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