Congresso A.I.C.A 1975, Genova

G. Casaglia*, N. Ljitmaer**, L. Pessina*

MECCANISMI DI COMUNICAZIONE NEI SISTEMI MODULARI

Sommario: Si considerano sistemi software composti da un insieme di processi cooperanti in evoluzione ed asincroni, che si sincronizzano tra loro mediante scambio di messaggi. In questo contesto è possibile individuare una classe di meccanismi di scambio messaggi, che permetta di realizzare sistemi modulari. Tali meccanismi vengono descritti mediante uno schema generale, che fa riferimento alle teorie di Dennis. Quindi, riferendosi allo schema generale, vengono descritti e confrontati due diversi sistemi ottenuti con: a) il port approach, orientato verso la trasmissione di messaggi tramite connessioni unidirezionali e mailboxes; b) l'interface approach, orientato verso lo scambio di messaggi tramite la gestione organizzata di strutture di interfaccia.

Introduzione

La modularità dei sistemi di calcolo in generale e dei sistemi software in particolare è una caratteristica che spesso viene descritta mediante i vantaggi, che essa permette di ottenere (estensione delle prestazioni, modifica delle prestazioni, facilità di debugging, di manutenzione, etc.).
La realizzazione di queste proprietà produce sensibili influenze sia sulla struttura dei programmi, sia sulle caratteristiche dei sistemi di calcolo.
Per affrontare quindi il problema della realizzazione dei sistemi software modulari è necessario arrivare ad una definizione strutturale di modularità, ovvero alla definizione delle regole che ciascun modulo deve rispettare e dei meccanismi di collegamento dei moduli tra di loro. I sistemi software che si considerano sono composti da un insieme di processi cooperanti in evoluzione e asincroni, che si sincronizzano tra loro mediante scambio di messaggi.
Scopo del lavoro che si presenta è quello di analizzare due tipi di meccanismi di collegamento di moduli, al fine di stabilirne l'applicabilità per ottenere:

- la armoniosa cooperazione dei processi
- la modularità del sistema
- la protezione dei processi.

Le definizioni dei concetti fondamentali (processo, processor, sequenzialità, parallelismo, modularità, etc.) costituiscono il supporto indispensabile per impostare l'analisi dei meccanismi di comunicazione nei sistemi modulari. La parte relativa ai processi e alla realizzazione di sistemi composti di processi cooperanti viene considerata nel primo capitolo, assumendo come base quanto esposto da Brinch Hansen (Han2 73).
Nel secondo capitolo viene invece discussa la modularità seguendo le definizioni di Dennis (Den4 71).
In base alle considerazioni esposte nei primi due capitoli si ottiene uno schema generale di interconnessione tra moduli, mediante il quale si realizza un sistema software modulare, in cui i processi cooperano armoniosamente.
Vengono quindi presi in considerazione due esempi di meccanismi di interconnessione tra moduli:

- "port approach" (Anc2 73, Bal 71) , basato tipicamente sullo scambio di messaggi tramite porte, collegate con connessioni unidirezionali;
- "interface approach" (Weis 73), basato invece sulla gestione, esterna ai moduli, delle strutture dati, che costituiscono le interfacce tra i moduli stessi.

Nell'ultima parte, infine, questi due meccanismi vengono esaminati al fine di verificare se, e in quali condizioni, realizzano un sistema modulare di processi concorrenti protetti, per cui sia valido il teorema di Dennis sulla funzionalità. Tali condizioni, insieme a condizioni di generalità, flessibilità e complessità dei meccanismi, permettono di eseguire una valutazione comparata dei due approcci, e, in particolare, permettono di verificare che il "port approach" può essere ricondotto, con certi vincoli, all'"interface approach".

---

*) Ricerca & Sviluppo - Olivetti, Ivrea
**) Istituto di Elaborazione dell'Informazione del CNR, Pisa

1. PROCESSI CONCORRENTI E COOPERANTI

1.1. Processo e processor

Il concetto di processo e quello di processor sono strettamente collegati.

Per processo intendiamo (Ran 73) una tripla (S,f,s), dove S è lo spazio degli stati del sistema di elaborazione, f è una funzione d'azione in questo spazio (cioè una funzione che fa passare da uno stato a un'altro) e s è un sottoinsieme di S, che determina gli stati iniziali del processo.

Per "processor" intendiamo (Ran 73, p.9) la coppia (D, I) dove D è un dispositivo fisico (cioè un "device" fisico), che può essere posto in stati iniziali specificati, e I è un'interpretazione del suo stato fisico, la quale indica a quali istanti di tempo e in che modo il "device" rappresenta stati successivi.

Se per computazione intendiamo (Ran 73, p.9) una sequenza di stati dello spazio degli stati, possiamo affermare che, grazie all'interpretazione, un "processor" possiede un suo spazio degli stati e un insieme di computazioni (prendendo tutte le sequenze di calcolo a cui siamo interessati). Considerando un processo che abbia questo stesso insieme di computazioni si può dire che esso è un modello del "processor", mentre il "processor" è una realizzazione del processo (Mon73, p.333).

Da queste definizioni risulta quindi che un processor è strettamente legato al concetto di istanti distinti di tempo, mentre nel caso di processi si parla solo di evoluzione temporale. Infatti nella definizione di processo non si fa riferimento al tempo, ma ad una sequenza ordinata di stati.

Una funzione d'azione è deterministica in senso stretto (Ran 73, p. 8), se è a un sol valore su tutto il suo dominio. Chiamando l'insieme degli stati formanti il dominio $X=\{x_1,x_2,\ldots x_n,\ldots\}$ e l'insieme degli stati formanti il codominio $Y=\{y_1,y_2,\ldots y_n,\ldots\}$, la funzione d'azione f è strettamente deterministica, se $\forall y_1,y_2$ tali che $y_1 \neq y_2$, risulta essere $y_1=f(x_1)$ e $y_2=f(x_2)$ con $x_1 \neq x_2$. La funzione trasforma cioè un elemento del suo dominio sempre in uno stesso ben determinato elemento del suo codominio, indipendentemente dall'istante in cui la si applica.

1.2. Processi paralleli sequenziali

I processi possono essere combinati tra di loro in modo sequenziale o parallelo, e questa loro combinazione può a sua volta venir considerata un unico processo più complesso.

In fig. 1.2.1. il grafo orientato descrive un insieme di processi $P_i, i \in 1,5$ in cui P_2 evolve sequenzialmente rispetto a P_1, mentre P_3 è in parallelo con P_4.

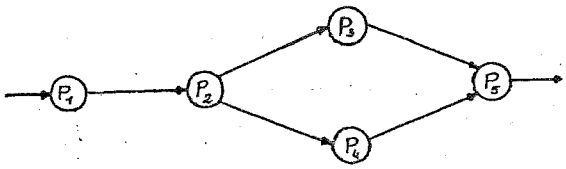

Fig. 1.2.1.

Nel caso di processi sequenziali tutti gli stati finali di un processo sono stati iniziali di quello successivo, per cui per poter iniziare l'esecuzione di quest'ultimo bisogna prima aver portato a termine quella del processo precedente.

Nel caso di processi paralleli, invece, ogni processo agisce sulle proprie variabili di stato a istanti che sono indipendenti dall'evoluzione degli altri processi. Si parla di parallelismo proprio per mettere in evidenza l'indipendenza di tali processi e il fatto che possano venir eseguiti contemporaneamente. Naturalmente un sistema di processi paralleli può venir trasformato in uno di processi sequenziali, fissando arbitrariamente l'ordine di esecuzione dei processi paralleli.

Processi che procedono in parallelo e che competono per risorse comuni vengono detti concorrenti (Anc3 73). I processi concorrenti possono a loro volta essere di due tipi : disgiunti o cooperanti. Sono cooperanti se collaborano agendo su variabili comuni per raggiungere un unico scopo, e disgiunti se non agiscono su variabili comuni.

In generale i processi possono interagire in tre modi distinti (Anc3 73) : a) cooperano , se la interazione è prevedibile e desiderata; b) interferiscono, se le interazioni sono non prevedibili e non desiderate; c) sono in competizione, se le interazioni sono prevedibili, ma non desiderate.

Essendo un sistema software un insieme di componenti software, che aggiunte a un sistema ospite preesistente lo estendono (Den2 71), si può anche affermare che un sistema software è un insieme di processi cooperanti, in evoluzione.

1.3. Sincronizzazione

Un sistema software che realizzi un ambiente di processi concorrenti (ad es. per ottenere la multiprogrammazione) è naturalmente molto più complesso di uno che realizzi un ambiente di processi sequenziali. Nascono infatti dei problemi di sincronizzazione relativi ai punti nei quali i processi concorrenti (cooperanti) hanno delle interazioni. La sincronizzazione può avvenire per interazione o per cooperazione.

A seconda del tipo di problema che si deve affrontare è più conveniente utilizzare certe primitive piuttosto che altre (Han2 73), dove per primitiva si intende un'operazione, la cui esecuzione non è interrompibile al livello linguistico in cui viene usata.

Le primitive usate da Brinch Hansen (Han2 73) per risolvere i problemi di sincronizzazione e tramite le quali ha esteso il linguaggio Pascal (introdotto da Wirth) al "Concurrent Pascal", sono le seguenti :

1) regioni critiche: esclusione mutua
2) semafori: scambio di segnali temporali
3) buffer per messaggi: scambio di dati
4) regioni critiche condizionali: blocco condizionato di un processo
5) code degli eventi: esplicito scheduling di processi

Tutte queste primitive sono equivalenti da un punto di vista logico, in quanto sono tutte in grado di risolvere qualunque problema di sincronizzazione. Però da un punto di vista pratico ognuna di esse è particolarmente adatta a risolvere un certo tipo di situazione come evidenziato nei punti 1÷5).

2. MODULARITA'

Il termine modularità è correntemente usato con diversi significati; spesso tali significati sono molto lontani dalla definizione di modularità che qui daremo.

Una delle considerazioni più importanti riguardo alla caratteristica di un sistema di essere modulare o no , è che la modularità è sempre limitata entro un certo contesto . Con la definizione di modularità da noi assunta, vedremo come il termine "livello linguistico" (par. 2.1.) permetta di chiarire quale sia il contesto in cui si realizza tale modularità.

Una volta introdotta la definizione di modularità e di modulo vengono definiti i metodi mediante i quali è possibile collegare tra loro moduli, in modo tale che producano dei sistemi di processi cooperanti; verranno infine definite le condizioni, che producono la determinatezza di tali sistemi.

Poichè i vantaggi possono però essere ridotti dalla complessità dei meccanismi di sincronizzazione, che per essere generalizzati possono essere costosi, sia in tempo di esecuzione, sia in spazio di memoria, è necessaria una accurata valutazione delle prestazioni e della implementazione dei sistemi di scambio delle informazioni tra processi.

2.1. Livelli linguistici

Un sistema di elaborazione può essere definito mediante livelli linguistici. Ad esempio in un sistema operativo con struttura gerarchica il livello più basso e semplice di tale forma gerarchica è rappresentato dall'hardware del sistema. Questo è in grado di interpretare ed eseguire un numero limitato di istruzioni, che sono le sue istruzioni macchina. Sopra questo è implementato il primo livello software, detto nucleo o "kernel", che consente di ampliare il set di istruzioni disponibili. Attraverso successive e analoghe implementazioni è possibile ottenere "set di istruzioni" via via più complessi.

I metodi per aggiungere un livello nuovo alla macchina preesistente, detta sistema ospite, sono tre (Den2 71):

. estensione
. traduzione
. interpretazione

Mediante l'estensione, si aggiunge al sistema ospite un software che è una collezione di procedure, che implementano le operazioni primitive del nuovo livello tramite le primitive del sistema ospite. Il linguaggio risultante comprende sia le primitive dell'ospite, sia quelle nuove.

Nel caso della traduzione il nuovo livello è completamente diverso da quello vecchio. E' necessario un compilatore per il sistema ospite, che traduca i programmi scritti nel nuovo linguaggio in programmi scritti in quello vecchio. Questi programmi scritti nel nuovo livello linguistico non sono direttamente eseguibili.

Infine, nel caso dell'interpretazione, esiste un interprete che permette di passare dal nuovo livello al livello dell'ospite. Programmi scritti nel nuovo livello linguistico sono direttamente eseguibili. La differenza tra il metodo dell'estensione e quello dell'interpretazione è che nel primo caso tutte le istruzioni del vecchio livello vengono accettate, mentre nel secondo caso questo non è detto.

2.2. Sistemi modulari

Molto spesso la modularità è una proprietà attribuita ad un programma. E' invece conveniente considerarla non come una proprietà del programma che si vuole eseguire, bensì come una proprietà posseduta o meno dal sistema di elaborazione (Den4 71, p. 129).

Diciamo che un sistema è modulare se il livello linguistico da esso definito gode delle seguenti proprietà (Den4 71, p. 130):

1) gli è associata una classe di oggetti che sono l'unità di rappresentazione del programma e che sono detti moduli di programma (i moduli sono quindi i "mattoni", coi quali si costruiscono i programmi più complessi);
2) fornisce il mezzo per collegare moduli di programma in modo da realizzarne di più complessi senza modifiche ai singoli moduli. E' molto importante mettere in rilievo che questi devono sempre avere un significato indipendente dal contesto in cui vengono usati.

Un sistema modulare presenta notevoli vantaggi rispetto a uno non modulare. Infatti permette di realizzare i seguenti due scopi (Den2 71, p. 129) :

a) esecuzione del "debugging" su ogni modulo separatamente;
b) connessione di moduli di programma scritti da persone diverse per formare un unico programma, conoscendo solo le interfacce e non il funzionamento interno dei singoli moduli.

Oltre che facilitare il debugging, l'organizzazione in moduli indipendenti e con interfacce definite limita i problemi derivanti dalla propagazione degli errori e permette di localizzare i "side effects".

2.3. Determinismo

Quando, come nel caso dei sistemi operativi, il sistema software produce un ambiente di processi asincroni cooperanti in evoluzione, un problema essenziale è quello del determinismo di tale sistema (par. 1.1.).

Se il sistema è costruito seguendo la definizione di modularità (par. 2.2.) esistono condizioni perchè il sistema stesso sia deterministico in senso lato.

Il teorema di Patil afferma (Den1 71): un sistema S formato da un insieme di sottosistemi S_i è funzionale se lo sono tutti gli S_i e se questi sono connessi come specificato nel punto 2.3.1. seguente. Quindi la classe dei sistemi funzionali è chiusa rispetto all'operazione di unione.

2.3.1. Connessione richiesta:

S_i e S_j siano due sistemi collegati in modo tale che un "outlet" p di S_i sia associato a un "inlet" q di S_j (Fig. 2.3.1.1.).

Fig. 2.3.1.1.

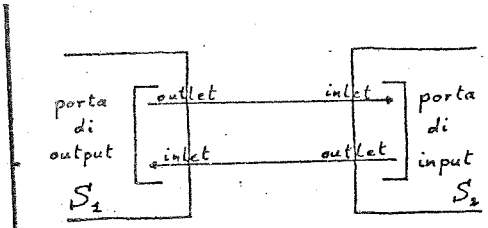

Fig. 2.3.2.1.

Per soddisfare il teorema di Patil è necessario che i messaggi arrivino nello stesso ordine in cui sono inviati e che non ne vada perso alcuno. Ci sono due metodi per soddisfare queste condizioni:

1) introdurre una coda, anche infinita, di tipo FIFO tra S_i e S_j, destinata a contenere i messaggi in attesa di essere ricevuti;
2) impedire al sistema S_i di emettere un segnale prima che S_j sia pronto a riceverlo, cioè prima che S_j abbia assorbito quello precedente.

Questo secondo metodo è definito tramite la condizione di Dennis :

Condizione α: per ogni associazione di un "outlet" p di un certo S_i a un "inlet" q di un qualunque S_j, il sistema totale S deve contenere un cammino dall'"inlet" q all'"outlet" p tramite i sottosistemi di S o anche tramite l'ambiente esterno, tale che ogni segnale emesso dall'"outlet" p richieda il precedente assorbimento di un segnale dall'"inlet" q.

Qualunque dei due metodi si usi per l'interconnessione, un sistema che sia una collezione di sottosistemi funzionali è pur esso funzionale.

2.3.2. Sistemi β

La condizione α è sempre verificata nei sistemi di tipo β, i quali godono delle seguenti caratteristiche (Den3 71):

1) ogni sottosistema contiene almeno una porta di "input" e una di "output", ognuna con un "inlet" e un "outlet", cosicchè due sottosistemi vengono collegati nel modo indicato in Fig. 2.3.2.1. .
2) questi sosttosistemi obbediscono alla regola di non mandare un messaggio dal loro "outlet" se prima non hanno ricevuto un segnale nel loro " inlet " associato. Questo segnale viene mandato dal sistema ricevente solo quando esso è pronto a elaborare nuovi dati.

Verificando in questo modo la condizione α, un'interconnessione di sistemi β funzionali è anch'essa funzionale. Essendo anche una coda di tipo FIFO un sistema β funzionale, sistemi β funzionali interconnessi tramite code FIFO danno origine a un sistema β funzionale, più complesso.

2.4. Strutture dati

Oltre a un livello linguistico adeguato la modularità richiede anche particolari strutturazioni dei dati. Tre sono le caratteristiche delle quali deve godere un livello linguistico per garantire la modularità (Den4 71, p. 140):

1) qualunque struttura dati deve poter diventare componente di un'altra struttura dati;
2) qualunque struttura dati deve poter essere passata (tramite riferimento) a/da un modulo di programma come parametro effettivo;
3) un modulo di programma deve poter costruire strutture dati di complessità arbitraria.

Ci sono tre modi per realizzare un livello linguistico adatto alla modularità, partendo da un livello ospite H (Den4 71, p. 141) :

1) si può usare un linguaggio di programmazione L standard, avente un traduttore e una classe di strutture dati e operazioni primitive adeguate;
2) si può estendere il linguaggio L' che non offre una classe adeguata di strutture dati ad uno L che la offre;
3) si può introdurre un nuovo linguaggio L costruendo :
   a - un traduttore da L ad H

b - un interprete di L funzionante a livello H.

In tutti e tre questi casi le strutture dati di L vengono memorizzate nella memoria lineare di H, in modo tale che le primitive di L vengano implementate tramite le primitive di H. Nel caso 1) L è un linguaggio standard e il passaggio da L ad H è uniforme rispetto a tutti i moduli di programma. Invece nei casi 2) e 3) la memorizzazione delle strutture dati di L in H viene realizzata indipendentemente dai progettisti dei singoli moduli, per cui generalmente è diversa da modulo a modulo. Di conseguenza sorgono difficoltà nella connessione dei moduli.

Se due moduli A e B sono realizzati con livelli linguistici diversi, B non potrà utilizzare direttamente una struttura dati generata da A (e viceversa). Si possono usare entrambi i moduli A e B all'interno dello stesso programma (Den4 71 , P. 143),

1) se tra di essi non deve avvenire alcuno scambio di dati o,
2) se esistono due routine t, t^{-1} funzionanti al livello H, che fanno passare dalla rappresentazione della struttura dati di A in H a quella di B in H e viceversa.

In Fig. 2.4.1. :

S_A = struttura dati di A al livello L

S_B = struttura dati di B al livello L'

S_H = trasformazione delle strutture dati di A e B al livello H

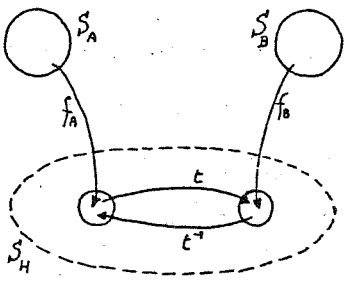

Fig. 2.4.1.

3. MODULARITA' STATICA E/O DINAMICA

Analizzeremo adesso in quale modo si possano collegare moduli software tra di loro, in modo di permettere la modularità in un ambiente di processi asincroni cooperanti.

Riferendosi alla Fig. 2.4.1. che rappresenta la situazione di un sistema modulare, si possono distinguere due casi: se i collegamenti alle routine t e t^{-1} (e quindi la topologia del sistema) vengono fissati in fase di definizione del sistema e non sono modificabili durante l'evoluzione dei processi, abbiamo una situazione di modularità statica; se invece i collegamenti a t e t^{-1} vengono definiti durante l'evoluzione dei processi quando questi le richiamano per collegarsi ad altri processi, si parla di modularità dinamica. Di seguito si chiariscono gli aspetti più importanti di questi due tipi di modularità.

Un qualsiasi programma, una volta codificato, passa attraverso le fasi di:

. compilazione
. caricamento
. esecuzione.

In fase di compilazione il programma, scritto in un linguaggio ad alto livello o in assembler, viene tradotto in un linguaggio intermedio che ad esempio è il linguaggio di input del "linkage editor". Una volta terminata correttamente la fase di compilazione, si ha generalmente una fase in cui intervengono il "linker" e il caricatore ("loader"); mentre il linker ha il compito di stabilire i collegamenti tra i moduli e produce un codice binario quasi sempre rilocabile, il caricatore serve per caricare in memoria centrale il programma.

Se abbiamo il sistema con modularità statica le fasi di "linking" e di caricamento sono contemporanee e operano nella fase di generazione del sistema in cui sono creati i processi e interconnessi in modo definitivo tutti quanti i moduli, che verranno richiamati nel corso dell'esecuzione del sistema. Da questa fase si passa alla fase di esecuzione. Se invece abbiamo un sistema con modularità di tipo dinamico, la fase di linking - caricamento e quella di esecuzione non sono più nettamente distinte e sequenziali; esse si alternano invece in continuazione, in base alle richieste emesse dal programma. La fase di esecuzione verrà cioè interrotta per tornare a quella di linking-caricamento, tutte le volte che un modulo richiederà di interagire con un altro modulo.

4. COLLEGAMENTO TRA MODULI IN UN AMBIENTE DI MODULARITA' STATICA

Ci sono essenzialmente due possibilità di realizzare tale collegamento. La prima utilizza il concetto di porte di I/O, l'altra le strutture di interfaccia, e sono dette rispettivamente "port approach" e "interface approach".

4.1. Port approach

Per porta si intende un punto di ingresso/uscita attraverso il quale fluiscono i messaggi che vengono scambiati tra i moduli (Anc2 73). Ogni porta di uscita è connessa ad una porta di ingresso mediante un collegamento unidirezionale. A seconda dell'hardware preesistente ci saranno poi piccole variazioni nei dettagli del meccanismo, in modo di adattarlo il più possibile alla configurazione fisica dell'elaboratore. Caratteristica del collegamento tramite porte è che i moduli ignorano con quale altro modulo siano collegati, ma conoscono soltanto il nome, ad essi locale, della loro porta, a cui i moduli si riferiscono per le operazioni di scambio messaggi. E' evidente che, ignorando i moduli con chi avvenga il collegamento, sono verificate le condizioni per la modularità del sistema cui danno origine.

Con un'organizzazione a porte come quella fin qui descritta, nella quale cioè le porte sono

collegate direttamente le une alle altre, i moduli sono strettamente legati alle relative velocità; essi non possono mandare (vedi condizione ω, par. 2.3.1.) un nuovo messaggio a un altro modulo, se quello inviato precedentemente non è ancora stato ricevuto.

Allora per rendere i moduli più indipendenti dalle relative velocità, si può inserire nei collegamenti tra porte una memoria tampone in modo tale che possa contenere diversi messaggi contemporaneamente (Fig. 4.1.1.). Questa memoria tampone è detta "mailbox" e contiene un certo numero di "slot"; ciascun "slot" contiene un messaggio. L'indipendenza dalle velocità relative è totale e il sistema è completamente asincrono nel caso limite, in cui il numero di "slot" è infinito.

Una "mailbox" è rappresentata da un semaforo binario e da una coda. Il meccanismo di scambio messaggi tra moduli utilizza due sole primitive del tipo "send m,p" e "receive p".

send m,p: m è il nome del messaggio e p è la porta, tramite la quale m viene inserito nella mailbox. Se questa è già piena, il processo viene bloccato, finchè non vi si libera spazio sufficiente per inserirvi il messaggio.

receive p: p è il nome della porta, attraverso la quale il messaggio viene inviato al modulo ricevente. Se la mailbox non è vuota, il modulo che ha richiesto l'esecuzione della receive riceve uno dei messaggi contenuti nella mailbox, scelto secondo la politica di scheduling seguita (priorità, coda FIFO, etc.). Se invece la mailbox dovesse risultare vuota, il modulo ricevente viene bloccato in attesa che un qualche modulo invii un messaggio alla mailbox.

Le primitive send e receive con questa particolare definizione semantica, permettono l'eliminazione dell'uso di nomi globali di variabili all'interno dei moduli.

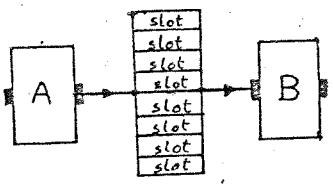

Fig. 4.1.1.

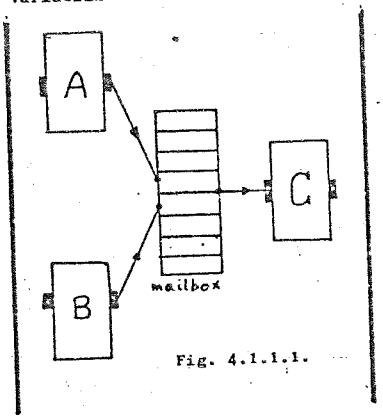

Fig. 4.1.1.1.

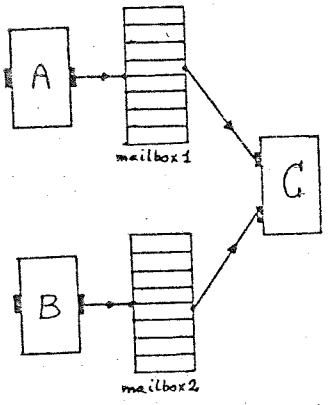

Fig. 4.1.1.2.

4.1.1. Collegamento generalizzato di più moduli

Abbiamo detto che attraverso una mailbox possono essere collegati più moduli, ma questo implica una particolare situazione. Se abbiamo una situazione come in Fig. 4.1.1.1., A, B e C sanno soltanto che, tramite una loro porta, sono collegati a una mailbox, ma ignorano chi altri vi sia collegato. Se A e B inviano, tramite le loro porte, dati alla mailbox, C non è in grado di distinguere l'origine di tali dati. Questa proprietà che caratterizza il "port approach" è detta indistinguibilità della provenienza dei messaggi. Se invece si vogliono separare completamente i dati a seconda della loro provenienza, il modulo C dovrà avere due porte collegate a due mailbox distinte, a loro volta collegate una alla porta di A e una alla porta di B (Fig. 4.1.1.2.).

Si noti che un sistema del tipo di quello in Fig. 4.1.1.1. non garantisce la funzionalità del sistema.

Una situazione analoga si ha quando C è il produttore e A e B i consumatori. Due esempi di sistemi che seguono il port approach sono reperibili in (Bal 71) e (Anc2 73).

4.2. Interface approach

4.2.1. Generalità

Consideriamo il secondo meccanismo possibile per realizzare la comunicazione tra moduli: quello che si basa sulle strutture di interfaccia. Con questo tipo di organizzazione i messaggi che sono output di un modulo e input di un altro, e che nel caso visto del port approach vengono memorizzati temporaneamente in una mailbox, ora non vengono più trattati tutti indistintamente nello stesso modo. Il tipo di struttura in cui vengono depositati non è più unico (cioè la mailbox), ma ne esistono varie classi.

Inoltre, mentre nel caso delle porte il collegamento tra moduli è realizzato tramite connessioni unidirezionali, in questo caso la connessione è ottenuta fornendo ai moduli il diritto di accesso alle interfacce mediante primitive di gestione delle varie strutture dati. E' possibile progettare le strutture di interfaccia prima che i moduli siano completati in tutti i loro particolari. In questo modo esse rappresentano una cornice che semplifica la fase di progetto dei moduli, ma che spesso deve in seguito venir modificata a causa di caratteristiche non previste dei moduli.

I moduli possono avere accesso fisico alle strutture di interfaccia. In questo caso devono conoscere la locazione della struttura e il suo formato, detto "layout"; questo però crea due problemi:

1) se si modifica l'interfaccia, bisogna modificare anche i moduli che vi si riferiscono;

2) se nella fase di "debugging" un modulo modifica una struttura di interfaccia a causa di un errore di programmazione, questo fatto può venir notato solo in seguito, e cioè quando un altro modulo si riferisce alla stessa interfaccia. Trovare quindi la causa dell'errore è molto difficile.

Per risolvere questi problemi si può organizzare il sistema in modo tale che i moduli abbiano accesso logico alle strutture, anzichè un accesso di tipo fisico. Esisterà cioè un sistema intermediario gestore dell'interfaccia (detto "interface-handling system", e che corrisponde al meccanismo richiesto da Dennis per lo scambio dei messaggi), il quale farà corrispondere gli indirizzi fisici ai nomi simbolici, tramite i quali i moduli si riferiscono alle strutture.

Dato che i moduli hanno accesso diretto alle strutture (logico o fisico), sorge il problema della protezione. Per risolverlo e impedire ai moduli di accedere e manipolare le strutture in modo non consentito, sarà necessario fornire al sistema gestore dell'interfaccia, in corrispondenza di ogni struttura, il nome di tutti i moduli aventi il permesso di utilizzarla, specificandone in particolare i diritti di accesso (Lam 69).

4.2.2. Esempio di sistema con struttura di interfaccia

Allo scopo di illustrare in dettaglio certi aspetti di un sistema basato sulle strutture di interfaccia, analizziamo il caso presentato da Weissman e Stacey (Weis 73). In questo sistema vengono considerati quattro tipi di strutture di interfaccia: blocchi di controllo, tabelle, stack e code, che si differenziano nel seguente modo (Weis 73).

I blocchi di controllo sono le strutture più generiche, in quanto sono formati da diversi campi che non sono tenuti a godere di particolari proprietà.

Le tabelle contengono una serie di ingressi individuabili tramite il primo campo (indice).

Gli stack sono formati da una serie di ingressi, ai quali si accede secondo la regola LIFO (last in, first out).

Infine le code sono una serie di ingressi, ai quali si accede secondo la regola FIFO (first in, first out).

Di una data struttura possono esistere contemporaneamente varie istanze, "occorrenze", individuabili tramite un indice. Ad ogni istante ci sarà una particolare occorrenza considerata corrente, sulla quale verranno eseguite le operazioni. Le istanze di una certa struttura possono a loro volta venir raggruppate a formare sottogruppi. Ad ogni istante ogni sottogruppo conterrà una istanza considerata corrente.

Sono ammessi tre tipi di accesso: read, write, control.

read permette solo la lettura;

write permette sia la lettura, sia la scrittura;

control permette a un modulo di aggiungere un'istanza a una struttura di interfaccia o di togliervela.

4.2.2.1. Meccanismo di gestione dell'interfaccia

Agisce in due fasi: definizione ed esecuzione.

1) Nella fase di definizione, in base agli ingressi ricevuti riguardo i moduli e i loro diritti di accesso, il sistema costruisce delle tabelle che memorizza su disco.
2) Nella fase di esecuzione le trasferisce nella memoria centrale e fa partire l'esecuzione del particolare sistema software cui si riferisce.

Vediamo ora come si svolge la fase di definizione. Il suo ingresso è a sua volta suddiviso in due parti. Nella prima si definiscono i nomi delle strutture di interfaccia (ad uso dell'utente), la classe a cui esse appartengono e i moduli che possono accedervi, con i relativi diritti di accesso. Come già detto precedentemente questi sono READ, WRITE e CONTROL, mentre le classi sono CBLOCK(blocco di controllo), TABLE (tabella), STACK (pila) e QUEUE (coda). Nella seconda parte si definisce invece il formato di ogni struttura.

Le operazioni che i moduli possono eseguire sulle strutture di interfaccia e che sono gestite nella fase di esecuzione sono 12 e per la maggior parte specifiche di una particolare classe di strutture. Sono invece comuni a tutte le classi le operazioni ALLOCATE, FREE, MAKECURR.

ALLOCATE: crea un istanza di una struttura e permette l'inizializzazione nel caso di tabelle e blocchi di controllo;

FREE : cancella un'istanza di una struttura;

MAKECURR: rende un'istanza corrente .

Invece nel caso dei blocchi di controllo abbiamo in particolare: GET che passa l'informazione dal blocco di controllo al modulo, e la sua duale PUT.

Nel caso di tabelle abbiamo: FIND che individua un ingresso, ne estrae il dato, e/o cancella l'ingresso; INSERT , che aggiunge o sostituisce una voce nella tabella; GETNO , che stabilisce il numero di ingressi in una tabella.

Nel caso di stack abbiamo: PUSH , che aggiunge un elemento in cima allo stack e la sua duale POP.

Nel caso di code abbiamo: ADD , che aggiunge un elemento in fondo alla coda e la sua duale REMOVE.

5. CONFRONTO TRA PORTE E STRUTTURE DI INTERFACCIA

In questo capitolo faremo un confronto tra i due approcci rispetto ad alcuni dei molti problemi che essi devono risolvere. Il confronto è eseguito riferendosi a tre aspetti distinti:

. generalità dei messaggi

. concorrenza
. collegamento statico e/o dinamico dei moduli

5.1. Generalità dei messaggi

5.1.1. Considerazioni relative al meccanismo

La differenza tra i due tipi di collegamento è la seguente:
Nel caso delle strutture i moduli accedono direttamente ad esse, fisicamente o logicamente (utilizzando il sistema gestore dell'interfaccia). Dato che le strutture sono di vari tipi e che a ogni tipo corrisponde un certo numero di primitive, le strutture influenzano direttamente i moduli, in quanto li costringono a utilizzare una primitiva piuttosto che un'altra. Le strutture sono oggetti attivi, in quanto tanto le code di tipo FIFO o LIFO, tanto tabelle e blocchi di controllo sono implicati esplicitamente nello scambio di messaggi.
Nel caso dell'organizzazione basata sulle porte è previsto un unico tipo di struttura per lo scambio dei messaggi: la mailbox; i moduli non possono mai accedervi direttamente, in quanto conoscono solo il nome locale delle loro porte, mentre ignorano la mailbox e quindi a quale altro modulo sono collegati. Per questo motivo le mailbox sono da considerarsi degli oggetti passivi. Esistendo un unico tipo di messaggi, sono sufficienti due sole primitive per accedervi: una send per inviarli e una receive per riceverli.

5.1.2. Protezione

La protezione si ottiene verificando il diritto di accesso ai messaggi dei moduli/processi.
Per realizzare il controllo nel caso dell' interface approach, sarà necessario associare ad ogni struttura il nome dei moduli che vi possono accedere e specificare per ognuno di essi i diritti di accesso posseduti (Lam 69). In questo modo, se ad esempio il modulo A vuole eseguire una ADD su una coda C, per prima cosa il meccanismo verificherà se nel descrittore della coda è specificato che A possiede il diritto di aggiungere elementi a C; in caso affermativo la ADD verrà eseguita.
Nel caso delle porte, invece, la protezione è implicita nel meccanismo; infatti non conoscendo i moduli la locazione o il nome delle mailbox, è soltanto il meccanismo che gestisce le mailbox d'accordo con le definizioni specificate in fase di definizione del sistema. Infine, l'elaborazione dei dati da parte di un modulo avviene nel suo spazio dei nomi, al quale nessun altro modulo ha diritto di accesso.

5.1.3. Analogia tra i due meccanismi

Confrontando port approach e interface approach, si vede che il primo è caratterizzato dall'esistenza di mailbox e di un meccanismo centralizzato (meccanismo di gestione delle comunicazioni), mentre il secondo è caratterizzato dall'esistenza di una molteplicità di strutture e da un meccanismo distribuito (in quanto le strutture possono essere di vari tipi) ad esse associate.
Analizzando uno alla volta i vari tipi di strutture, si osserva che code FIFO e mailbox rappresentano la stessa struttura. Anche per ciò che riguarda gli stack non c'è differenza fondamentale con le mailbox, in quanto basta assegnare una priorità ai messaggi che vengono posti nella mailbox, per poterli estrarre in ordine diverso da quello seguito nell'inserimento (ad esempio LIFO). Dunque un interface approach in cui si utilizzano solo code FIFO e stack può essere ricondotto al port approach. Per quanto riguarda le tabelle, sorgono invece dei problemi derivanti dai seguenti due fatti:

1) oltre al caso in cui i moduli leggono un elemento della tabella e lo cancellano, esiste anche quello in cui i moduli leggono o cancellano solamente. Siamo cioè in una situazione, in cui per alcuni elementi della tabella esiste un'interpretazione diretta mentre per altre no, senza che sia possibile distinguere a priori tra l'una e l'altra situazione.
2) generalità dell'ordine di accesso agli elementi della tabella: nel caso delle strutture i moduli determinano l'ingresso desiderato tramite un campo indice, che implica la conoscenza della struttura interna della tabella: invece nel caso delle porte è impossibile che un modulo acceda all'ingresso desiderato, in quanto non è in grado di sapere dove questo si trovi o come individuarlo.

Problemi del tutto analoghi sorgono nel caso dei blocchi di controllo, per cui si può concludere che, se nell'interface approach si fa riferimento a tabelle e blocchi di controllo, il sistema non può essere ricondotto direttamente al port approach.

5.1.4. Conclusioni: vantaggi e svantaggi

Si nota che un collegamento basato sulle porte è più orientato verso il messaggio che non uno basato sulle strutture di interfaccia. Il primo è infatti basato su messaggi generalizzati, mentre il secondo sulla semantica del dato. Nell'interface approach, infatti, si sceglie il tipo di struttura più adatto al particolare messaggio da mandare; questo è senza dubbio un vantaggio, ma implica la necessaria esistenza di un grande numero di primitive, più o meno complicate, necessarie per poter agire in modo corretto sulle quattro classi di strutture.

Un altro svantaggio dell'interface approach è che i moduli sono molto più specializzati che non nel caso del port approach, per cui la libreria del sistema aumenta. Oltre alle primitive caratteristiche dei vari tipi di struttura ne dovrà esistere anche un certo numero atto a trasformare primitive caratteristiche di una certa classe in primitive caratteristiche di un'altra. Tutto questo comporta un "overhead" non indifferente.

Un altro grosso svantaggio delle strutture rispetto alle porte è il seguente: dal punto di vista della funzionalità il teorema di Dennis (cfr. teorema di Patil par.2.3)afferma che moduli funzionali collegati tramite code FIFO generano sistemi anch'essi funzionali, ma non dice nulla riguardo gli altri tipi di strutture. Il problema della funzionalità è quindi molto più complesso nel caso delle strutture che non nel caso delle porte.

Si può ancora obiettare che un sistema di comunicazione basato sulle porte non permette che un messaggio sia destinato a più moduli. Sarà quindi necessario inviarne una copia a ogni modulo. Questo "overhead" è però giustificato, in quanto permette la modularità.

5.2. Concorrenza

L'ultimo punto su cui si basa il nostro confronto tra interfacce e porte è la concorrenza dei processi. Si vede subito che essa è possibile in entrambi i casi.

Infatti, considerando il collegamento con le porte, abbiamo visto che i moduli possono procedere in parallelo, finchè continuano a ricevere dati di ingresso e finchè le mailbox, nelle quali inviano i risultati, non sono piene. Per quanto riguarda il caso delle strutture, l'esempio portato da Weissman e Stacey è costituito da un sistema, nel quale l'esecuzione dei moduli è possibile solo in modo sequenziale, ma è molto semplice modificare il sistema per permettere ai moduli di procedere in parallelo. Questa modifica consiste nell'introduzione di semafori di mutua esclusione su ogni struttura, in modo da impedire a due o più moduli di accedere contemporaneamente alla stessa struttura di interfaccia. Dunque anche dal punto di vista della concorrenza porte e strutture si comportano allo stesso modo.

5.3. Collegamento statico e/o dinamico

L'organizzazione con le porte, studiata in questo lavoro, realizza una modularità statica, nella quale cioè la creazione di tutti i processi e la interconnessione tra i moduli avvengono in fase di generazione del sistema. Una modularità di tipo statico non è una limitazione quando, come nel nostro caso, ci si limiti a considerare sistemi di tipo "Real Time", orientati all'evento. In questi sistemi tutti i processi che verranno eseguiti sono creati all'atto della generazione del sistema; è quindi ammissibile, che in questa fase vengano definiti tutti i collegamenti tra i vari moduli, e che i moduli - processi vengano posti inizialmente nella coda dei processi bloccati per essere trasferiti in quella dei processi pronti ("ready"), non appena arrivino i dati stabiliti in base ai collegamenti prefissati.

Nel sistema che implementa le struttura di interfaccia (Weis 73), le corrispondenze tra i nomi delle strutture simboliche e i campi possono essere fissate in fase di esecuzione, ottenendo una gestione dinamica della memoria. Questa caratteristica però non è da confondersi con la modularità dinamica; infatti la modularità è statica e arriva al livello del compilatore e quindi a un livello inferiore a quello del port approach. Le tre primitive che mettono in evidenza la dinamicità della gestione nel caso delle strutture di interfaccia sono ALLOCATE, FREE e MAKECURR relative alle istanze delle strutture. Siamo quindi in presenza di un tentativo molto rudimentale e parziale di collegamento dinamico dei moduli. Questo crea un "overhead" di esecuzione piuttosto sensibile, ma d'altro lato permette una maggiore flessibilità.

L'unico esempio noto a chi scrive, in cui si affronta completamente il problema della modularità dinamica è quello dovuto ad Alan Kay (per alcune notizie generali si veda Kay 75).

6. CONCLUSIONE

E' stata introdotta la definizione di uno schema generale di un sistema software modulare. Si sono quindi esaminati in dettaglio due dei possibili meccanismi di comunicazione tra moduli, cioè il "port approach" e l'"interface approach", per verificarne l'applicabilità rispetto ad alcuni punti fondamentali per la modularità.

Si è potuto così stabilire, che entrambi i meccanismi permettono la costruzione di sistemi modulari e un collegamento tra i moduli sia in fase di compilazione, sia in fase di caricamento; la differenza sostanziale consiste nel fatto, che il "port approach" è tipicamente orientato alla trasmissione dei messaggi tramite connessione unidirezionale tra porte di I/O e "buffer" ("mailbox"), mentre l'"interface approach" produce un approccio più tradizionale, basato sull'organizzazione generalizzata di una struttura di dati di interfaccia.

In ogni caso è interessante notare come restringendo tutte le possibili strutture dell'"interface approach" a code FIFO e "stack", i due meccanismi possono essere considerati equivalenti, a parte la maggiore generalità del "port approach".

Possiamo concludere che le caratteristiche del "port approach" sono le più generali, anche se questo va a discapito della facilità di implementazione; le caratteristiche dell'"interface approach" ammettono invece ampia flessibilità nell'implementazione, ma questa si ripercuote sulla possibilità di garantire la funzionalità e la modularità dei sistemi risultanti.

BIBLIOGRAFIA

(Akk 72) : Akkoyunlu, Bernstein, Schantz: An Operating System for a Network Environment ; Proceedings of the Symposium on Computer Communication Networks and Teletraffic; Vol.XXII of the MRI Symposia Series; Politechnic Institute of Brooklyn, April 4-6, 1972; pp. 529-538

(Akk 73) : Akkoyunlu, Bernstein, Schantz: Software Communication across Machine Boundaries; Proceedings of COMPCON; Boston, USA; September 1973; pp. 203-205

(Anc1 73) : Ancilotti, Cavina, Fusani, Gramaglia, Ljitmaer, Martinelli, Thanos: Designing a Software Laboratory; Rivista di Informatica, 1973 e Proceedings of the 8th Yugoslav International Symposium, Bled 1973; pp. a3, 1-7

(Anc2 73) : Ancilotti, Ljitmaer: Un esempio di multiprogrammazione strutturata, P.S.L.: un laboratorio software; Rivista di Informatica, Vol. 4 N° 3/4/ Settembre-Dicembre 1973 pp.357-366

(Anc3 73) : Ancilotti, Boari, Ljitmaer: Tecniche di programmazione strutturata estese a un ambito di processi concorrenti; Rivista di Informatica, Vol. 4, N° 3/4 Settembre-Dicembre 1973; pp. 335-356

(Bal 71) : Balzer: Ports - A Method for Dynamic Interprogram Communication and Job Control; Proceedings of the Spring Joint Computer Conference. Vol. 38, 1971; pp. 485 - 489

(Den1 71) : Dennis, Patil: Computation Structures: Course Notes; Dept. Elct. Eng., MIT 1971

(Den2 71) : Dennis: The Design and Construction of Software Systems; Advanced Course on Software Engineering; Bauer editor, Springer Verlag, 1973; pp. 12-27

(Den3 71) : Dennis: Concurrency in Software Systems; Advanced Course on Software Engineering; Bauer editor, Springer Verlag 1973; pp. 111-127.

(Den4 71) : Dennis; Modularity; Advanced Course on Software Engineering; Bauer Editor; Springer Verlag, 1973; pp. 128-150.

(Han 72) : Brinch Hansen: A comparison of two Synchronizing Concepts; Acta Informatica 1, 1972; pp. 190-199

(Han1 73) : Brinch Hansen: Concurrent Programming Concepts; ACM Computing Surveys, Vol. 5; N° 4, December 1973; pp. 223-245

(Han2 73) : Brinch Hansen: Operating System Principles; Prentice Hall, Inc. Englewood Cliffs, N.J.

(Ho 74) : Hoare: Monitors: An Operating System Structuring Concept; Communications of the ACM ; October 1974, Vol. 17, N° 10

(Lam 69) : Lampson B.W.: Dynamic Protection Structures; Proc. AFIPS 1969, FJCC, Vol. 35, AFIPS Press, Montvale, N.J, pp. 27-38

(Kay 75) : Alan Kay: Personal Computing; Meeting on 20 years of Computer Science, IEI PISA, Giugno 1975

(Met 72) : Metcalfe: Strategies for Interprocess Communication in a Distributed Computing System; Symposium on Computer Communications and Teletraffic; Politechnic Institute of Brooklyn, April 1972, pp. 1-20

(Mon 73) : Montanari: Processi cooperanti; Rivista di Informatica Vol. 4 N° 3/4/ Settembre-Dicembre 1973 pp. 323-333

(Ran 73) : Horning, Randell: Process Structuring; Computing Surveys Vol. 5 N° 1 March 1973 pp.5-30

(Weck 73) : Wecker: A design for a Multiple Processor Operating Environment; Proceedings of COMPCON; Boston, USA, September 1973, pp. 143-146

(Weis 73) : Stacey, Weissman: An Interface System for Improving Reliability of Software Systems: IEEE Symposium on Software Systems' Reliability, 1973 pp. 136-140

(Wu 72) : Corwin, Wulf: SL 230 - A software Laboratory Intermediate Report; Carnegie Mellon University, Department of Computer Science, May 1972

(Wu 73) : Shaw, Wulf; Global Variables Considered Harmful; SIGPLAN Notices 8,2, February, 1973 pp. 28 - 34

(Wu 74) : Cohen, Corwin, Jones, Levin, Pierson, Pollack, Wulf: Hydra: The Kernel of a Multiprocessor Operating System; Communications of the ACM, Vol. 17, N° 6, June 1974; pp. 337-345

(Zel 74) : Zelkowitz: Structured Operating System Organization; Information Processing Letters, Vol. 3, N° 2 , November 1974 pp. 39 - 42