usano lo stesso blocco degli scrittori --> `mutexR` : blocco dei lettori per aggiornare strutture dati in comune (nr)

```
sem_bin mutexR = (1,{})
int nr = 0
sem_bin rw = (1,{})
database db

reader                      writer
loop forever                loop forever
p1: wait(mutexR)            q1: wait(rw)
p2: if(nr==0)               q2: item el = db.create()
p3:    wait(rw)             q3: db.write(el)
p4: nr+=1                   q4: signal(rw)
p5: signal(mutexR)
p6: item el = db.read()
p7: wait(mutexR)
p8: nr-=1
p9: if(nr==0)
p10:   signal(rw)
p11: signal(mutexR)
```

Deadlocks

Situazione in cui due o più azioni concorrenti aspettano che l'altra finisca, e nessuno lo fa. Condizioni necessarie affinchè ci sia deadlock:

- **mutex**: una risorsa che non può essere usata da più di un processo per volta
- **hold and wait**: processi che trattengono risorse potrebbero richiedere nuove risorse
- **no preemption**: nessuna risorsa può essere rimossa dal processo che lo trattiene. Le risorse possono essere rilasciate solo tramite esplicita azione del processo.
- **circular wait**: due o più processi da una catena circolare dove ogni processo attende per una risorsa che il prossimo processo in catena mantiene.

Deadlock con blocchi: accade quando più thread aspettano all'infinito a causa di dipendenze cicliche bloccanti, es. quando il thread A blocca L e prova a bloccare M, ma allo stesso tempo il thread B blocca M e prova ad acquisire L.

Ricerca e ripristino: ad esempio i database sono progettati per trovare e ripristinare dai deadlock. Il set di transazioni deadlockate si identifica analizzando il grafo di dipendenze *is-waiting*, dove si cercano cicli e, se presenti, si sceglie una vittima e si cancella la transazione. I meccanismi non sono automatici con JVM, bisogna chiudere l'applicazione.

Regola generale per evitare deadlock: N processi che condividono e acquisiscono più blocchi. Regole: assegnare un ordine totale ai blocchi e acquisire i blocchi sempre nello stesso ordine. Funziona perchè rende possibile avere dipendenze wait-for circolari tra processi.

Monitor

- Astrazione dati per la programmazione concorrente che incapsula **sincronizzazione** e **mutua esclusione** in accesso (stato + operazioni + policy di concorrenza).
- Generalizzazione del concetto del kernel o supervisore nei sistemi operativi, dove le CS come l'allocazione di memoria sono centralizzate in un programma privilegiato
  - Le app richiedono servizi che sono forniti dal kernel
  - I kernel vengono eseguiti in una modalità hw che non interferisce con le app
  - Monitor come versioni decentralizzate del kernel monolitico
- Generalizzazione della nozione di oggetto in OOP (classe che incapsula dati + operazione + sync/mutex)

Definizione astratta

```
monitor MonitorName {
   declaration of permanent variables
   initialization statements
   procedures (or entries)
}
```

Proprietà e feature

Tipi di dato astratto

- Monitor come **istanze di tipi di dato astratto**
  - Solo i nomi delle procedure sono visibili fuori dal monitor (interfacce)
  - Istruzioni all'interno del monitor non possono accedere a variabili dichiarate fuori dal monitor
  - Variabili permanenti sono inizializzate prima che ogni procedura è chiamata

Mutua esclusione

- **Mutua esclusione** implicita/intrinseca
  - Procedure eseguite di default in mutex

- Procedura del monitor chiamata da un processo esterno, attiva se un processo sta eseguendo un'istruzione nella procedura.
- Almeno un'istanza di una procedura del monitor potrebbe essere attva per volta
- Processi che trovano il monitor occupato sono sospesi.

Esempio: contatore classico, thread-safe

```
monitor Counter {
  int count;
  procedure inc(){
    count := count + 1
  }

  procedure getValue():int {
    return count;
  }
}
```

Osservazioni: mutex implicita, non richiede che i programmatori usino altri meccanismi (wait/signal/etc). Se le operazioni dello stesso monitor sono chiamate da più di un processo, l'implementazione assicura che queste siano eseguite sotto mutex (atomicamente). Se le operazioni di diversi monitor sono chiamate, la loro esecuzione può essere sovrapposta. Non c'è una coda esplicita associata con la procedura monitor (starvation).

Sincronizzazione

- Supporto esplicito alla **sincronizzazione**
  - tramite **variabili di condizione**, usate nei monitor dai programmatori per ritardare un processo che non può continuare ad essere eseguito in sicurezza finchè lo stato del monitor soddisfa qualche condizione booleana. Usato anche per risvegliare un processo ritardato quando la condizione diventa true.

Variabili di condizione: tipi di dato primitivi che possono essere usati per sospendere (wait) e riesumate (signal) processi dentro ai monitor. Rappresentano le condizioni (eventi) dello stato del monitor che aspettano di essere soddisfatte e che diventano soddisfatte. Due operazioni atomiche di base, `waitC` e `signalC`. Ogni v.c. è associata a una coda FIFO di processi bloccati.

- **`waitC`** : sospende l'esecuzione del processo e rilascia il blocco del monitor
- **`signalC`** : sblocca un processo in attesa di una condizione

```
waitC(cond) { signalC(cond) { cond.queue.push(p) if(!cond.queue.empty) { p.state = blocked q = cond.queue.pop() monitor.lock = releas
```

Esempio:

```
monitor SynchCell {
   int value
   bool available = false
   cond isAvail

   void set(int v) {
     value = v
     available = true
     signalC(isAvail)
   }

   int get() {
     if(!available)
        waitC(isAvail)
     return value
   }
}
```

Osservazioni: c'è un collegamento esplicito tra variabili di condizione e il monitor che le raccchiude. **`wait`** **rilascia il blocco del monitor**, ciò è essenziale per evitare che un processo che esegua `waitC` blocchi l'accesso al monitor.

Altre primitive:

- `emptyC(cond)` controlla se la coda è vuota
- `signalAllC(cond)` come signal, ma tutti i processi aspettano che le condizioni siano riesumate
- `waitC(cond,rank)` attende in ordine crescente il valore di rank
- `minrank(cond)` ritorna il valore di rank del processo in fronte alla coda di attesa

Esempio SynchCell, revisited:

```
monitor ImprovedSynchCell {
  int value;
  boolean available;
  cond isAvail;
  procedure set(int v){
    value := v
    available := true
    signalAllC(isAvail)
}
  procedure get():int {
    if (!available)
      waitC(isAvail)
    return value
}
}
```

```
monitor ImprovedSynchCell {
  int value;
  boolean available;
  cond isAvail;
  procedure set(int v){
    value := v
    available := true
    signalC(isAvail)
}
  procedure get():int {
    if (!available){
      waitC(isAvail)
      signalC(isAvail)
    }
    return value
  }
}
```

Implementare un semaforo

```
monitor Semaphore {
  integer s := <InitValue>
  cond notZero
  procedure wait(){
    if s = 0
      waitC(notZero)
    s := s - 1
  }
  procedure signal(){
s := s + 1
    signalC(notZero)
  }
}
```

```
monitor Semaphore {
  integer s := <InitValue>
  cond notZero
  procedure wait(){
    if s = 0
      waitC(notZero)
    else
s := s - 1 }
  procedure signal(){
    if emptyC(notZero)
s := s + 1 else
      signalC(notZero)
  }
}
```

Semafori vs Variabili di Condizione

Il semaforo:

- `wait` potrebbe/non potrebbe bloccare
- `signal` ha sempre un effetto
- `signal` slocca un processo arbitrariamente bloccato

- Un processo sbloccato da un segnale può riesumare l'esecuzione immediatamente

Il monitor: - `waitC` blocca sempre - `signalC` non ha effetto se la coda è vuota - `signalC` sblocca il processo in testa alla coda - un processo sbloccato da `signalC` deve attendere il processo segnalante per lasciare il monitor

Disciplina della segnalazione

Semantica

- S = precedenza ai processi segnalanti
- W = precedenza ai processi in attesa
- E = precedenza ai processi bloccati su una procedura

- **signal and continue**: il segnalatore continua e il processo segnalato esegue più avanti nel tempo. Non preemptive. **E < W < S**
- **signal and wait**: il processo segnalato esegue e il segnalatore aspetta, eventualmente in competizione con altri processi in attesa di entrare nel monitor. Preemptive. **E = S < W**
- **signal and urgent wait**: come signal e wait, ma il segnalatore ha priorità sui processi in attesa di blocco. Soluzione classica per i monitor. **E < S < W**

Problemi comuni usando i monitor

Produttori/consumatori

Lettori/scrittori

Allocazione e gestione di risorse

Codice sulle slide!

PCD Module 2.5

Elementi di progettazione di programmi concorrenti

Passi nella progettazione

Analizzare il problema per identificare concorrenze sfruttabili: decomposizione task e dati e loro dipendenze

Task

- Sequenza di istruzioni che opera assieme come gruppo (parti di un algoritmo/programma)
- Specifico "pezzo" di lavoro che deve essere necessariamente portato a termine come responsabilità di un qualche agente (definizione più astratta/OO)

Scelta dell'architettura concorrente

- Mappatura task <-> agenti - incapsulano logica e controllo dell'esecuzione del task
- Risorse condivise che incapsulano il risultato del lavoro di un agente (entità passive/monitor)
- Coordinazione media-risorse per definire l'interazione tra agenti e gestire le dipendenze (entità passive/monitor)
- Protocolli di scambio messaggi

Analisi del problema

Decomposizione dei task

Problema decomposto in una collezione di task indipendenti o quasi, principio divide et impera. Task sufficientemente indipendenti cosicchè gestire le dipendenze necessiti solo di una piccola frazione di tutto il tempo di esecuzione del programma.

Es. sistemi producer/consumer

Decomposizione dei dati

Possono essere facilmente decomposti in unità che possono operare relativamente indipendentemente. Prime unità di dato identificato, quindi task relativi a una unità possono essere identificati --> Decomposizione dei task segue la decomposizione dei dati.

Scala bene con il numero di processori disponibili. Esempio: moltiplicazione di matrici.

Analisi delle dipendenze

Analisi dipendenze relative ai task, per **raggruppare e ordinare** i task relativi ai tipi di dipendenze e massimizzare le performance.

Tipi di dipendenze:

- **temporali**
- **data/risorse**

Livello di progettazione - alcune classi concettuali

Parallelismo di risultato

Parallelismo visto in termini di risultati del programma. Progettazione del sistema intorno alla struttura dei dati o alla risorsa resa come risultato finale.

- otteniamo il parallelismo computando contemporaneamente tutti gli elementi del risultato
- strettamente correlati al modello di decomposizione dei dati

Ogni agente è assegnato a produrre un pezzo del risultato: tutti lavorano in parallelo fino alla naturale limitazione imposta dal problema. Le strutture dati adatte sono progettate per avvolgere il risultato (dati) in costruzione, incapsulando problemi di mutex e sincronizzazione.

Esempio costruzione casa: singole parti costruite singolarmente in parallelo da diversi lavoratori.

Parallelismo specialistico

Parallelismo visto in termini di un insieme di specialisti collegati in una rete logica di qualche tipo

Il parallelismo risulta da tutti i nodi della rete logica (tutti gli specialisti) attivi contemporaneamente. Ogni agente è assegnato per eseguire un tipo specifico di attività. Tutti lavorano in parallelo fino alla naturale limitazione imposta dal problema

I mezzi di coordinamento (protocolli di comunicazione, strutture dati condivise) vengono utilizzate per supportare la comunicazione e il coordinamento degli specialisti (caselle di messaggi, lavagne, servizi per eventi, ecc.). L'approccio è opposto rispetto al parallelismo dei risultati.

Esempio casa: parallelismo visto come singole unità di lavoro (elettricisti, carpentieri, idraulici, ecc.)

Parallelismo "agenda"

Parallelismo visto come agenda di task del programma. Progettazione del sistema intorno ad un particolare ordine del compito, dopodichè si assegnano molti lavoratori ad ogni passo, i quali possono possibilmente svolgere diversi passaggi in parallelo (flusso di lavoro). Ogni agente è assegnato per aiutare con l'elemento corrente all'ordine del giorno: tutti lavorano in parallelo fino alla naturale limitazione imposta dal problema. Le strutture dati condivise sono progettate per gestire i dati consumati e prodotti dalle attività di agenda (esempi: buffer limitati)

Esempio casa: agenda dei lavori, con lavoratori assegnati ad ogni fase a comptii diversi (lavoratori generalisti).

Recap

- I confini tra le tre classi non sono rigidi, si possono anche mixare
- Nonostante ciò, le tre classi individuano tre modi diversi di lavorare (rispettivamente: forma del lavoro finito, suddivisione crew di lavoro, lista di task)
- L'approccio può essere applicato ricorsivamente seguendo decomposizione task+dati

Uso delle classi concettuali

Per scrivere un programma concorrente:

- scegliere la classe di concetto più naturale per il problema;
- scrivere un programma utilizzando il metodo più naturale per quella classe concettuale; e
- se il programma risultante non è accettabilmente efficiente, trasformarlo metodicamente in una versione più efficace passando da un metodo più naturale ad un più efficiente

Dalle classi concettuali alle architetture + pattern

Alcune architetture possono aiutare a colmare il divario tra classi concettuali e implementazione.

Master-Workers

L'architettura è composta da un agente principale e da un insieme probabilmente dinamico di agenti di lavoro, che interagiscono con un mezzo di coordinamento adeguato che funge da sacco di attività:

- **agente principale**: decompila l'attività globale in sottotask, assegna i sottotask inserendo la loro descrizione nella bag e raccoglie i risultati delle attività
- **agente di lavoro**: ottiene l'attività da fare dalla bag, eseguire i compiti e comunica i risultati (variante con strutture di monitoraggio adatte)
- **bag di task resources**: tipicamente implementato come una blackboard o un buffer limitato

Fork-Join

Applicando ricorsivamente l'architettura del master worker i lavoratori stessi sono padroni, producendo compiti e raccogliendo risultati

- approccio efficace per implementare strategie concorrenti di divide-et-conquer
- può essere realizzato sia utilizzando un sacco di attività o direttamente

Filter-Pipeline (pattern basati su flussi)

L'architettura è composta da una catena lineare di agenti che interagiscono con alcune risorse del buffer o delle risorse del canale o del buffer limitato

- **agente generatore**: l'agente che inizia la catena, generando i dati elaborati dalla pipeline
- **agente filtrante**: un agente intermedio della catena, che consuma informazioni d'ingresso da una pipe e produce informazioni in un altra pipe
- **agente sink**: l'agente che chiude la catena, raccogliendo i risultati

Esempio: image processing

Announcer-Listeners (coordinazione basata su eventi)

L'architettura è composta da un agente di annunciatore e da un insieme dinamico di agenti di ascolto, che interagiscono con un servizio di evento

- **agente annunciatore**: annuncia l'avvenimento di un evento sul servizio degli eventi
- **agenti ascoltatori** registrati sul servizio dell'evento in modo da essere informati del verificarsi di eventi interessanti per l'ascoltatore
- **event-service resource**: disattiva l'interazione degli annunciatori-ascoltatori e raccoglie e invia eventi

PCD Module 2.6

Formalismi visuali

Formalismi visuali

Servono per descrivere rigorosamente modelli di sistemi concorrenti per mezzo di un qualche tipo di diagramma visuale. Possono rappresentare, ad esempio, le dinamiche di un sistema o la struttura di task e dipendenze.

Sono utili sia per la specifica dei requisiti che per l'analisi e la progettazione. L'analisi è formale quando è formalmente specificata.

In questo capitolo:

- Reti di Petri
- Diagrammi a stati
- Diagrammi di attività

Reti di Petri

- Modello formale e astratto del flusso di informazioni; descrive ed analizza i flussi di informazioni e controllo nel sistema, in particolare nei sistemi che esibiscono attività concorrenti e asincrone.
- Utilizzo principale: modellazione di sistemi ad eventi in cui è possibile che alcuni eventi possano occorrere concorrentemente, ma sono presenti vincoli sulla concorrenza, precedenza o frequenza di queste occorrenze.

Grafo delle reti di Petri

- Grafo bipartito che rappresenta una rete di Petri,
  - due tipi di nodi: **posti** (cerchi), **transizioni** (barre)
  - connnessi da archi direzionati

- Modella le proprietà statiche del sistema

Token e marcatori

- Una rete di Petri ha proprietà dinamiche che risultano dalla sua esecuzione
  - l'esecuzione di una rete è controllata dalla posizione e dal movimento di marcatori chiamati

token, indicati da punti neri.
    - una rete con token è chiamata **rete di Petri contrassegnata**

Regole di esecuzione

- I token sono mossi dall'esecuzione di transizioni nella rete
- Una transizione deve essere *abilitata* affinchè venga eseguita
- Una transizione è abilitata quando tutti i suoi *posti di input* hanno un token all'interno
- Le transizioni vengono eseguite rimuovendo i token di abilitazione dai relativi posti di input e generando nuovi token che sono depositati nel posto di output della transizione. (vedi figure, si capisce di piu)

Marcatori

La distribuzione dei token in una rete di Petri marcata definisce lo stato della rete ed è chiamata marcatore. Il marcatore può cambiare come risultato dell'esecuzione di una transizione. Transizioni diverse possono essere eseguite con risultati di marcatori diversi (non-determinismo inerente).

(vedi figure)

Transizioni di demarcazione

- transizione **sorgente**: senza alcun posto di input, produce solo token ($|| \rightarrow \circle$)
- transizione **rubinetto**: senza alcun posto di output, consuma solamente token ($\circle \rightarrow ||$)

Archi pesati

- Una variante consiste nell'uso di archi pesati
    - una transizione è abilitata se ogni posto di input p di t è marcato con almeno $w(p,t)$ token, ovvero con il peso dell'arco da p a t.
    - l'esecuzione di una transizione abilitata t rimuove $w(p,t)$ token da ogni posto di input p di t e aggiunge w token ad ogni posto di output p di t, con $w(t,p)$ peso dell'arco da t a p.

Esempio: reazione H2O

Modellazione con reti di Petri

Le RdP possono essere usate per modellare naturalmente sistemi concorrenti in termini di **eventi**, **condizioni** e le **relazioni** tra di essi. Ovvero: in un sistema, in un qualsiasi momento, ci saranno certe condizioni. Il fatto che tali condizioni siano mantenute può causare l'occorrenza di certi eventi. Le occorrenze possono cambiare lo stato del sistema, causando alcune delle condizioni precedenti per cessare il mantenimento e causando altre condizioni ad essere mantenute ???. Esecuzione di una transizione = occorrenza di un evento, atomico.

Modellazione di concorrenza e parallelismo

Le RdP sono ideali per modellare sistemi a controllo distribuito con multipli processi che occorrono concorrentemente. **Vedi img su slide**

Modellare conflitti ed eventi concorrenti

- **eventi di conflitto**: due eventi sono in conflitto se uno dei due può accadere ma non entrambi.
- **eventi concorrenti**: due eventi sono concorrenti se entrambi gli eventi possono accadere in qualsiasi ordine senza conflitti.
- Una situazione dove conflitto e concorrenza sono mischiati è chiamata **confusione**.

Asincronia e località

In una RdP non ci sono inerenti misure di tempo o di flusso di tempo. L'unica importante proprietà del tempo, da un punto di vista logico, è definire un **ordinamento parziale** di occorrenze degli eventi. Eventi che hanno bisogno di non essere vincolati in termini di ordine di occorrenza non sono vincolati.

Località: in un sistema complesso composto da sottoparti operative asincronicamente e indipendente, ogni parte può essere modellata tramite RdP. L'abilitazione e l'esecuzione di transizioni sono affette da, e affliggono solo, cambiamenti locali nella marcatura di una RdP.

Non-determinismo

Una RdP è vista come una sequenza di eventi discreti il cui ordine di occorrenza è uno dei possibili consentiti dalla struttura base. Se in un qualsiasi momento più di una transizione è abilitata, allora qualsiasi delle diverse transizioni abilitate può essere eseguita. La scelta di quale transizione venga eseguita è fatta in maniera non deterministica (random).

Eventi atomici vs nonatomici

L'occorrenza di eventi primitivi è istantanea: eventi non primitivi (con una durata) devono essere modellati da multipli eventi, ad esempio attività con eventi di inizio e di fine.

Gerarchie

- Supporto naturale alla modellazione di gerarchie:
  - **astrazione** un'intera rete può essere rimpiazzata da un singolo posto o transizione per modellare ad un livello più astratto
  - **rifinimento**: posti e transizioni possono essere rimpiazzati da sottoreti per fornire una modellazione più dettagliata.

Applicazioni alla progettazione concorrente e programmazione

Le RdP possono essere utilizzate per modellare sistemi software, in particolare quelli concorrenti, per:

- Rappresentare problemi
  - problemi di CS e Mutex
  - sincronizzazione
- Rappresentare il comportamento di meccanismi
  - semafori
  - sincronizzatori
- Rappresentare interi problemi
  - produttore / consumatore
  - lettori / scrittori
  - filosofi a cena

Problemi di CS e mutex

Vedi slide

Sincronizzazione

Si impone un ordine tra le azioni dei processi, rappresentate tramite transizioni e relazionando le azioni (transizioni) tramite le condizioni (posti)

Vedi slide

Semafori

- Semafori modellati come risorsa condivisa (posti)
  - **wait(P)** modellato come transizione dove il risultato del semaforo è il posto in input
  - **signal(Q)** modellato come transizione dove il risultato del semaforo è il posto in output

Produttori e consumatori

Vedi slide

Lettori e scrittori

Gli n token in p_1 rappresentano n processi che possono voler leggere o scrivere in una memoria condivisa rappresentata da p_3.

Vedi slide

Filosofi a cena

Le forchette sono rappresentate dai posti a_i. I filosofi pensano nei posti A_t, B_t, C_t, D_t, E_t e mangiano nei posti A_e, B_e, C_e, D_e, E_e.

Reti di Petri estese con archi inibitori

Estensione zero-testing: si estendono le RdP base con la possibilità di eseguire una transizione solo se certi posti hanno zero token. Gli **archi inibitori** sono rappresentati da archi con un piccolo cerchio alla fine.

RdP + Archi inibitori = Turing-equivalente: problemi di espressività e indecidibili.

Descrizioni formali di Reti di Petri

Le RdP possono essere formalmente descritte per abilitare un'analisi rigorosa delle proprietà e dei problemi del sistema modellato:

- **proprietà strutturali**, indipendenti dalle marcature iniziali
- **proprietà comportamentali**, dipendente dalle marcature

Mappando la correttezza del sistema su proprietà strutturali/comportamentali - proprietà di sicurezza e vitalità.

Struttura delle Reti di Petri

- Può essere descritta dalla tupla $C = (P, T, I, O)$.
    - P = insieme di posti
    - T = insieme di transizioni
    - I = funzione di input, definisce l'insieme di posti di input per ogni transizione t_j
    - O = funzione di output, definisce l'insieme di posti di output per ogni transizione t_j

Esempio

```
P = { p1, p2, p3, p4, p5 }
T = { t1, t2, t3, t4 }

I(t1) = {p1}
I(t2) = {p2,p3,p5}
I(t3) = {p3}
I(t4) = {p4}

O(t1) = {p2,p3,p5}
O(t2) = {p5}
O(t3) = {p4}
O(t4) = {p2,p3}
```

Marcatori

Un marcatore è un'assegnazione di token ai posti della rete. Può essere rappresentata formalmente come:

- un vettore di N elementi, uno per posto, rappresentanti il numero di token per ogni posto
- una funzione $\mu : P \to N$, $\mu(p_i)$ = numero di token nel posto p_i

Una RdP marcata è rappresentata dalla 5-tupla $M = (P, T, I, O, \mu)$

Semantica di esecuzione delle regole

- Lo stato di una RdP è definito dai suoi marcatori. L'esecuzione di una transizione rappresenta un cambiamento in uno stato della rete
- **Funzione parziale prossimo-stato** $\delta(\mu, t_j)$: la funzione è indefinita se la transizione non è abilitata alla marcatura. Se t_j è abilitata, $\mu' = \delta(\mu, t_j)$ è la marcatura che risulta dalla rimozione dei token dall'input di t_j e aggiungendo token all'output di t_j.
- Data una RdP e un marcatore iniziale, possiamo eseguire la RdP tramite successive esecuzioni di transizioni
  - Eseguire una transizione t_j in un marcatore iniziale produce un nuovo marcatore $\mu^1 = \delta(\mu^0, t_j)$
  - In questo nuovo marcatore possiamo eseguire qualsiasi nuova transizione abilitata, chiamiamola t_k, risultante in un nuovo marcatore $\mu^2 = \delta(\mu^1, t_k)$
  - Ciò può continuare finchè c'è almeno una transizione abilitata in ogni marcatore
  - Se raggiungiamo un marcatore dove nessuna transizione è abilitata, allora nessuna transizione può essere eseguita e la RdP deve fermarsi.

- **Nondeterminismo**

  - Sequenze multiple di marcatori $(\mu^0, \mu^1, \mu^2 \dots)$ e le relative transizioni $(t_j^0, t_j^1, t_j^2 \dots)$ possono risultare dall'esecuzione di una RdP

Insieme raggiungibile

- **Marcatori immediatamente raggiungibili**: un marcatore m' è immediatamente raggiungibile da m se possiamo eseguire qualche transizione abilitata in m risultante in m'.
- **Marcatori raggiungibili**: un marcatore m' è raggiungibile da m se è immediatamente raggiungibile da m o è raggiungibile da qualsiasi marcatore m'' che è immediatamente raggiungibile da m'.
- **Insieme raggiungibile di una RdP**: insieme di tutti gli stati in cui la RdP può entrare in una qualsiasi esecuzione

Rappresentazone matriciale

- I componenti di una RdP possono essere convenientemente rappresentati come matrici:
  - matrice $m \times n$, m righe = transizioni, n colonne = posti
  - **input**: matrice $D-$. Elemento $D - [i, j]$ = peso dell'arco che connette p_j con la transizione t_i.
  - **output**: matrice $D+$. Elemento $D + [i, j]$ = peso dell'arco che connette t_i con la transizione p_j.
  - **incidente**: matrice $D = D + -D-$
  - **transizione**: matrice T di dimensione $(1 \times m)$ rappresentante l'esecuzione della RdP
  - **marcatore corrente**: matrice M di dimensione $(1 \times n)$ rappresentante il numero corrente di token a posto

- Per calcolare l'evoluzione di una RdP: $M' = T \times D + M$

Analisi dei modelli di RdP

- **reti sicure**: reti di Petri in cui non più di un token può mai essere in un qualsiasi posto nella rete allo stesso tempo. Giustificazioni basate sulla definizione originale di eventi e condizioni.
- **reti delimitate o k-reti delimitate**: reti in cui il numero di token in qualsiasi posto è limitato da k. Le reti sicure sono 1-rete delimitata. La delimitazione è una proprietà pratica molto importante.
- **reti conservative**: una RdP è conservativa se il numero di token nella rete è conservato.

Vitalità

- Basata su **analisi delle transizioni**

  - **transizioni morte** in una marcatura, se non c'è sequenza di transizioni che possano abilitarla (deadlock)
  - **potenzialmente eseguibile**, se non esiste qualche sequenza che la abilita (starvation)
  - **transizione viva** eseguibile in tutti i marcatori raggiungibili

- Per la vitalità, è importante non solo che una transizione sia eseguibile in un dato marcatore, ma rimangono potenzialmente eseguibili in tutti i marcatori raggiungibili per quel marcatore

Estensioni di RdP

- **Reti temporizzate**
    - reti temporizzate deterministiche
    - **Reti stocastiche**
- **Reti ad alto livello**
    - **Reti colorate**

Statecharts

- Per modellare **sistemi reattivi complessi**, ora parte di UML
- **Sistemi reattivi**: sistemi guidati dagli eventi, devono continuamente reagire a stimoli interni e esterni (automobili, reti di comunicazione, OS, ecc...). In contrasto con sistemi trasformazionali (I/O, processazione dati)
- *Obiettivo*: introdure un modo per descrivere comportamenti reattivi che sono chiari e realistici, e allo stesso momento formali e rigorosi, così da essere simulati e analizzati.

Oltre i diagrammi a stati di base

- **Stati ed eventi** sono un mezzo naturale per descrivere il comportamento dinamico di un sistema complesso.
- **Transizione di stato**: quando un evento accade in A, se la condizione C è vera in quel momento, il sistema passa allo stato B
- FSM e transizioni non scalano con la complessità: ingestibili, crescono esponenzialmente gli stati, i quali devono essere gestiti in maniera "piatta". Portano inoltre a diagrammi di stato caotici, non strutturati e irrealistici.
- Per essere utile, un approccio stato/evento deve essere *modulare, gerarchico e ben strutturato*. Deve risolvere il problema di ingrandimento esponenziale, rilassando i requisiti per cui tutte le combinazioni di stati devono essere rappresentate esplicitamente.
- **Statecharts**: estensione dei diagrammi di stato convenzionali tramite meccanisimi che migliorano la potenza descrittiva.

Formalismo

- Formalismo formale per descrivere stati e transizioni in modo modulare:
    - **gerarchia**: raggruppamento, raffinamento, promozione di capacità di "zoom" per muoversi facilmente tra i vari livelli di astrazione
    - **ortogonalità**: indipendenza/concorrenza dei sottostati, sincronizzazione tra sottostati

Eventi e stati negli statecharts

- rettangoli arrotondati = stati

- frecce etichettate = eventi (opzionalmente con condizioni tra parentesi e azioni)
- diversi livelli di stati (gerarchia): l'incapsulamento esprime la gerarchia, le frecce possono partire e terminare in qualunque livello.

Gerarchia

- Decomposizione XOR

Raggruppamento (clustering)

- Vedi immagine: dal momento che β porta il sistema verso B da entrambi gli stati A o C, si possono raggruppare questi ultimi all'interno di un super-stato D e rimpiazzare le due frecce con una.
- La semantica di D è uno XOR tra A e C: per essere nello stato D uno deve essere in A o C, e non in entrambe. D è un'astrazione di A e C.
- Approccio bottom-up

Raffinamenti

- Direzione opposta, rifinimento degli stati
- Approccio top-down
- Supporto a zoom-in (guardando dentro lo stato) e zoom-out (astraendo)

Stati di default

Freccie speciali che rappresentano esplicitamente lo stato di entrata di default, a qualsiasi livello.

Ortogonalità

- **Decomposizione AND**
- Cattura la proprietà per cui, in uno stato, il sistema possa essere in tutti i suoi componenti AND
- La notazione è quella di suddividere un box in due componenti tramite tratteggiatura
- (Vedi figura) Lo stato Y consiste nell'AND dei componenti A e D, con la proprietà che essere in Y significa essere in una qualche combinazione di B o C con E, F o G. Y è il prodotto ortogonale di A e D.
- Indipendenza e/o concorrenza

Indipendenza

- (Vedi immagine) Mi accade a (B,F) e affligge solo il componente D, risultante in (B,E).
- Ciò illustra un certo tipo di indipendenza: la transizione è la stessa sia che il sistema sia in B o C nella sua componente A.

Equivalente senza AND

- L'equivalente senza AND ha il prodotto degli stati (vedi immagine)
- Se abbiamo due componenti con 1000 stati, avremo un milione di stati nel prodotto. 3 componenti: 10^9.

Sincronizzazione

- (Vedi immagine) se accade un evento α, trasferisce B a C e F a G simultaneamente, risultando in uno stato combinato (C,G)
- **Sincronizzazione**: un singolo evento causa eventi simultanei

Notazioni per la decomposizione AND, esempi

Vedi slide

Azioni

Le **azioni** rappresentano la capacità dei diagrammi di stati di generare eventi e di cambiare il valore delle condizioni, influenzando altri componenti e l'ambiente del sistema.

Espresse dalla notazione ".../W" che può essere attaccata all'etichetta della transizione. W è una azione portata avanti dal sistema. Le azioni hanno occorrenze istantanee che impiegano tempo zero (idealmente).

Le azioni possono essere riprodotte anche mentre si entra o esce da uno stato.

Vedi immagini.

Rappresentare le attività

Le attività impiegano sempre un tempo diverso da zero (beep, displaying, ecc). Negli statechart, le attività sono associate con gli stati, esplodendo le azioni di entrata e uscita.

Statemate

Strumento di modellazione e simulazione grafica per lo sviluppo di sistemi embedded complessi basati su statechart. Fornisce un collegamento formale e diretto tra requisiti dell'utente e l'implementazione software consentendo all'utente dio creare specifiche complesse ed eseguibili.

Diagrammi di attività

Sono uno degli schemi adottati in UML per rappresentare il business e i flussi operativi di un sistema software. Un diagramma di attività è uno schema dinamico che mostra l'attività e l'evento che causa

l'oggetto deve essere in questo stato

- Diagrammi di attività vs di stato
  - un diagramma di stato mostra i diversi Stati di un oggetto durante il ciclo di vita della sua esistenza nel sistema e le transizioni degli Stati degli oggetti. Queste transizioni raffigurano le attività che causano queste transizioni, indicate dalle frecce
  - un diagramma di attività parla di più su queste transizioni e le attività che causano i cambiamenti negli Stati oggetto

Panoramica

Mostrando il flusso delle attività attraverso il sistema, i diagrammi sono letti dall'alto verso il basso e hanno rami e forche per descrivere condizioni e attività parallele. Una forchetta viene utilizzata quando più attività si verificano allo stesso tempo.

Il diagramma seguente illustra una forchetta dopo activity1. Questo indica che sia activity2 e attività3 si verificano allo stesso tempo. Dopo activity2 c'è un ramo. Il ramo viene descritto quali attività si svolgeranno basato su un insieme di condizioni. Tutti i rami a un certo punto sono seguiti da un'Unione per indicare la fine del comportamento condizionale iniziato da quel ramo.

Dopo l'Unione tutte le attività parallele devono essere combinate da un join prima di passare allo stato di attività finale.

Esempio: gestione ordini

Il diagramma mostra il flusso delle azioni del flusso di lavoro del sistema

- una volta ricevuto l'ordine le attività divise in due serie parallele di attività
- un lato riempie e invia l'ordine mentre altre la gestisce la fatturazione
- sul lato ordine di riempire, il metodo di consegna viene deciso in modo condizionale.
- a seconda della condizione viene eseguita sia la consegna durante la notte o l'attività di consegna regolari.
- Infine le attività parallele si combinano per chiudere l'ordine.

Corsie

Un corsia è un modo per attività di gruppo svolte dall'attore stesso su un diagramma di attività o per attività di gruppo in un singolo thread

PCD Module 3.1

Modelli a scambio di messaggi

Scambio di messaggi

Idea di base: l'unico modo per i processi di interagire è quello di scambiarsi messaggi - primitive **send** e **receive**. Nessun'altra astrazione è permessa (oggetti condivisi, monitor, lock, semafori...)

Modello naturale per la programmazione distribuita, ma considerata sempre di più anche per la programmazione concorrente in generale e nei linguaggi/piattaforme/tecnologie più popolari, al fine di evitare le insidie introdotte dalla programmazione multithreading, sincronizzazioni e mutex.

Implementato in HTML5 Web Workers, framework ad attori, linguaggi (Go, Dart, ecc...)

- Storia

Primitive di base

- Tipo canale
  - `chan ch(type id1, ..., type idn)`
    - `ch` : nome del canale
    - `type` , `id` : tipi e nomi dei campi dati nei messaggi trasmessi sul canale
- Primitive di comunicazione
  - `send ch(expr1, expr2, ... exprn)`
    - Per inviare un messaggio composto dalle espressioni `expr` tramite il canale `ch` .
    - `expr` sono espressioni il cui tipo deve coincidere ai campi corrispondenti nella dichiarazione del canale
  - `receive ch(var1, var2, ... varn)`
    - Per ricevere un messaggio da un canale `ch`
    - `var` sono le variabili il cui tipo deve coincidere ai campi corrispondenti nella dichiarazione del canale

Atomicità: l'accesso al contenuto di ogni canale è atomico. I canali sono dichiarati globali ai processi.

Sincronismo/asincronismo

- **Comunicazione sincrona**
  - L'invio di un messaggio su un canale è bloccato finchè il messaggio non viene ricevuto sullo stesso canale.
  - La ricezione è bloccata finchè il messaggio è inserito nel canale
  - Modelli primitivi, usati nell'algebra dei processi
- **Comunicazione asincrona**
  - I canali hanno un buffer FIFO dove i messaggi sono accodati
  - L'invio di un messaggio sul canale ha successo nel momento in cui il messaggio viene accodato nel canale.
  - La ricezione è bloccata finchè il messaggio diventa disponibile sul canale
  - In alcuni modelli/sistemi, i canali con scambio assincrono di messaggi vengono chiamati **porte**.

Pattern

- **1-a-1**: un canale può essere usato solo da una coppia di processi, tipico nella comunicazione sincrona
- **molti-a-molti**: lo stesso canale può essere usato da più mittenti e più ricevitori. Ciò porta a competizione nella ricezione dei messaggi e nondeterminismo
- **molti-a-uno**: usato con le porte, ogni porta ha un solo ricevitore ma mittenti multipli

Esempi

```
chan requestA(int value)
chan requestB(int value)
chan response(int value)
P                           Q                           R
r1,r2: integer          r: integer              r: integer
...                         ...                         ...
send requestA(5)            receive requestA(r)     receive requestB(r)
send requestB(6)            send response(r*2)      send response(r+1)
receive response(r1)    ...                         ...
receive response(r2)
write(r1+r2)
...
```

Produttore-consumatore

P-C che interagisce per mezzo di un singolo canale

```
channel buf(int)
PRODUCER                CONSUMER
integer x               integer y
loop forever:       loop forever:
p1: x ← produce         q1: receive buf(y)
p2: send buf(x)     q2: consume(y)
```

Pipeline

Un processo che assembla linee dei caratteri, funzionante come filtro in un'architettura a pipeline.

```
chan input(char), output(char[MAXLINE])
process   CharToLine {
    char line[MAXLINE+1];
    int i = 0;
    while (true) {
        receive input(line[i]);
        while (line[i]!=CR and i<MAXLINE){
            i = i + 1;
            receive input(line[i]);
        }
        line[i] = EOL;
        send output(line)
        i = 0;
    }
}
```

Interazioni client-server

```
chan  request(int,kind,arg_type);
chan[NCLIENTS] reply(arg_result);

process Client[i = 0...N-1] {
    arg_type myargs;
    res_type myres;

    <init args>
    send request(i,opXXX, myargs);
    receive reply[i](myres);
}
```

Gestori di risorse attive vs monitor

I processi server attendono richieste su un canale e forniscono un risultato su un canale dedicato. Funzionano come **monitor attivi**, tipicamente usato per implementare **allocatori di risorse**.

```
chan  request(int,kind,arg_type);
chan[NCLIENTS] reply(arg_result);

process Server {
    int clientID; op_kind kind;
    arg_type args; res_type res;

    <init code>
    while (true) {
        receive request(clientID,kind,args);
        if (kind == op1){
            body of op1
        } else ...
        } else if (kind == opN){
            body of opN
        }
        send reply[clientID](results);
    }
}
```

Gestori di risorse attive - vedi slide

Comunicazione guarded

Come ricevere messaggi che possono arrivare su canali multipli allo stesso tempo? Utilizzando i guard (**guardie**), Dijkstra 1974, sia nel caso sincrono che in quello asincrono. È un meccanismo utilizzato per realizzare **ricezioni selettive** utilizzando delle istruzioni **select** (sockets, Java nio...)

Istruzioni

- $B; C \rightarrow S$
    - B = espressione booleana (default: true)
    - C = istruzione di comunicazione (default: receive)
    - S = blocco di istruzioni o lista
- B e C compongono ciò che è chiamato **guardia**
    - Una guardia ha successo se B è vera e l'esecuzione di C non causa un ritardo
    - Una guardia fallisce se B è falsa
    - Una guardia blocca se C non può ancora essere eseguita

Selezione + guardie

```
if B1;C1 → S1;
[] B2;C2 → S2;
[] B3;C3 → S3;
fi
```

1. Valuta le espressioni booleana di guardia
   - se tutte falliscono, l'if termina senza effetti
   - se almeno una ha successo, scegline una non deterministicamente
   - se tutte le guardie si bloccano, aspetta finchè una guardia ha successo
2. Esegue l'istruzione C per la guardia scelta
3. Esegue il blocco S

Esempio:

```
if nReqs < max; receive computeSum(a,b,i)
    → nReqs+=1; send result[i](a+b)
[] nReqs < max; receive computeMul(a,b,i)
    → nReqs+=1; send result[i](a*b)
fi
```

Loop + guardie

Con istruzioni "do"

```
do B1;C1 → S1;
[] B2;C2 → S2;
[] B3;C2 → S3;
od
```

In questo caso la selezione del processo è ripetuta finchè tutte le guardie falliscono

```
process Copy(chan in(char), chan out(char)) {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; receive in(buffer[rear])
        → count++; rear = (rear+1)%10;
    [] count > 0; send out(buffer[front])
        → count--; front = (front+1)%10;
    od
}
```

Esempio producer/consumer su slide

Interazione peer-to-peer

Decentralizzazione della responsabilità tra le parti interessate (**peers**). Coordinazione per mezzo di protocolli di interazione basati su messaggi tra i peers.

Esempio: problema dello scambio di valori. N processi, ognuno con un intero locale v. L'obiettivo per ogni processo è imparare il più piccolo e più alto dei valori locali n.

Soluzioni possibili:

- **centralizzato**
- **simmetrico**
- **anello**

Soluzione centralizzata

- Un processo coordinatore riceve i valori da qualsiasi altro processo e stabilisce il minimo e il massimo.
- Numero basso di messaggi
- Collo di bottiglia con il processo coordinatore
- La **receive** del coordinatore è ritardata

```
chan values(int), results[n](int smallest, int largest);

process P[0] { #coordinator
    int v = ....;
    int new, smallest = v, largest = v; # initial state
    for i in [ 1... n-1] {
        receive values(new);
        if (new < smallest)
            smallest = new;
        if (new > largest)
        largest = new;
    }

    # send the result to the other processes
    for i in [1...n-1]{
        send results[i](smallest,largest);
    }
}

process P[i]{
    int v = ..., smallest, largest;
    send values(v);
    receive results[i](smallest, largest);
}
```

Soluzione simmetrica

- Ogni processo esegue lo stesso algoritmo
  - Invia il suo valore a tutti gli altri
  - Ogni processo in parallelo calcola il minimo e il massimo

- Numero alto di messaggi $N \times (N-1)$
- Massimizzazione della concorrenza nella distribuzione

```
chan values[n](int);

process P[i = 0 to n - 1] {
    int v = ....;
    int new, smallest = v, largest = v; # initial state
    # send my values to other processes
    for j in [0...n-1], j != i {
        send values[j](v);
    # gather values and save the smallest and largest
    for k in [1...n-1]{
        receive values[i](new);
        if (new < smallest)
            smallest = new;
        if (new > largest)
            largest = new;
    }
}
```

Soluzione ad anello

- Organizzazione dei processi in un anello logico.
- Ogni processo P_i riceve messaggi dal predecessore e lo invia al successore
- 2 stadi: determinare il max e min globale, diffonderlo sull'anello.
- Numero basso di messaggi
- Concorrenza limitata

```
chan values(int), results[n](int smallest, int largest);

process P[0] { #coordinator
    int v = ....;
    int new, smallest = v, largest = v; # initial state
    send values[1](smallest,largest);
    receive values[0](smallest,largest);
    send values[1](smallest,largest);
}

process P[i = 1 to n - 1]{
    int v = ..., smallest, largest;
    receive values[i](smallest, largest);
    if (v < smallest)
        smallest = v;
    if (v > largest)
        largest = v;
    send values[(i+1)%n](smallest,largest);
    receive values[i](smallest, largest);
    send values[(i+1)%n](smallest,largest);
}
```

Esempi

Filosofi a cena

- Problema in un ambiente distribuito:
- Ogni filosofo è in esecuzione su un nodo diverso della rete
- Anche le forchette (risorse) possono essere distribuite
- Problema generale: risolvere conflitti tra processi in un sistema distribuito

- **Soluzione centralizzata**: processo *Cameriere* che agisce come allocatore di risorse (forchette), deve garantire equità ed evitare deadlock.

- **Soluzione distribuita**: un *Cameriere* per ogni forchetta presente, un allocatore di risorse per ogni forchetta, adottando un'adeguato protocollo di coordinamento tra filosofi e camerieri per evitare deadlock.

Scambio di messaggi sincrono

Semantica sincrona: *send* bloccata finchè il messaggio non è stato ricevuto sul canale. Non c'è bisogno di buffer.

Comunicazione tramite (inter)azioni atomiche: i dati sono trasferiti su un canale che ha lugoo solo quando il

control pointer del mittente è nell'istruzione *send* sul canale e il control point del destinatario è sull'istruzione *receive* del canale.

Esempio producer-consumer

Con la semantica sincrona, il trasferimento dei dati avviene solo quando il c.p. di P = p2 o Q = q1.

```
chan ch(int)
PRODUCER                CONSUMER
x: integer          y: integer
loop forever:       loop forever:
p1: x ← produce         q1: receive ch(y)
p2: send ch(x)      q2: consume(y)
```

Il linguaggio Go

- Linguaggio staticamente tipizzato, sviluppato da Google nel 2007 (golang).
- Sintassi vagamente derivata da C, con garbage collector, type safety, qualcosa di dynamic typing, tipi built-in come array di lunghezza variabile e k-v maps.
- Molto diffuso
- Primitive concorrenti incluse: processi lightweight (**goroutines**), **channels** e **select**.
- Concorrenza originata nel Hoare's Communicating Sequential Processes: non comunicare tramite la memoria condivisa, ma usa la memoria condivisa per comunicare.

Rendez-vous

Rendez-vous: concetto di due persone che scelgono un posto per incontrarsi. Il primo aspetta sempre il secondo.

Sincronizzazione estesa, con il mittente che aspetta non solo che il destinatario riceva la risposta, ma che anche risponda al risultato.

Due ruoli:

- **processi chiamanti**: processi che chiamano un *entry* in un processo accettante. Necessita di sapere l'identità del processo accettante e il nome della entry. Primitiva **call**.
- **processi accettanti**: processano l'accettazione della richiesta su specifiche entry e inviano il risultato con una risposta.

```
CLIENT                                          SERVER
integer parm, result                          integer p, r
loop forever                                      loop forever
p1: param ← ...                               q1: accept service(p,r)
p2: call server.service(parms,result)   q2: <do the service with p> qX: ...
p3: use(result)                               qN: r ← <result>
```

RV in ADA

Supporto diretto tramite chiamata entry, meccanismo di comunicazione di base tra task.

Istruzione accept.

- entries declared in the task public interface – entries implemented in the task body by means of accept statement

Codice su slide

RPC

- Utilizzando il passaggio dei messaggi per realizzare l'astrazione di chiamata di procedura nel contesto distribuito
  - consentire un client richiedere un servizio da un server che può essere localizzato a un processore diverso.
    - il client chiama un server come una normale procedura
    - viene creato un processo per gestire la chiamata
    - È diverso da un rendez-vous RPC perché quest'ultimo prevede la partecipazione attiva di due processi in comunicazione sincrona

Implementazioni disponibili per scambio di messaggi basato su canale

- MPI
- PVM

Middleware orientato ai messaggi

- Abilitazione della comunicazione basata su messaggi tra applicazioni distribuite eterogenee
  - asincrono, basato su code/argomenti, punto-punto + pubblicare / iscrizione
  - in gran parte utilizzato nelle architetture orientate ai servizi

- Gran numero di implementazioni esiste. Standard proposti
  - AMQP
  - XMPP

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.

PCD Module 3.3

Programmazione asincrona

Programmazione asincrona

Stile di programmazione molto importante ai giorni d'oggi. Riguarda l'esecuzione e la gestione di computazioni/richieste/processi asincroni, astraendoli dai thread. Recentemente supportato da tutti i principali framework e piattaforme. Include una varietà di meccanismi ed architetture, inclusa la programmazione guidata dagli eventi. È una problematica ancora in corso d'opera.

Task e futures

Future: fornendo una computazione asincrona (task), un oggetto rappresentante il risultato della future o lo stato della computazione è creato e ritornato immediatamente. L'oggetto permette di:

- controllare lo stato del task (poll)
- bloccarlo quando il risultato del task viene richiesto
- cancellare il task in esecuzione quando possibile
- catturando errori/eccezioni relativi al task in esecuzione

Esempi: Java Task Executor, MS Task-based Async Pattern, Apple's GCD, Android's AsyncTask.

Esempio: JTE

Task richiamabili negli executor che rappresentano una computazione in differita, completati con qualche tipo di risultato. Il metodo `call` incapsula il comportamento del task che fa qualche tipo di computazione e ritorna qualche tipo di valore V. (`V call() throws Exception`)

Inviato ad un executor service per mezzo del metodo `submit` , ritornante una Future. Le Future forniscono medoti per vedere quando il task è stato completato/cancellato/risolto/eliminato.
(`Future<T> submit(Callable<T> task)`)

Esempio: MS TAP

Basato sui tipi `Task` e `Task<TResult>` del namespace `System.Threading.Tasks` , utilizzati per rappresentare operazioni arbitrarie asincrone.

TAP utilizza un singolo metodo per rappresentare l'inizializzazione e il completamento di una operazione asincrona. Il metodo ritorna un `Task` o un `Task<TResult>` in base alla risposta (`void` o

`TResult`).

Implementazione tramite compilatori C# o VB, manualmente o tramite aiuto del compilatore.

Implementazione manuale

```
public static Task<int> ReadTask(this Stream stream, byte[] buffer,
                                 int offset, int count, object state)
{
    var tcs = new TaskCompletionSource<int>();
    stream.BeginRead(buffer, offset, count, ar =>
    {
        try { tcs.SetResult(stream.EndRead(ar)); }
        catch (Exception exc) { tcs.SetException(exc); }
    }, state);
    return tcs.Task;
}
```

Con `async` e `await`

Metodi che possono essere sospesi tramite `async` e `await` rilasciando il controllo al chiamante. Un metodo `await` può essere a sua volta aspettato dai metodi che lo chiamano.

```
private async Task SumPageSizesAsync() {
    byte[] urlContents = await client.GetByteArrayAsync(url);
    // ...
}
```

Prima di TAP

- **APM**, Async Programming Model, con metodi `begin` e `end`, deprecato
- **EAP**, Event-based Async Pattern, metodi con suffisso `async`, eventi, tipi delegati e tipi derivati da `EventArg`, deprecato
- **TAP**, Task-based Async Pattern, un solo metodo per rappresentare inizializzazione e completamento di op. asincrona.

Continuation Passing Style e event-driven

Ripensare a computazioni e modelli di programmazione per renderli completamente basati sulla programmazione asincrona.

Eseguendo una computazione asincrona, specifichiamo le funzioni che devono essere chiamate come **continuazione** della computazione quando la computazione asincrona è completata o ha avuto errori. La continuazione ha un parametro, ovvero i lvalore di ritorno passato dalla computazione asincrona.

```
//   pseudo-definition
function myAsyncFunc(param1, ..., paramN, continuation){
  // when the result of myAsyncFunc is ready, then
  continuation(result) is called
}
// non blocking invocation
myAsync(...,cont_function(value) {...})
```

CPS in generale

È uno stile di programmazione in cui il controllo è passato esplicitamente nella forma di **continuazione**.

Una funzione scritta in CPS prende un argomento in più, ovvero una funzione di un argomento. Quando la funzione CPS ha computato il valore del risultato, lo ritorna chiamando la funzione di continuazione con il valore come argomento. Invocando una CPS, la funzione chiamante fornisce una procedura che deve essere invocata con il valore di ritorno della subroutine. (tipo callback objc)

Callback come continuazioni

Callback: continuazioni chiamate quando il risultato di una computazione asincrona è pronta. Esempio in JS:

```
function loadUserPic(userId, ret) {
  findUserById(userId, function(user) {
    loadPic(user.picId, ret);
  });
}

loadUserPic('john', function(pic) {
  ui.show(pic);
});
```

Modello event-loop

Modello esecutivo

Chi invoca le continuazioni/callback? Due possibilità:

- un thread di controllo separato, in esecuzione concorrentemente al thread di controllo che ha eseguito la richiesta, concorrentemente. Problemi: inversione del controllo, race condition
- lo stesso thread logico di controllo che ha eseguito la richiesta: **modello event loop**, adottato da web app moderne e nella programmazione event-driven

Architettura event-loop e programmazione event-driven

Comportamento organizzato come insieme di **event handlers** che incapsulano la computazione per essere eseguita quando un evento è ricevuto.

Viene usata una coda degli eventi per tenere traccia di eventi generati dall'ambiente o dall'event handler.

Comportamento concettualmente rappresentato da un loop:

```
loop {
    Event ev = evQueue.remove()
    Handler handler = selectHandler(ev)
    execute(handler)
}
```

Esecuzione della computazione asincrona atomica: gli eventi accadono finchè un handler è in esecuzione e in coda nella coda eventi.

Osservazioni

- Gli event handler sono pensati per essere eseguiti senza blocchi (primitive/chiamate/richieste). Un comportamento bloccante deve essere sostituito da una richiesta asincrona o computazione.
- L'event loop è embeddato dentro al runtime che esegue il codice: gli sviluppatori non lo vedono, non hanno bisogno di creare i loop - basta specificare come scegliere ed eseguire gli event handlers. Forti similitudini con l'event loop degli attori.
- La programmazione guidata dagli eventi è chiamata anche **programmazione senza stack delle chiamate**: la gestione degli eventi non è una chiamata alla procedura.

Callback come continuazioni: esempi

- Gli eventi sono impliciti e si riferiscono al completamento con successo o errore di richieste/computazioni asincrone
- Gli event handler sono rappresentati dalle funzioni di continuazione specificate quando viene invocata una richiesta/computazione asincrona
- Le closure sono usate per definire il contesto da usare quando si processano gli eventi.

Esempi Javascript su slide

L'inferno delle callback

Programmazione guidata dagli eventi: benefici

- Uso di un singolo thread per coordinare e gestire molteplici task asincroni (eventualmente eseguito da thread esterni)
- Non ci sono race a basso livello (no stati condivisi)
- Non ci sono deadlock a basso livello (handler async non bloccano mai)

... e problemi

- **spaghetti asincroni**: computazione generale spezzata in handler asincroni. Va a minare la modularità e la struttura del programma e rende difficile comprendere il codice.
- **piramide della dannazione**: composizione basata su CPS può portare a callback/continuazioni innestate. Ciò aumenta la complessità e porta a una leggibilità del codice povera, poca riusabilità, poca estendibilità.

Piramide della maledizione

Callback innestate. Esempio di chat con tre chiamate asincrone:

```
registerToChatOnServer(username,function(rooms){
    joinAvailableRoom(rooms, function(roomname){
        sendChatToAll(roomname, msg, function(reply){
            showChatReply(reply);
        })
    })
});
```

Promises

Una soluzione parziale: le promises

Rappresentano l'eventuale completamento e risultato di una singola operazione asincrona. Incapsulano azioni asincrone, agendo più come un valore ritornato dal risultato di una computazione con la differenza che il valore potrebbe non essere sempre disponibile. Una promise può essere rigettata o risolta una sola volta. Permette di appiattire l'innestamento delle callback.

In Javascript

Definite nelle standard, chiamate anche **thenables**. Supportate da diversi framework (Q, When.js, jQuery Promises, ecc)

```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then...
  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
}
else {
    reject(Error("It broke"));
  }
});

promise.then(function(result) {
  console.log(result); // "Stuff worked!"
}, function(err) {
  console.log(err); // Error: "It broke"
});
```

Incatenamento di promises:

```
var promise = new Promise(function(resolve, reject) {
  resolve(1);
});
promise.then(function(val) {
  console.log(val); // 1
  return val + 2;
}).then(function(val) {
  console.log(val); // 3
});
```

Esempi con Q:

```
Q.fcall(registerToChatOnServer)
 .then(joinAvailableRoom)
 .then(sendChat)
 .then(function (reply){
        showChatReply(reply)
    },function (error){
}) .done();

// configuring the promise..
get('story.json').then(function(response) {
  console.log("Success!", response);
}, function(error) {
console.error("Failed!", error); });
```

Comporre le promise

In molte librerie le promise sono astrazioni ad alto livello. Possono quindi essere passate tra funzioni.

Un programmatore può esegure un numero di operazioni ad alto livello sulle promise come, ad esempio, comporre un gruppo di chiamate asincrone indipendenti. Esempio di Fibonacci in Q:

```
fibpromise = Q.all([ computeFibonacci(n-1), computeFibonacci(n-2) ]);

fibpromise.spread(function (result1, result2) {
    //resolver for both operations..
    console.log(result1 + result2);
}), function(err){
    //error occurred
});
```

Per reagire a `fibpromise` attacchiamo un resolver, tramite metodo `spread`, che permette agli argomenti di essere inviati al resolver.

Altri framework

- **AngularJS**
- **Dart**
- AmbientTalk
- Argus
- Fignale

Le promise non sono la pallottola d'argento

Sono una soluzione elegante a risolvere il problema dell'innestamento, ma non tutto può essere *promesso*:

- incremental processing
- event processing & streams • UIlogic
- temporal logic

Esempi concreti:

- Live search
- Processing streams of pushed data (SSE, WebSockets...)
- Incremental processing if large data sets as streams

Meccanismi async/await

Dart

Integra async/await in un linguaggio basato sull'event loop. Integra Java + JS, può anche essere tradotto in puro Javascript. Competitor: Typescript di Microsoft.

Dart è basato su event loop e meccanismi relativi, ma una estensione recente propone meccanismi async/await e stream reattivi per semplificare la programmazione asincrona.

Event Loop

Un'app Dart ha un singolo event loop con due code:

- **event queue**, contiene tutti gli eventi esterni (IO, eventi mouse, grafici, timer, ecc)
- **microtask queue**, contiene gli eventi relativi ai task da eseguire, ordinati dal programma stesso. È necessario poichè il codice per gestire gli eventi qualche volta ha bisogno di completare un task successivamente, ma prima di ripassare il controllo all'event loop.

Diagramma ed esempi su slide.

Introduzione di Async/Await

- **async**: funzione il cui corpo è marcato con `async`. Quando chiamiamo un'async, ritorna una Future e il corpo della funzione è programmato per essere eseguito più tardi. Quando viene eseguito, la Future viene completata. Permette di usare le await nella funzione.
- **await**: `await funzione`, sospende la funzione attualmente in esecuzione finchè il risultato non è pronto (la Future non è completa). Il risultato della funzione await è il completamento della Future.

```
import "dart:html";
main() async {
    var context = querySelector("canvas").context2D;
    var running = true;    // Set false to stop game.
    while (running) {
        var time = await window.animationFrame;
        context.clearRect(0, 0, 500, 500);
        context.fillRect(time % 450, 20, 50, 50);
    }
}
```

PCD Module 3.4

Programmazione reattiva

Programmazione reattiva

Sia le CPS che le Promises richiedono che le computazioni asincrone facciano recapitare il risultato in un colpo. È comune però, ad esempio, nelle applicazioni la necessità di gestire **flussi** asincroni di dati o eventi. Questo è l'obiettivo della **programmazione reattiva**.

Tale paradigma è orientato ai flussi di dati e la propagazione dei cambiamenti, andando a facilitare la gestione di flussi asincroni di dati ed eventi. Relazionato con il pattern Observer e la programmazione guidata dagli eventi. Sta avendo molta attenzione nello sviluppo di applicazioni web responsive e big data.

Linguaggi e framework

- FrTime
- MS Reactive Extension
- Reactive Js - Flapjax, Bacon.js, AngularJs
- Reactive Dart
- Scala.React
- RxJava
- RxSwift...

Astrazioni

La RP astrae eventi variabili nel tempo per un loro consumo da parte del programmatore. Il programmatore può quindi definire elementi che reagiranno ad ogni evento in arrivo variabile nel tempo.

Le astrazioni sono valori di prima classe e possono essere passati o composti dal programma.

Due tipi di astrazioni reattive:

- **flussi di eventi**: modellano valori variabili nel tempo continui e discreti. Astrazione asincrona dal flusso progressivo di dati sequenziali e intermittenti in arrivo da un evento ricorrente (es. eventi del mouse)
- **comportamenti (segnali)**: valori continui nel tempo; astrazioni che rappresentano un flusso fluido e ininterrotto di dati in arrivo da un evento regikare (es. timer)

Esempio in Flapjax

```
var timer = timerB(100); // timer ogni 100ms - *comportamento*
var seconds = liftB( // altra var. comportamentale, memorizza tempo in sec.
    function (time){
        return Math.floor(time / 1000);
    }, timer );

insertDomB(seconds, 'timer-div'); // seconds è inserita nel DOM
// il valore del timer è continuamente aggiornato e mostrato nella pagina
```

Questo esempio crea un elemento DOM in una pagina web che mostra un timer in secondi.

Punto chiave: via completamente dichiarativa di gestire eventi e flussi.

Lifting

- I valori delle espressioni che dipendono da *valori reattivi* devono essere reattivi a loro volta.
- Una variabile che viene assegnata con qualche espressione che coinvolge _comportamenti o flussi di eventi diventa una variabile reattiva a sua volta, dal momento che l'aggiornamento da diversi comportamenti/flussi di eventi interessa la variabile stessa
- Il processo di convertire una normale variabile in una reattiva è chiamato **lifting**.
- Per ricomputare tutte le espressioni reattive una volta che un flusso di eventi o comportamento è eseguito, molte librerie costruiscono un **albero delle dipendenze**: quando un'espressione cambia, le espressioni dipendenti sono ricalcolate e i loro valori aggiornati.

Lifting implicito e esplicito

- Alcune librerie eseguono implicitamente il lifting (es. Bacon.js)
- Altre hanno bisogno che il programmatore esegua il lifting manualmente (es. React.js)
- Altre ancora fanno entrambe le cose (Flapjax)

Comporre flussi di eventi e comportamenti

Composizione reattiva astratta: permette di evitare l'incubo delle callback. Ad esempio, invece di avere tre callback separate per 3 eventi del mouse, si possono comporre come un singolo flusso id eventi che risponde a tutti e 3 gli eventi. Molte librerie forniscono questo tipo di supporto (Bacon.js con operatore combinazione, Flapjax con mergeE).

```
var saveTimer = timerE(10000); //10 seconds
var saveClicked = extractEventE('save-button','click');
var save = mergeE(saveTimer,saveClicked);
save.mapE(doSave); //save received data
```

Estensioni reattive

Per integrare aspetti di programmazione reattiva nei linguaggi principali si usano le **estensioni reattive (Rx)**. Rx è una libreria per comporre programmi asincroni e basati sugli eventi utilizzando collezioni osservabili.

Proprietà principali:

- **asincrona e basata sugli eventi**: la missione di Rx è semplificare i modelli di programmazione, un aspetto chiave per sviluppare programmi reattivi, dove l'asincronismo è essenziale.
- **composizione**: semplifica la composizione di computazioni asincrone
- **collezioni osservabili**: sfruttando la conoscenza attiva di modelli di programmazione come Linq, vedendo le computazioni asincrone come fonti di dati osservabili. Un mouse, ad esempio, diventa un db di eventi generati dal mouse, composti da diversi combinatori Linq.

Originalmente introdotte in .NET.

Principi

- Con Rx è possibile
  - rappresentare più *flussi di dati asincroni* in arrivo da diverse fonti
  - sottoscrivere al flusso di eventi utilizzando `IObserver<T>` (e `IOBservable<T>`
- Poichè le sequenze osservabili sono flussi di dati, possiamo eseguire query tramite operatori standard di Linq implementati in `Observable`. Inoltre è possibile filtrare, aggregare, comporre eventi multipli facilmente.
- Esistono un numero di altri operatori reattivi specifici per i flussi che permettono di scrivere query potenti. Cancellazioni, eccezioni e sincronizzazioni sono gestiti tramite i metodi di estensione di Rx.

`IObservable`

Rx va a completare e interoperare facilmente con flussi di dati sincroni `IEnumerable<T>` e computazioni asincrone a valore singolo `Task<T>`.

Interfaccia observer e observable

```
interface IObservable<T> {
  IDisposable Subscribe(IObserver<T> observer);
}
interface IObserver<T> {
  void OnNext(T value);
  void OnError(Exception error);
  void OnCompleted();
}
interface IDisposable {
  void Dispose();
}
```

Esempio

```
IObservable<int> source = Observable.Range(1, 10);
IDisposable subscription = source.Subscribe(
    x => Console.WriteLine("OnNext: {0}", x),
    ex => Console.WriteLine("OnError: {0}", ex.Message),
    () => Console.WriteLine("OnCompleted"));
Console.WriteLine("Press ENTER to unsubscribe...");
Console.ReadLine();
subscription.Dispose();
```

Quando un observer sottoscrive una sequenza osservabile, il thread che chiama il metodo `Subscribe` può essere diverso da quello in cui la sequenza viene eseguita fino al completamento. Di conseguenza, la chiamata Subscribe è asincrona in quanto il chiamante non viene bloccato fino al completamento l'osservazione della sequenza.

Ponti

Con eventi .NET esistenti

```
var lbl = new Label();
var frm = new Form { Controls = { lbl } };
IObservable<EventPattern<MouseEventArgs>>
      move = Observable.FromEventPattern<MouseEventArgs>(frm, "MouseMove");
move.Subscribe(evt => {
  lbl.Text = evt.EventArgs.Location.ToString();
});
```

Il flusso di eventi `mouse-move` è convertito in una sequenza osservabile. Ogni volta che viene eseguito `mouse-move`, il sottoscrittore riceve una notifica `OnNext`. Possiamo quindi esaminare il valore `EventArgs` di tale notifica e avere la posizione del `mouse-move`.

Con fonti asincrone esistenti

```
Stream inputStream = Console.OpenStandardInput();
var read =
  Observable.FromAsyncPattern<byte[], int, int, int>
                           (inputStream.BeginRead, inputStream.EndRead);
byte[] someBytes = new byte[10];
IObservable<int> source = read(someBytes, 0, 10);
IDisposable subscription = source.Subscribe(
        x => Console.WriteLine("OnNext: {0}", x),
        ex => Console.WriteLine("OnError: {0}", ex.Message),
        () => Console.WriteLine("OnCompleted"));

Console.ReadKey();
```

`BeginRead` e `EndRead` per un oggetto `Stream` che usa il pattern `IAsyncResult` sono convertiti in una funzione che ritorna una sequenza osservabile.

Queryzzare sequenze osservabili tramite operatori Linq

```
var s1 = Observable.Range(1, 3);
var s2 = Observable.Range(1, 3);
```

Per:

- **trasformare dati**: tramite aggregazione e proiezione
  - *aggregazione*: `s1.Concat(s2).Subscribe(Console.WriteLine)`. Risultato: 1,2,3,1,2,3
  - *unione*: `s1.Merge(s2).Subscribe(Console.WriteLine)`. Risultato: 1,1,2,2,3,3
  - *proiezione*: `var seqNum = Observable.Range(1, 5); var seqString = from n in seqNum select new string('*', (int)n); seqString.Subscribe( str =>` Mappa un flusso di numeri in un flusso di stringhe.
  - **comporre dati**: tramite zip e selectMany
  - **chiedere dati**: tramite where e any

RxJava

Implementazione Java VM di Rx. Estende il pattern observer per supportare sequenze di dati/eventi e aggiunge operatori che permettono di comporre sequenze assieme dichiarativamente, astraendo problematiche riguardanti threading a basso livello, sincronizzazione, thread-safety, strutture dati concorrenti, IO bloccante. Supporta Java 5+ e linguaggi basati su JVM (es. Kotlin e Scala).

Operatori RX

- **creazione di observable**
- **trasformazione di observable**: trasformano elementi emessi da un observable
  - `Map`
  - `Scan`
  - `FlatMap`
  - `Buffer`
  - `GroupBy`
  - `Window`
- **filtro di observable**: emettono selettivamente oggetti da una fonte osservabile
  - `Filter`
  - `Skip`
  - `Take`
  - `Debounce`
  - `Distinct`

    - `ElementAt`
    - `First`
    - `IgnoreElements`
    - `Last`
    - `Sample`
    - `SkipLast`
    - `TakeLast`

- **combinazione di observable**: lavorano con più fonti osservabili per creare un singolo observable

    - `Merge`
    - `Zip`
    - `And/Then/When`
    - `Join`
    - `CombineLatest`
    - `StartWith`
    - `Switch`

- **operatori condizionali e booleani**: valutano uno o più osservabili o elementi emessi dagli osservabili
- **operatori matematici e di aggregazione**: operano sull'intera sequenza di oggetti emessi da un observable
- **operatori di contropressione**: strategie per "coping" con osservabili che produce elementi più rapidamente di quanto li consumano gli osservatori

Diagrammi di Marble

Vedi slide

Sfide

Nè la programmazione reattiva, nè le estensioni possono essere considerate la pallotola d'argento della programmazione asincrona. Sono effettive per gestire i flussi di dati/eventi in uno stile funzionale, ma non come modello di programmazione asincrona general-purpose.

La sfida oggi è come integrare tutti questi approcci e tecniche:

- sync + async + push + pull
- exploiting concurrency
- devising models that work also in the case of ds

PCD Module 4.1

Sistemi distribuiti: introduzione

Sistemi distribuiti

- Computer che contengono più processori connessi tramite una rete di comunicazione.
- **Perchè usarli?** Scalabilità, modularità, eterogeneità, condivisione dei dati e delle risorse, struttura geografica, affidabilità, basso costo.
- **Perchè non usare solo i distribuiti?** (es. paralleli). Efficienza: aggiornare la memoria è comunque più veloce che scambiarsi messaggi.

Caratteristiche chiave e sfide

- **Assenza di un clock condiviso**: impossibile sincronizzare clock di processori diversi a causa dell'incertezza del ritardo di comunicazione
- **Assenza di una memoria condivisa**: impossibile per un processore conoscere lo stato globale del sistema
- **Assenza di un'accurata gestione dei fallimenti**: in sistemi asincroni distribuiti è impossibile distinguere tra un processore lento o fallito.

Modelli

Modello di programmazione canonico

- Applicazioni come processi "pesanti" che comunicano tramite scambio messaggi su dei canali
- Ogni processo può avere più thread, comunicando tramite una memoria condivisa
- Es.: procedurale (RPC), OO (distribuito, remoto), implementato tramite middleware (CORBA, Java RMI, ecc.)

Modellazione di calcoli distribuiti

- insieme di processi poco accoppiati che mandano messaggi tramite canali unidirezionali: buffer infinito, senza errori, nessun ordine dei messaggi, qualsiasi messaggio inviato ha un ritardo arbitrario ma finito
- stato del canale: sequenza di messaggi inviati ma non ancora ricevuti

Processi: stati ed eventi

- Modello dei processi: set di stati, condizioni iniziali e set di eventi. Ogni evento può cambiare stato del processo e lo stato di un canale incidente a quel processo
- Il comportamento di ogni processo può essere descritto visualmente tramite diagrammi di transizione tra stati.

Modelli principali

- **interleaving**: presupponendo un ordinamento totale tra eventi
- **happended-before**: presupponendo un ordinamento parziale, con ordinamento totale per eventi relativi allo stesso processo
- **potenzialmente causale**: non si presuppone l'ordine di singoli processi composti da più di un thread

Modello a sovrapposizione (interleaving)

- Un'esecuzione è modellata come una sequenza globale di eventi: tutti gli eventi in esecuzione sono sovrapposti. Esempio del sistema bancario con 2 processi (server della banca e cliente)
- Modello formale:
  - *Stato globale*: prodotto incrociato di stati locali e stati dei canali
  - *Stato globale iniziale*: stati locali iniziali + canali vuoti
  - Una *sequenza di eventi* $seq = (e_i : 0 \leq i \leq m)$ è una computazione dei sistemi con modello a sovrapposizione se esiste una sequenza di stati globali $(G_i : 0 \leq i \leq m + 1)$ tale che $G0$ è uno stato iniziale e $G_{i+1} = next(G_i, e_i)$ for $0 \leq i \leq m$, dove $next(G, e)$ è la prossima funzione di stato globale, che ritorna il prossimo stato globale quando l'evento e è eseguito nello stato globale G

Modello "successo prima" (happened-before)

In un sistema distribuito reale possono essere definiti solamente ordini parziali tra eventi, definiti tramite la relazione **happened-before**

Modello:

- il processo P_i genera una sequenza di stati locali ed eventi: $s_{i,0}e_{i,1}\ s_{i,1}e_{i,2}\ ...\ e_{i,l-1}s_{i,l}$ (boh)
- la relazione *happened-before* è la più piccola relazione che soddisfa $(e \lessdot f)$ or $(e \rightsquigarrow f) \Rightarrow e \rightarrow f$
  - $\lessdot$ = relazione *immediately-precedes*, definita tra eventi dello stesso processo: $e \lessdot f$ se e precede immediatamente f
  - $\rightsquigarrow$ = relazione *remotely-precedes*: $e \rightsquigarrow f$ se e è l'evento inviato di un messaggio e f è l'evento ricevuto dello stesso messaggio.

Esecuzione in un modello happened-before: un'esecuzione o computazione in un modello *happened-before* è definito come una tupla $(E, \rightarrow)$ dove E è l'insieme di tutti gli eventi e $\rightarrow$ è un'ordinamento parziale di eventi in E tali che tutti gli eventi all'interno di un singolo processo sono totalmente ordinati. (diagramma sulle slide)

- $e \to f$ se contiene un percorso direzionato dall'evento e a f
- se due eventi non sono relazionati da $\to$ allora sono *concorrenti*: $e \vee f = \neg(e \to f) \wedge \neg(f \to e)$

Modello potenzialmente causale

- *happened-before*: ordinamento totale tra eventi dentro allo stesso processo.
    - Non è vero che tutti questi eventi abbiano relazione causa/effetto
    - La relazione di causalità è parzialmente ordinata tra eventi dello stesso processo
    - difficile o dispendiosa da determinate, consideriamo la relazione potenzialmnete causale
- **potential causality relation** $\to^p$ sul set degli eventi: la più piccola relazione che soddisfa:
    - se un evento e potenzialmente causa un altro evento f sullo stesso processo, allora $e \to^p f$
    - se e è l'invio di un messaggio e f la ricezione, allora $e \to^p f$
    - se $e \to^p f$ and $f \to^p g$, allora $e \to^p g$
    - se due eventi non sono relazionati da $\to^p$ allora sono chiamati indipendenti

Esempi: un processo riceve 2 messaggi da porte differenti e aggiorna oggetti differenti basandosi su questi due messaggi. Un processo basato su 2 thread che accede a insiemi di oggetti mutualmente disgiunti.

Diagramma sulle slide

Modello appropriato

Un programma distribuito può essere visto come un insieme di diagrammi potenzialmente causali che può generare. Un diagramma potenzialmente causale è equivalente a un set di diagrammi happened-before. Ogni happened-before è equivalente a un set globale di sequenze di eventi.

- Interleaving: on a physical time basis, total order on all events
- Happened-before: on a logical order basis, total order on one process
- Potential causality: on a causality basis, partial order on a process

Modelli ad eventi vs. a stati

A seconda delle applicazioni, può essere che un'applicazione sia modellata meglio in termini di stati e non di eventi.

Meccanismi:

- Clock logici
- Clock vettoriali

Clock logici

Tracciano la relazione *happened-before*, dando un ordine totale che può essere successo invede dell'ordine totale che è successo.

Un clock logico C è una mappatura dall'insieme degli stati S all'insieme dei numeri naturali N con i seguenti vincoli: $\forall s, t \in S : s \preccurlyeq t$ or $s \rightsquigarrow t \Rightarrow C(s) < C(t)$. Data la definizione della relazione *happened-before*, il clock logico può essere definito dalla condizione $\forall s, t \in S : s \to t \Rightarrow C(s) < C(t)$.

Implementazione su slide.

Problema: con i clock logici $\forall s, t \in S : s \to t \Rightarrow s.c < t.c$, ma non viceversa:
$\forall s, t \in S : s.c < t.c \Rightarrow s \to t$ non è vero. Risolvibile tramite clock vettoriali.

Clock vettoriali

A vector clock is a map from S to Nk (vector of size k) with the following constraint: ∀ s,t: s → t ⇔ s.v < t.v where – s.v is the vector assigned to state s – given 2 vectors x,y: - x<ymeans∀k∈[1..N],x[k] <=y[k]and∃j∈[1..N]|x[i]<y[j] - x<=ymeans(x<y)||(x=y)

Theorem: Let s and t be states in processes Pi and Pj with vectors s.v and t.v. Then: s → t iff s.v < t.v

PCD Module 5.1

Ingegneria dei sistemi distribuiti: integrazione aziendale

Integrazione (delle applicazioni)

L'integrazione è una chiave moderna dello sviluppo software. Interessa applicazioni raramente vive in isolazione. Tutte le soluzioni di integrazione devono avere a che fare con alcune sfide fondamentali:

- **le reti sono inaffidabili**: la programmazione distribuita deve essere preparata ad avere a che fare con un insieme più largo di problemi possibili.
- **le reti sono lente**: ordine di magnitudo più lenta del fare chiamate a metodi locali
- **due applicazioni qualsiasi sono diverse**: linguaggi, OS, versioni, ecc.
- **il cambiamento è inevitabile**: avere a che fare con cambiamenti nelle applicazioni che sono integrate minimizzando le dipendenze (**loose coupling**)

Approcci principali

- **Trasferimento file**: un'app scrive i file che un'altra legge
- **Database condiviso**: app multiple condividono lo stesso schema del database, posizionato in un db fisico singolo.
- **Remote Procedure Call (RPC)**: un'app espone alcune delle sue funzionalità a cui le app possono avere accesso, come procedure remote. La comunicazione avviene in tempo reale e in modo sincrono
- **Messaging**: un'app pubblica un messaggio in un canale comune, altre app possono leggere tale messaggio dal canale in un secondo tempo. Le app devono mettersi d'accordo sul canale e sul formato. La comunicazione è asincrona.

La necessità dell'integrazione

Aziende del mondo reale: centinaia, migliaia di applicazioni sono costruite in mdoo personalizzato, acquistate da terzi o parte di sistemi obsoleti ed operanti su tanti sistemi operativi diversi.

Ragioni:

- scrivere applicazioni aziendali è difficile, creare una singola app enorme per tutta l'azienda è impossibile
- diffusione di business funzionali tra più applicazioni forniscono il business con la flessibilità di selezionare il "migliore" per ogni aspetto

Sfide dell'integrazione

L'integrazione aziendale non è un compito facile. Sfide:

- richiede uno spostamento significativo nelle politiche aziendali, stabilendo una comuniczione non solo tra più sistemi informatici ma anche tra BU e dipartimenti IT
- avere a che fare col poco controllo sull'integrazione che tipicamente hanno gli sviluppatori su app a cui partecipano
- adottare standard, non solo di sintassi ma di semantica
- deployment più complesso, monitoraggio dei problemi

Il mondo selvaggio dell'integrazione

La nozione di integrazione è veramente vasta. Alcuni tipi di integrazione:

- Portali di informazioni
- Repliche dei dati
- Funzioni business condivise
- Architetture orientate ai servizi
- Processi business distribuiti
- Integrazione B2B

Portali di informazione

Alcune aziende devono accedere a più di un sistema per rispondere a una specifica domanda o per eseguire certe funzioni business (es. verificare stato di un'ordine). I portali di informazioni **aggregano le informazioni da più sorgenti in una singola visualizzazione**, per evitare che gli utenti debbano accedere a più sistemi informativi.

Repliche dei dati

Molte aziende richiedono l'accesso agli stessi dati (es. indirizzi dei clienti per sistema customer care + accounting + logistica). Molti di questi sistemi hanno nei loro archivi i dati relativi ai clienti. Quando un utente cambia un'informazione, **tutti i sistemi devono cambiare la loro copia**. Questo può essere fatto implementando una strategia di integrazione basata sulla replica dei dati.

Funzioni business condivise

Per gli stessi motivi che portano molte aziende ad archiviare dati ridondanti, tendono anche a ridondare le funzionalità (es. controllare se un SSN è valido). Ha senso esporre queste funzioni come **funzioni business condivise implementate una volta** e disponibili come servizio per gli altri sistemi.

Architetture orientate ai servizi (SOA)

Le funzioni business condivise sono a volte chiamate anche **servizi**: una funzione ben definita universalmente disponibile e che risponde alle richieste da parte di un cliente del servizio.

Dal momento in cui un'azienda assembla una collezione di servizi, gestirli può essere critico:

- **directory dei servizi**
- **directory delle interfacce e descrizione dei contratti**

=> **Servizi di ricerca dei servizi e negoziazione**. Le SOA offusca la linea tra integrazione e applicazioni distribuite – fornendo strumenti per sfruttare servizi facilmente come chiamate di metodo.

Processi business distribuiti

Un driver di integrazione è che una singola transazione business può essere diffusa tra molti sistemi diversi. In molti casi tutte le funzioni rilevanti sono incorporate all'interno delle applicazioni esistenti. Ciò che manca è la coordinazione tra queste applicazioni. È possibile aggiungere un **componente di gestione dei processi business** che gestisca l'esecuzione di una funzione su diversi sistemi multipli esistenti.

Il confine tra SOA e business distribuito può essere sfocato: potremmo esporre tutte le funzioni business rilevanti come servizi e quindi codificare i processi business all'interno come applicazioni che accedono ai servizi tramite SOA.

Integrazione B2B

In molti casi le funzioni business non sono nella stessa/singola azienda, ma possono estendersi su multipli partner aziendali o fornitori. In questo caso, comunicazione tramite Internet o altre reti fa emergere nuove problematiche relative al protocollo di trasporto e alla sicurezza. Inoltre è molto importante che il formato dei dati sia standardizzato.

Accoppiamento (Coupling)

Accoppiamento debole (loose)

Serve a ridurre i presupposti/ipotesi che due parti (componenti, app, servizi, programmi, utenti..) fanno tra loro stesse quando scambiano informazioni.

Più ipotesi = più efficienza, ma meno tolleranza ai cambiamenti, interruzioni, meno flessibilità.

Accoppiamento forte (tight): esempio delle invocazioni locali ai metodi

- Basate su molti presupposti tra la routine chiamante e chiamata:

    - entrambi i metodi devono essere eseguiti sugli stessi processi
    - devono essere scritti negli stessi linguaggi (o VM)
    - i metodi chiamanti devono passare l'esatto numero di parametri attesi usando i tipi di dati accordati
    - la chiamata è immediata: i metodi chiamati iniziano a processare immediatamente dopo che il metodo chiamante fa la chiamata
    - i metodi chiamanti ripristineranno il processo solo quando il metodo chiamato completa
    - la comunicazione è immediata e istantanea

- Tutti questi presupposti rendono molto facile il lavoro di scrivere applicazioni strutturate bene che dividono le funzionalità in metodi individuali che possono essere chiamati da altri metodi. Il risultato è un grande numero di piccoli metodi che permettono flessibilità e riuso.
- Diversi approcci integrativi hanno aiutato a creare comunicazioni remote semplicemente impacchettando uno scambio di dati remoto con la stessa semantica dei metodi locali (RPC).
    - la semantica delle chiamate ai metodi sincroni sono molto familiari agli sviluppatori, perciò perchè non usarla?
    - utilizzando la stessa sintassi e semantica per entrambe le chiamate a metodi locali e remoti ci dovrebbe permettere di differire a tempo di deploy la decisione riguardo quali componenti dovrebbero essere eseguiti localmente e quali remotamente.

E quindi?

Il problema

Il problema è che **le chiamate remote invalidano molti dei presupposti su cui sono basate le chiamate locali**. Come risultato, astrarre la comunicazione remota in una semantica semplice di una chiamata a metodo può confondere e ingannare.

Esempio: una chiamata tramite la rete tende ad essere molto più lenta di una chiamata locale. Il metodo quindi deve aspettare? E se la rete si interrompe? Quanto dovremmo aspettare?

Diventa chiaro che l'integrazione remota porta a galla tanti problemi che un metodo locale non avra mai a che fare.

Un esempio di integrazione di applicazione aziendale

- Online banking
- Permette di depositare soldi da altre banche
- Per fare ciò, il frontend ha bisogno di essere integrato con il backend finanziario.

```
String hostname = "finance.bank.com"
int port = 80
IpHostEntry hostInfo = Dns.GetHostByName(hostName);
IpAddress address = hostInfo.AddressList[0];
IPEndPoint endpoint = new IPEndPoint(address, port)
Socket socket = new Socket(address.AddressFamily,
                           SocketType.Stream, ProtocolType.Tcp);
socket.Connect(endpoint);
byte[] amount = BitConverter.GetBytes(1000);
byte[] name = Encoding.ASCII.GetBytes("joe");
int byteSent = socket.Send(amount);
byteSent += socket.Send(name);
socket.Close();
```

Problemi e presupposti

La soluzione minimale fa tali presupposti:

- **tecnologia della piattaforma**
  - indipendenza debole dalla piattaforma - funziona solo con flussi di byte
  - rappresentazione eterogenea di numeri, oggetti, codifiche
- **locazione**
  - se muovessimo la funzione su un computer diverso su dominio diverso? Se le macchine fallissero e avessimo bisogno di altre macchine? Dovremmo cambiare il codice. Dal momento che utilizziamo funzioni remote, diventa un incubo molto velocemente. Serve trovare un modo per rendere la comunicazione indipendente da una specifica macchina della rete
- **tempo**
  - Tutti i componenti devono essere disponibili allo stesso tempo
  - TCP-IP è orientato alla connessione
  - Se qualcuna delle parti coinvolte nella comunicazione non fosse disponibile in tal momento, il dato non può essere inviato
- **formato dei dati**
  - La lista dei parametri e i tipi devono combaciare
  - Se volessimo inserire un terzo parametro, avremmo bisogno di modificare sia il mittente che il destinatario

Rimuovere le dipendenze

- **dipendenti dalla piattaforma**: utilizzando formati dati standard (XML, JSON, ecc...)
- **relative alla locazione**: invece che inviare informazioni a una macchina, le si mandano ad un **canale** indirizzabile. Un canale è un indirizzo logico su cui mittente e destinatario devono mettersi d'accordo senza conoscere la relativa identità.
- **relative al tempo**: si accodano richieste inviate finchè la rete e il sistema ricevente non è pronto. Accodare le richieste dentro al canale richiede che i dati vengano spezzati in messaggi autocontenuti affinchè il canale conosca quanti dati deve bufferare ed inviare in qualsiasi momento.
- **sul formato dati**: permettendo trasformazioni del formato dati

Interazione poco accoppiata

I meccanismi come i formati dati comuni, la comunicazione asincrona su canali con code e i trasformatori aiutano una soluzione fortemente accoppiata a farla diventare poco accoppiata.

Svantaggio principale: **complessità addizionale**. Soluzioni con codice più lungo, modelli di programmazione più complessi che rendono la progettazione, la costruzione e il debugging più complesso.

Elementi base

- **Canale di comunicazione** per muovere le informazioni da un'app all'altra
- **Messaggi** con dati spezzettati, con un significato già preaccordato
- **Traduzione** per supportare i formati dati interni delle varie applicazioni
- **Instradamento** per muovere i dati verso un determinato posto e indirizzo
- **Funzioni di gestione del sistema** per monitorare il flusso dei dati, assicurarsi che tutte le app e i componenti siano funzionanti, riportando condizioni di errore.

PCD Module 5.3

Ingegneria dei sistemi distribuiti: EDA e ESB

Event-Driven Architectures (EDA)

Da Publish/Subscribe agli Event Services

Il modello Publish/Subscribe visto nel modulo MOM è la base per impostare gli **Event Services** e le **Architetture guidate dagli eventi (EDA)**.

- I messaggi rappresentano eventi come cose che succedono dentro o fuori il sistema o l'azienda (problemi, opportunità, soglie, deviazioni...)
- Il significato è a livello dominio/azienda (non applicativo) (es. inventario di un rivenditore)
- Compongono/correlano eventi per semantiche di alto livello (congestione del traffico, inquinamento e traffico...)

Modello di comunicazione effettiva in diversi domini applicativi (es. sensori che generano dati, notificato come eventi)

Architettura guidata dagli eventi

Architettura software adottata nel modello P/S per disseminare **eventi** immediatamente o a tutte le parti interessate. Promuove la produzione, la ricezione, il consumo e la reazione agli eventi. Le parti interessate valutano l'evento e opzionalmente compiono azioni.

Un evento è un **cambiamento di stato**, es. quando si acquista una macchina, il suo stato passa da "in vendita" a "venduta". L'architettura software deve trattare questo cambio come evento la cui occorrenza può essere far saputa ad altre applicazioni.

Gli eventi sono notificati tramite **notifiche degli eventi**, ovvero messaggi asincroni prodotti, pubblicati, propagati, trovati e consumati quando accade un evento. Gli eventi non viaggiano, capitano solamente.

Concetti principali di EDA

- **emettitori di eventi** (agenti): hanno la responsabilità di catturare, gestire e trasferire eventi. Un emettitore non conosce i consumatori degli eventi.
- **consumatori di eventi** (assorbitori): hanno la responsabilità di applicare una reazione appena un evento si presenta. La reazione può o non può essere completamente fornita dal consumatore.

- **canali degli eventi**: condotti in cui gli eventi sono trasmetti dagli emettitori ai consumatori. La conoscenza della corretta distribuzione degli eventi è esclusivamente presentata all'interno del canale. L'implementazione fisica di tali canali può essere basata su componenti tradizionali, e.s middleware orientati ai messaggi.

Accoppiamente estremamente leggero

Architettura software effettiva per progettare sistemi i cui componenti/servizi sono necessari per essere estremamente poco accoppiati. I componenti che sono sorgenti di un evento non hanno bisogno di essere accoppiate o conoscere il consumatore di una notifica dell'evento.

Proprietà: apertura, dinamismo, flessibilità.

Stili di processazione degli eventi

- Simple Event Processing, su eventi notabili
- Stream Event Processing, eventi notabili e ordinari
- Complex Event Processing, confluenza di eventi

Integrazione con SOA

Una EDA può essere usata a complemento di **architetture orientate ai servizi (SOA)**.

Mentre SOA generalmente si adatta meglio a uno scambio di richieste/risposte, EDA introduce capacità di **processi asincroni a lunga durata**. I servizi possono essere attivati tramite **trigger** eseguiti su eventi in arrivo. EDA a volte è chiamato *SOA guidato dagli eventi*.

Richiamo sulle funzionalità di SOA

SOA: tutte le funzioni o servizi sono definiti tramite un linguaggio descrittivo e dove le loro interfacce sono ritrovabili in una rete.

Permette di muoversi fuori dall'*approccio a silo*, dove ogni dipartimento costruisce il suo sistema senza nessuna conoscenza di cosa sia stato fatto da altri. L'approccio a silo porta a situazioni inefficienti e costose dove la stessa funzionalità è sviluppata, prodotta e mantenuta molte volte.

SOA è basato su un portfolio di servizi condivisi tra l'organizzazione: fornisce un modo per riusare e integrare efficientemente asset già esistenti.

Caratteristiche di SOA

- **Interazioni poco accoppiate**: i servizi sono invocati indipendentemente dalla loro tecnologia e posizione

- **Comunicazioni 1:1**: un servizio specifico è invocato da un consumatore per volta. Le comunicazioni sono bidirezionali.
- **Trigger basati sui consumatori**: il flusso di controllo è inizializzato dal client (il consumatore)
- **Sincrono**: SOA è a volte basato su un protocollo convenzionale request/reply basato sui messaggi, a volte implementato utilizzando un approccio sincrono.

Aggiungere caratteristiche di EDA

- **Interazioni disaccoppiate**: i pubblicatori non sono a conoscenza dell'esistenza di sottoscrittori di eventi
- **Comunicazioni molti-a-molti**: messaggi P/S dove un evento specifico può impattare molti sottoscrittori
- **Trigger basati sugli eventi**: flusso di controllo che è determinato dal destinatario, basato su un evento postato
- **Asincrono**: supporta operazioni asincrone tramite event messaging, usato per comunicare tra due o più processi applicativi. La comunicazione è iniziata da una notifica di evento.

Enterprise Service Bus (ESB)

- **ESB** combina approcci event-driven + service-oriented per semplificare l'integrazione di BU, unendo piattaforme e ambienti eterogenei.
- Agisce come livello intermediario per abilitare la comunicazione tra diversi processi applicativi.
    - un servizio prodotto su un ESB può essere eseguito da un consumatore o un evento
    - supporta sincrono e asincrono, facilitando le interazioni tra uno o più stakeholders

- Diverse implementazioni da diverse aziende propongono ESB come la chiave per risolvere l'integrazione aziendale. Commerciale (IBM WebSphere, SAP Process Integration, Windows Azure Service Bus) o open source (Apache Camel, JBoss ESB, Mule ESB...)

Servizi di ESB

Non ci sono specifiche ufficiali su cosa debba essere un'implementazione ESB, ma è comunemente accettato che debba fornire servizi di **trasporto**, **eventi** e **mediazione**.

- **Servizi di trasporto**: devono assicurare l'ivnio del messaggio tra processi aziendali interconnessi tramite un bus aziendale. Include l'instradamento basato sui contenuti (msg). Parte di un ambiente mission-critical, questi servizi sono transazionali, sicuri e monitorati.
- **Servizi di eventi**: forniscono capacità di cattura, esecuzione e distribuzione degli eventi. Relativi alla nozione di processo degli eventi.
- **Servizi di mediazione**: abilita l'interoperabilità tra servizi sul bus.

Servizi di mediazione

- Assicura il protocollo necessario per integrare sistemi eterogenei di corrispondenza
    - come due diversi servizi non è necessario utilizzare lo stesso protocollo di trasporto, il servizio di mediazione si prende cura della trasformazione da un protocollo a altro, in modo che la comunicazione è possibile.
    - l'interruttore protocol è trasparente per tutti i partecipanti servizi di una transazione di affari

- Permette di trasformare il contenuto di qualsiasi messaggio allo scopo di integrazione aziendale

    - assicura che i dati che transita attraverso il bus siano comprensibili da qualsiasi processo
    - consente l'aumento del contenuto di arricchire un messaggio con informazioni aggiuntive

  Esempi su slide

Imprese come sistemi reattivi

L'integrazione di EDA con SOA consente di aumentare SOA tempi di risposta. Aggiunta la capacità o reagendo agli ambienti imprevedibili e asincroni

Caratteristiche di sistemi reattivi

- tempi di risposta
- resilienza
- elasticità (scalabilità)
- basato sui messaggi

PCD Module 5.4

Ingegneria dei sistemi distribuiti: il Cloud

Introduzione

CC è una delle scelte principali nella progettazione di applicazioni distribuite su larga scala, grazie alla scalabilità, apertura e eterogeneità, disponibilità.

Cloud computing

Si spostano i servizi, per questioni di costi e vantaggi, fuori in strutture centralizzate, interne o esterne, trasparenti alla locazione. Driver principali: Google, Yahoo, Amazon, Microsoft...

Rendendo i dati disponibili sul cloud, possono essere acceduti più facilmente e ubiquamente ad un costo minore e con più opportunità di collaborazione, integrazione e analisi su una piattaforma comune condivisa. Ma ci sono anche problematiche di sicurezza e performance.

Da CapEx a OpEx

In passato, per sviluppare un servizio ci voleva una grande spesa capitale (CapEx) per costruire l'infrastruttura che reggesse una domanda alta.

Il CC permette di spendere tracciando le risorse e spostandosi verso OpEx, spese operative, permettendo allo stesso tempo scalabilità e flessibilità.

Grazie al CC è possibile sviluppare progetti da zero senza dover pensare a un'infrastruttura server che potrebbe poi non servire.

Background storico

.....

La rete è il computer

Dalla rete al web

Da statico a dinamico

Dai cluster al cloud

Modelli di servizio

- **Client cloud**: browser, emulatore, app mobile, ecc.

che si interfaccia con:

- **Software as a Service**: servizi, Google Apps, salesforce, WebEx...
- **Platform as a Service**: fondamenta, Google App Engine, Azure, Heroku...
- **Infrastructure as a Service**: infrastruttura, Amazon EC2, Google Computing Engine, Open Stack

Cosa posso controllare

OPE = On premise environment

```
                        OPE     IaaS    PaaS    SaaS
Applicazioni            x       x       x
Dati                    x       x       x
Runtime             x       x
Middleware          x       x
OS                      x
Virtualizzazione        x
Server                  x
Storage             x
Networking          x
```

Tecnologie sfruttabili

- Storage poco costoso
- CPU con larghezza di banda tale da supportare computazioni importanti poco costoso
- Algoritmi e tecnologie client sofisticati (HTML, CSS, AJAX, REST)
- Client broadband
- SOA
- Implementazioni infrastrutturali dei provider
- Virtualizzazione commerciale

Modelli di fornitura

Pubblico

Quando i servizi sono forniti su una rete aperta all'uso pubblico. Ci sono poche differenze col cloud privato,

a parte le misure di sicurezza. Generalmente, i provider possiedono e gestiscono le infrastrutture e le rendono disponibili tramite Internet (senza connettività diretta). Es. Google, Amazon AWS, Microsoft, ecc...

Privato

Infrastruttura operata solo per una singola organizzazione, gestita internamento o da una terza parte, ospitata internamente o esternamente.

Creare un cloud privato richiede un livello significativo e grado di impegno nel virtualizzare l'ambiente aziendale, e richiede che le aziende rivalutino decisioni riguardanti risorse esistenti. Se fatto bene può migliorare il business, ma ogni passo comporta problematiche di sicurezza che devono essere prese in considerazione. I datacenter proprietari sono generalmente costosi.

Questi asset devono essere aggiornati periodicamente --> spese addizionali. Critiche forti poichè non ci sono dei gran vantaggi.

Comunità

Condivide l'infrastruttura tra diverse organizzazioni con uno specifico scopo (es. sicurezza, giurisdizione, ecc..), gestita internamente o da una terza parte e ospitata internamente o esternamente. I costi sono spalmati su un numero di utenti minore rispetto a un cloud pubblico, ma più di un cloud privato.

Ibrido

Composizione di due o più cloud (private/pubbliche/community, da provider diversi) che rimangono entità distinte ma sono collegate assieme, offrendo i benefici di modelli di produzione multipli.

Caratteristiche e funzionalità

- **Agilità** migliora con l'abilità dell'utente di rieseguire provisioning di risorse tecnologiche infrastrutturali.
- **Accesso alle API** da parte del software, permettendo alle macchine di interagire con il software cloud nello stesso modo in cui un'interfaccia utente tradizionale facilita l'interazione tra umani e computer. I sistemi cloud tipicamente usano API basate su **REST**.
- **Costo**: i provider dicono che il cloud riduca i costi. Il modello di fornitura di una cloud pubblica converte le spese capitali in spese operazionali. Ciò abbassa gli ostacoli per usufruire di tali servizi, poichè l'infrastruttura è fornita da terze parti e non ha bisogno di essere acquistata per un uso infrequente.
- **Indipendenza dei dispositivi e delle locazioni**: permette agli utenti di accedere al sistema tramite un browser, non ha importanza la posizione o lo strumento che usano.
- **Tecnologie di virtualizzazione**: permettono la condivisione dei server e dello storage. Le applicazioni possono essere migrate facilmente da un server fisico ad un altro.
- **Multilocazione**: permette la condivisione di risorse e costi su un largo numero di utenti, permettendo così la centralizzazione di infrastrutture con costi minori (es. affitto), la capacità di carico a picco

aumenta, miglioramento dell'utilizzo e dell'efficienza per sistemi usati al 10-20%.
- **Affidabilità** migliora con l'uso di siti multipli ridondati, rendendo un cloud ben progettato adatto alla continuità e al recupero dai disastri.
- **Scalabilità** ed elasticità tramite fornitura dinamica on-demand di risorse fornite in tempo reale all'utente
- **Perfomance** monitorate, architetture poco accoppiate e consistenti costruite utilizzando i web service come interfaccia di sistema.
- **Sicurezza**: può migliorare grazie alla centralizzazione, ma ci sono ancora dei dubbi sulla perdita di controllo su certi dati sensitivi, e la mancanza di sicurezza per stored kernel.
- **Manutenzione** di sistemi cloud è facile, perchè non hanno bisogno di essere installati su ogni computer e possono essere acceduti da posti diversi.

Caratteristiche essenziali (NIST)

- **Self-service on daemand**: un consumatore può unilateralmente fornire capacità di calcolo, ad esempio tempo server e storage in rete, come necessario automaticamente senza richiedere l'interazione umana con ogni fornitore.
- **Accesso vasto alla rete**: le funzionalità sono disponibili sulla rete e accessibili tramite meccanismi standard che favoriscono l'eterogeneità.
- **Pooling di risorse**: le risorse del provider sono condivise per servire utenti multipli usando un modello multilocato, con risorse differenti fisiche e virtuali assegnate dinamicamente e riassegnate in accordo con le richieste del consumatore.
- **Elascitià rapida**: le funzionalità possono essere fornite e rilasciate elasticamente, in certi casi automaticamente, per scalare rapidamente in grande e in piccolo in base alla domanda. Al consumatore le risorse appaiono illimitate e possono essere regolate in qualsiasi momento.
- **Servizio misurato**: i sistemi cloud controllano e ottimizzano automaticamente l'uso delle risorse sfruttando capacità di conteggio ad un certo livello di astrazione appropriato per il tipio di servizi. L'uso di risorse può essere monitorato, controllato e segnalato, fornendo trasparenza sia al provider che al consumatore.

SaaS

Gli utenti hanno accesso ai software applicativi e al database, disponibili solo online - via browser o app - come servizio. I fornitori gestiscono l'infrastruttura. Riferito come "software on demand" e con prezzi pay-per-use o tramite abbonamenti.

Nel modello SaaS, i fornitori cloud installano e operano software applicativo nel cloud, e gli utenti cloud accedono al software dai client cloud. Gli utenti cloud non gestiscono l'infrastruttura e e la piattaforma su cui l'applicazione viene eseguita: ciò elimina il bisogno di installare e eseguire applicazioni nel computer dell'utente, semplificando manutenzione e supporto.

Le applicazioni cloud sono diverse da altre applicazioni in termini di scalabilità, che può essere ottenuta

clonando task su multiple macchine virtuali a tempo di esecuzione per incontrare la domanda di cambio lavoro.

Per accomodare un numero maggiore di utenti cloud, le applicazioni possono essere multilocate: ogni macchina serve più di un'organizzazione ed è comune riferirsi a tipi speciali di applicazioni cloud con nomi simili: desktop-aaS, business proces-aaS, test-aaS...

Il modello di prezzo per applicazioni SaaS è tipicamente una quota mensile o annuale per utente, così il prezzo può essere scalabile e aggiustabile se gli utenti vengono aggiunti o rimossi in qualsiasi momento.

Pro e contro

- **Pro**: SaaS permette alle aziende la riduzione dei costi operativi IT, mandando la manutenzione HW e SW in outsourcing, così da riallocare i costi relativi a gestione HW + SW + dipendenti. Con applicazioni ospitate centralmente, gli aggiornamenti possono essere rilasciati senza bisogno che gli utenti installino nuovo software.
- **Contro**: i dati degli utenti sono memorizzati sui server cloud del provider. Perciò, potrebbero esserci accessi non autorizzati ai dati. Per tale motivo, gli utenti stanno adottando sempre di più sistemi di gestione intelligenti delle chiavi di terze parti per mettere in sicurezza i propri dati.

IaaS

Nel modello più essenziale di servizio cloud, i provider di IaaS offrono macchine fisiche o virtuali e altre risorse come servizio. Un ipervisore (OpenStack, Xen, HyperV) viene eseguito sulla macchina virtuale come ospite. Insiemi di ipervisori all'interno del sistema di supporto operazionale del cloud possono supportare un numero elevato di macchine virtuali e l'abilità di scalare servizi in grande e in piccolo, in accordo con i requisiti variabili dei clienti.

Le cloud IaaS a volte possono offrire risorse aggiuntive come un disco virtuale, una libreria immagini, raw block storage, memorizzazione di file e oggetti, firewall, bilanciatori di carico, indirizzi IP, VLAN, bundle software.

I provider cloud IaaS forniscono queste risorse **on-demand** dai loro grandi pool installati nei datacenter. Per la connettività su larga scala, i clienti possono scegliere sia l'internet che il carrier clouds (VPN dedicate) ??

Questo paradigma è stato recentemente esteso a sensori e attuatori as a Service.

Per produrre le applicazioni, gli utenti cloud installano immagini degli OS e il software sull'infrastruttura cloud. In questo modello, gli utenti cloud aggiornano e mantengono l'OS e i software. I provider tipicamente fatturano i servizi IaaS su una base di utility utilizzate: quantità di risorse allocate e consumate.

PaaS

A questo livello di servizio, i venditori hanno cura dell'infrastruttura sottostante, dando agli sviluppatori solamente la piattaforma su cui ospitare la propria applicazione. Gli sviluppatori controllano l'applicazione, non come in SaaS. Gli sviluppatori non hanno a che fare con l'infrastruttura hardware e i problemi riguardanti OS e sistema (aggiornamenti, patch, ecc.).

Le app create su piattaforma PaaS sono applicazioni SaaS: gli utenti di tali app non hanno controllo sul codice. Un utente PaaS è uno sviluppatore di SaaS.

In comune tra le varie PaaS

- SOA, basato su REST
- Servizi disponibili fornite dalla piattaforma tramite API per
  - memorizzare e ritrovare dati
  - eseguire e gestire task a lungo termine
  - comunicazione
  - configurazione app e gestione

Esempi di PaaS

Vedi slide

Orleans Framework

Modello ad attori per organizzare le app cloud, chiamate grains. Basato su MS Azure.

Google App Engine, caso di studio

- Piattaforma e infrastruttura per sviluppo ed esecuzione di app cloud. Insieme di servizi ad alto livello accessibile tramite le App Engine API. Servizi GAE = Chiamate a sistema degli OS.
- Supporto a diversi linguaggi (Java e JVM-derivati, Python, Go) e piattaforme - **SDK open source**.
- Modello di sicurezza basato sul **sandboxing**: le applicazioni GAE vengono eseguite in un ambiente ristretto chiamato sandbox. Alcune azioni non sono permesse (es. apertura file locale per scrittura, apertura socket,...). In alternativa devono essere utilizzati i Servizi GAE
- Infrastruttura e runtime totalmente gestito da Google, il sistema si autoscala per allocare più istanze dell'app se richiesto. QoS negoziato al setup dell'account.
- Server di sviluppo fornito per testare il loro codice (con limitazioni) prima di entrare in produzione
- Strumenti di profilazione e amministrazione + servizi
- Progetti backend open source compatibili con GAE - AppScale, TyphoonAE, ...

Servizi principali

- Modello di interazione orientato ai servizi - REST style

- Categorie base
  - memorizzazione dati, recupero e ricerca
  - gestione dei processi e computazioni
  - comunicazione
  - configurazione e gestione app

Memorizzazione dati, recupero e ricerca

- **Google Cloud SQL**: servizio web completamente gestito che permette di creare, configurare e usare database relazionali ospitati sul Google Cloud
- **Datastore**: memorizzazione di dati per mezzo di oggetti senza schema, fornisce uno storage robusto e scalabile per le app, API ricche per la modellazione dei dati, linguaggio di query SQL-like chiamato GQL.
- **Blobstore**: permette alle applicazioni di fornire oggetti dati pesanti (video, immagini, ...) che sono troppo grandi per essere memorizzati in Datastore.
- **Memcache**: memoria cache distribuita e in memoria che può essere usata per migliorare le performance.
- **Dedicated Memcache**: fornisce una capacità di cache fissa assegnata esclusivamente per l'applicazione.
- **Logs**: fornisce accesso programmatico ai log dell'app e richieste.
- **Search**: permette all'applicazione di eseguire ricerche simil-Google su dati strutturati come testi, HTML, atom, numeri, date, geolocation.
- **HRD Migration Tool**: migra i dati delle applicazioni ospitate nel vecchio Datastore/Blobstore al nuovo Datastore.
- **Google Cloud Storage Client Library**: permette all'applicazione di leggere file da e scrivere file sui bucket del Google Cloud Storage, con gestione interna degli errori.

Comunicazione

- **Channel**: crea una connessione persistente tra l'app e i server Google, così da poter inviare messaggi a client Javascript in real time senza polling.
- **Google Cloud Endpoints**: permette la generazione automatica di API, rendendo più facile creare un backend web per client web e mobile.
- **Mail**: invia messaggi a amministratori e utenti con account Google, e riceve mail da diversi indirizzi.
- **URL Fetch**: usa l'infrastruttura di rete di Google per fornire richieste a URL web HTTP e HTTPS.
- **XMPP**: permette ad un'applicazione di inviare e ricevere messaggi in chat verso e da ogni servizio di messaggistica compatibile con XMPP.
- **Sockets**: permette di supportare sockets in uscita utilizzando le librerie specifiche per il linguaggio built-in.

Gestione dei processi e computazioni

- **Task Queue**: permette alle applicazioni di svolgere lavoro fuori da una richiesta dell'utente, e organizzare tale lavoro in piccole unità discrete chiamate "task", da esegurie in un secondo momento.
- **Scheduled Tasks**: permette alle applicazioni di configurare task regolarmente programmati per operare ad un certo orario o ad intervalli regolari.
- **Images**: manipola, combina, migliora immagini, converte in diversi formati, permette di chiedere i metadati di un'immagine (es. dimensione o frequenza colori).
- **MapReduce**: adattamento ottimizzato del modello di computazione MapReduce per computazioni distribuite efficiente su dataset larghi.

Configurazione e gestione app

- **App Identity**: fornisce accesso nel codice all'identità dell'applicazione, fornisce un framework per verificare l'identità tramite OAuth.
- **Capabilities**: fornisce la rilevazione delle interruzioni e manutenzione programmata per API specifiche e servizi, in modo che l'applicazione può escluderli o informare gli utenti.
- **SSL for Custom Domains**: permette alle applicazioni di essere servite con HTTPS e HTTP tramite un dominio personalizzato (invece di appspot.com).
- **Users**: permette alle applicazioni di loggarsi tramite Google Accounts o OpenID, e di memorizzare gli utenti tramite UUID
- **Modules**: permette ai dev di fattorizzare grandi app in componenti logici che condividono servizi stateful e comunicano in modo sicuro.
- **Multitenancy**: rende facile la compartimentazione dei dati per servire più organizzazioni client per una singola istanza dell'applicazione
- **Remote**: permette ad app esterne di accedere ai servizi App Engine in modo trasparente.
- **Traffic Splitting**: permette di lanciare nuove funzionalità in modo graduale (rollout), anche tramite AB Testing.

Dinamica delle richieste

- Ogni richiesta è prima ispezionata da un frontend di App Engine. Ce ne sono più di una e c'è un bilancio di carico che gestisce l'allocazione di richieste alle macchine
- Il frontend spedisce la richiesta al relativo handler dell'applicazione (tramite nome a dominio). L'obiettivo può essere un file statico o un handler.
- Le richieste gestite dall'handler sono reindirizzate ai server dell'app. Un server specifico viene selezionato e un'istanza dell'applicazione avviata. Se c'è già un'istanza in esecuzione può essere riutilizzata.
- L'handler della richiesta appropriato viene invocato e genera una risposta per la richiesta, basata sul codice dell'app
- Strategie di bilanciamento del carico e distribuzione delle richieste di applicazione hanno lo scopo di garantire un throughput elevato di richieste, con handler veloci. Quante istanze debbano essere avviate e quante richieste distribuite dipende dal traffico dell'applicazione e dall'uso di pattern.
- L'applicazione esegue il codice in un ambiente sandboxed

- Il server attende affinchè l'handler termina e ritorna la risposta al frontend, completando la richiesta
- Il frontend costruisce la risposta finale al client

Osservazioni

- Non viene garantito che due richieste vengano eseguite sulla stessa macchina, anche se le richieste arrivano una dopo l'altra dallo stesso client.
- Istanze multiple di app diverse possono essere eseguite sulla stessa macchina senza che una dia fastidio all'altra.

Ambiente runtime

- La sandbox è un ambiente isolato, basato su Python, Java o Go.
- Sandboxing è una scelta chiave: previene alle app di eseguire codice malevolo e permette ad App Engine di eseguire un bilanciamento automatico dei carichi di lavoro
- Restrizioni: nessun accesso diretto all'OS, uso diretto dei thread non permesso.