RELAZIONE PROGETTO MAILBOX JAVA

Stefano Cicero

Corso di Sistemi Concorrenti e Distribuiti
Corso di Laurea Magistrale in Ingegneria Informatica
Università di Pisa
A.A. 2011-2012

Sommario

1. INTRODUZIONE

Questa relazione ha lo scopo di spiegare nel dettaglio come si è deciso di sviluppare il progetto, e il funzionamento del codice.
Il progetto consiste in una mailbox che viene utilizzata da più thread mittenti e da un solo thread destinatario. La mailbox si interfaccia con i thread mittenti e destinatario con un array di porte sincronizzate, le quali utilizzano come meccanismo di sincronizzazione esclusivamente dei semafori FIFO, che a loro volta sfruttano i meccanismi di sincronizzazione offerti dal Java 4.0 (blocchi e/o metodi synchronized, metodi wait(), notify() e notifyall()).
Date le specifiche, il progetto è composto dai seguenti file, con estensione .java:

- cdata
- FairSemaphore
- SynchPort
- PortVector
- MailBox
- Mittente
- Ricevente
- Test

A ogni file corrisponde almeno una classe Java. Vediamo ora il contenuto di questi file.

2. CLASSI CDATA E FAIRSEMAPHORE

Nel file "cdata.java" è definita la classe generica "cdata" appunto, la quale fornisce una struttura dati del pacchetto, che verrà utilizzato per il trasporto delle informazioni tra mittente e destinatario attraverso la mailbox. Per questo motivo si farà un uso estensivo di oggetti di questa classe in tutto il progetto. Tale struttura dati prevede i seguenti attributi:

- "msg": campo di tipo T generico atto a contenere l'informazione vera e propria che il mittente intende inviare al destinatario
- "mit": campo di tipo stringa contenente il nome del mittente del messaggio
- "numporta": campo di tipo intero in cui verrà inserito, da parte della mailbox, il numero della porta dalla quale il messaggio e stato ricevuto

Nel file "FairSemaphore.java" è definita appunto la classe "FairSemaphore", nella quale è implementato il meccanismo semaforico che verrà utilizzato nella realizzazione delle porte sincronizzate e non solo. Il semaforo è di tipo generico. Il costruttore della classe "FairSemaphore" prende in ingresso il valore con cui inizializzare il contatore del semaforo. I thread, che invocando il metodo "P()" trovano la risorsa occupata (contatore minore o uguale a "0"), vengono messi in una lista i cui elementi (definiti nello stesso file dalla classe "Coppia") sono composti dall'id del thread messo in attesa e da un campo di tipo booleano che consente di sapere, quando qualche thread esegue il metodo "V()", quale thread deve essere risvegliato, rispettando la politica FIFO. I metodi "P()" e "V()" vengono eseguiti in mutua esclusione, grazie all'istruzione "synchronized" nell'intestazione di questi ultimi.

3. CLASSE SYNCHPORT

Nel file "SynchPort.java" troviamo la classe generica "SynchPort" che implementa una porta sincronizzata. Gli attributi presenti sono:

- "R": semaforo di sincronizzazione
- "mutex": semaforo di mutua esclusione per la porta
- "M[]": array di semafori di sincronizzazione
- "counter": contatore per tener conto del numero di mittenti connessi in un certo istante
- "nmitt": intero indicante il numero massimo di mittenti contemporanei supportato dalla porta
- "bufport": buffer di oggetti di classe "cdata" per la memorizzazione dei messaggi di dimensione pari a quella del numero massimo di mittenti supportati
- "primo" e "ultimo": due interi per usare "bufport" come un buffer circolare

I metodi implementati sono 3:

- "SynchPort": costruttore avente in ingresso il numero massimo di mittenti contemporanei. Si occupa sostanzialmente di inizializzare gli attributi sopra descritti
- "send()": metodo bloccante. Ha in ingresso il nome del mittente e un dato generico, e si occupa di costruire il pacchetto (oggetto cdata) con i dati ricevuti in ingresso e di inviarli alla porta. Tale metodo effettua anche un controllo per evitare che troppi mittenti si colleghino simultaneamente sulla stessa porta
- "receive()": metodo anch'esso bloccante, estrae un elemento (contenente mittente e messaggio) dal buffer circolare della porta e lo restituisce al chiamante

La disponibilità di "nmitt" posizioni nel buffer garantisce che ciascuno degli "nmitt" mittenti ha la certezza di trovare un elemento del buffer disponibile quando invoca il metodo "send()". Infatti, a causa della sincronizzazione implicita della "send()", ogni mittente non può inviare un nuovo dato prima che quello eventualmente inviato in precedenza non sia stato ricevuto. Inoltre, al fine di evitare interferenze tra le esecuzioni concorrenti del metodo "send()" , è necessario il semaforo "mutex" di mutua esclusione.

4. CLASSE PORTVECTOR

Nel file "PortVector.java" è implementata la classe "PortVector" la quale definisce un array di oggetti "SynchPort" (porte sincronizzate), che verrà utilizzato dalla mailbox per ricevere richieste di servizio da parte dei mittenti. Gli attributi di questa classe sono:

- "vporte": un vettore di oggetti di classe "SynchPort"
- "mutexCounters": un semaforo di mutua esclusione per l'utilizzo in maniera atomica dell'array di interi "counters[]"
- "PVectSem": un semaforo di sincronizzazione per i metodi "Send()" e "Receive()" di questa classe
- "counters[]": un array di interi per sapere in un dato istante quanti mittenti sono in attesa su ciascuna porta
- "mitxport": un intero indicante il numero massimo di mittenti contemporanei su ciascuna porta dell'array

- “dim”: un intero corrispondente alla dimensione dell’array di porte sincronizzate

I metodi previsti da questa classe sono 4:

- “PortVector()”: è il costruttore della classe, avente come ingresso la dimensione dell’array di porte sincronizzate e il numero massimo di mittenti contemporanei su ciascuna porta. Con tali dati si occupa di inizializzare gli attributi sopra descritti.
- “AllChecked()”: prende in ingresso un array di booleani e la dimensione di tale array. Restituisce “True” se tutti gli elementi dell’array valgono “True”, altrimenti restituisce “False”. Verrà utilizzato nel metodo “Receive()”
- “Send()”: prende in ingresso il nome del mittente, il messaggio e la porta scelta dal mittente stesso. Con tali dati provvede a richiamare il metodo “send()” sulla porta corrispondente. Si occupa inoltre di verificare che il numero della porta ricevuto sia valido e che tale porta non sia satura di richieste di servizio da parte di altri mittenti.
- “Receive()”: riceve in ingresso un vettore di interi e la dimensione di tale vettore. Gli elementi di questo vettore contengono i numeri delle porte dalle quali tale metodo dovrà ricevere i messaggi (se ce ne sono). Il funzionamento di questo metodo è illustrato nel diagramma di flusso di figura 1.

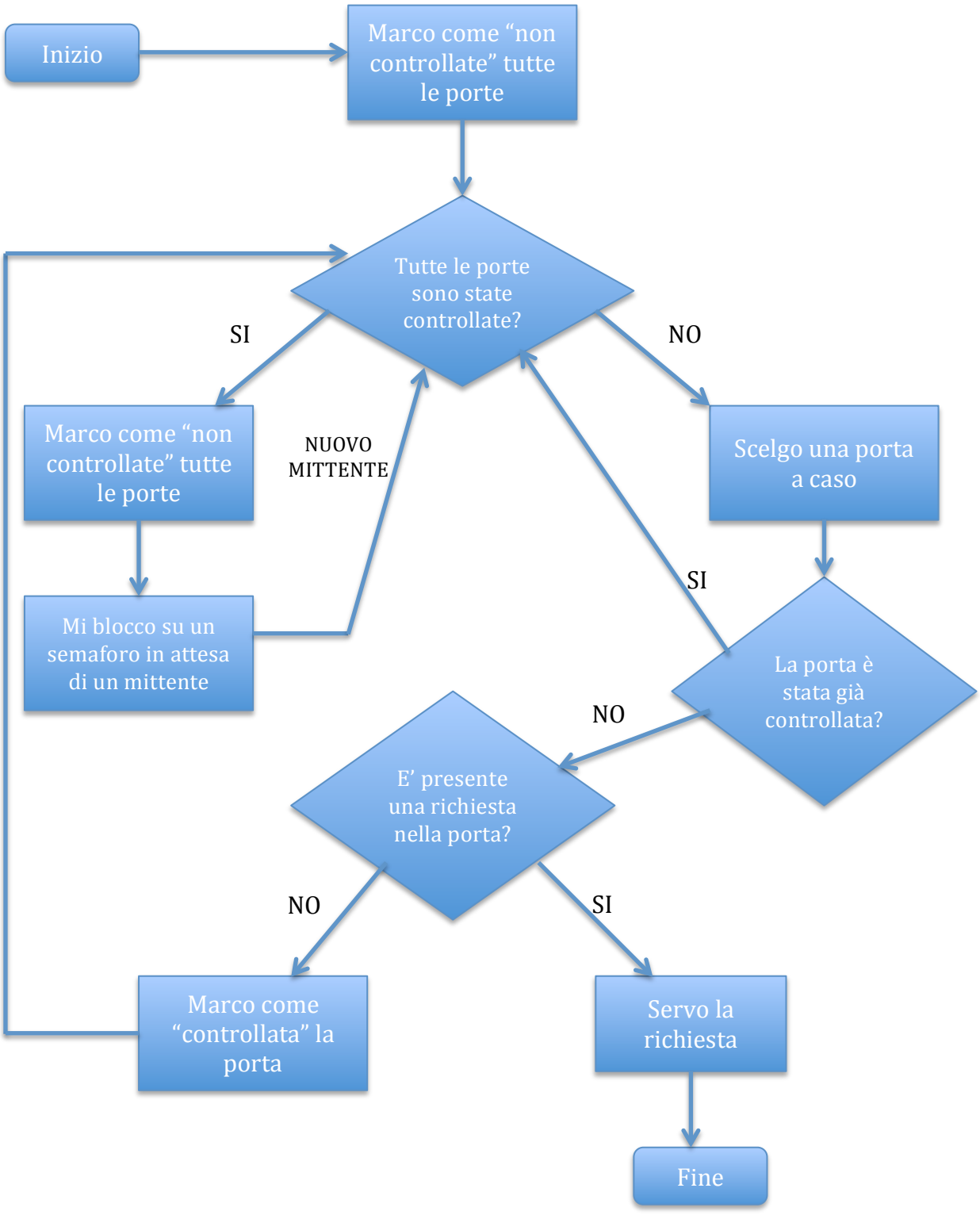
Inizio
Marco come "non controllate" tutte le porte
Tutte le porte sono state controllate?
SI
NO
Marco come "non controllate" tutte le porte
NUOVO MITTENTE
Scelgo una porta a caso
Mi blocco su un semaforo in attesa di un mittente
SI
La porta è stata già controllata?
NO
E' presente una richiesta nella porta?
NO
SI
Marco come "controllata" la porta
Servo la richiesta
Fine

Figura 1

5. CLASSE MAILBOX

Nel file “MailBox.java” viene implementata oltre alla classe “MailBox”, anche la classe generica “Buffer”. In quest’ultima classe sono presenti tutti gli attributi e i metodi per la realizzazione di un buffer circolare di n elementi di tipo “cdata”. Sono presenti quindi i metodi per l’estrazione (“get()”) e l’inserimento (“put()”) di elementi nel buffer in maniera atomica, grazie al semaforo “lockbuf”. Sono presenti inoltre altri due semafori “fullbuf” e “emptybuf”, per consentire la sincronizzazione nei thread, nei casi di buffer pieno e di buffer vuoto. Vi è poi il metodo “CheckFreeSpace()” che ritorna “True” se è presente almeno una locazione libera nel buffer, altrimenti ritorna “False”. L’ultimo metodo è dato da “waitForElem()” il quale si blocca nel caso il buffer sia vuoto.

La classe “MailBox” derivata dalla classe “Thread”, si occupa di inizializzare il vettore di porte sincronizzate denominato “io”, e di creare un oggetto della classe “Buffer” di dimensione 3 come richiesto dalle specifiche. Inoltre viene assegnato il tipo “integer” agli oggetti delle classi generiche “PortVector” e “Buffer”. Dalle specifiche risulta che la classe “MailBox” debba offrire due servizi:

- ricezione ed inserimento nel buffer (quando c’è spazio disponibile) dei valori inviati dai mittenti
- estrazione di un valore dal buffer e suo invio al processo ricevente

Ipotizzando che queste operazioni dovessero essere eseguite in contemporanea, si è deciso di “partizionarle” su due thread differenti. Per questo sono state definite altre due classi derivate dalla classe “Thread”, che si occupano di svolgere i servizi sopra descritti. Le classi in questione sono “TDown” che svolge il primo servizio, e “TUp” che svolge il secondo.

Gli attributi della classe “MailBox” sono dati da:

- mutlist: semaforo di mutua esclusione per la lista delle richieste pendenti in caso di buffer pieno
- auxlist: lista per la memorizzazione delle richieste pendenti in caso di buffer pieno
- io: oggetto di classe “PortVector” rappresentate un array di porte sincronizzate
- rangeport: array di interi contenente gli indici delle porte sincronizzate da utilizzare dalla mailbox e dai mittenti per comunicare
- buffo: oggetto di classe “Buffer” atto a memorizzare i messaggi inviati dai mittenti
- nporte: intero rappresentante il numero di porte sincronizzate nell’oggetto di classe “PortVector”
- nmitt: intero rappresentante il numero massimo mittenti contemporanei che la mailbox dovrà servire

I metodi della classe “MailBox” sono dati da:

- “MailBox()”: il costruttore della classe prende in ingresso un intero che viene interpretato come il numero di porte costituenti l’array di porte sincronizzate. Inoltre provvede all’inizializzazione degli attributi sopra descritti.
- “run()”: questo metodo (ereditato dalla classe “Thread”) provvede semplicemente a inizializzare e mandare in esecuzione un thread di classe “TDown” e un thread di classe “TUp”.

Ai thread “TUp()” e “TDown()”, la mailbox passa tutti i parametri necessari per espletare le loro funzioni, i quali vengono salvati nei corrispondenti attributi dai costruttori.

Il funzionamento del thread “TUp” è il seguente: ciclicamente, attende che il buffer non sia vuoto richiamando il metodo “waitForElem()” sull’oggetto buffer, dopo che viene segnalata la presenza di almeno un elemento, viene effettuata un’estrazione dal buffer e il dato inviato al destinatario, invocando il metodo “send()” sull’oggetto di classe “SynchPort” del destinatario. Infine controlla se ci sono elementi nella lista delle richieste pendenti dei mittenti, e se c’è spazio nel buffer (nel frattempo “TDown()” potrebbe aver riempito il buffer): in caso positivo tramite il metodo “fromListToBuff()” sposta un elemento dalla lista al buffer in base alla priorità.

Il thread “TDown”, ciclicamente, richiama il metodo “Receive()” sull’oggetto di classe “PortVector”, il quale restituisce un oggetto di classe “cdata”. Nel caso che il buffer non sia pieno e la lista dei mittenti sospesi sia vuota, l’oggetto viene inserito nel buffer; altrimenti l’oggetto viene inserito nella lista.

6. FUNZIONAMENTO GLOBALE

Nei restanti 3 files (Mittente.java, Ricevente.java, Test.java) vengono implementate le classi per i threads mittenti e ricevente, che effettueranno appunto l’invio e la ricezione dei messaggi. Questi threads saranno creati ed avviati all’interno della classe “Test”, avente metodo “main()”. Per spiegare come questi moduli sono stati implementati, occorre chiarire come tutto il sistema funzioni. Per farlo mi servirò del seguente schema logico:

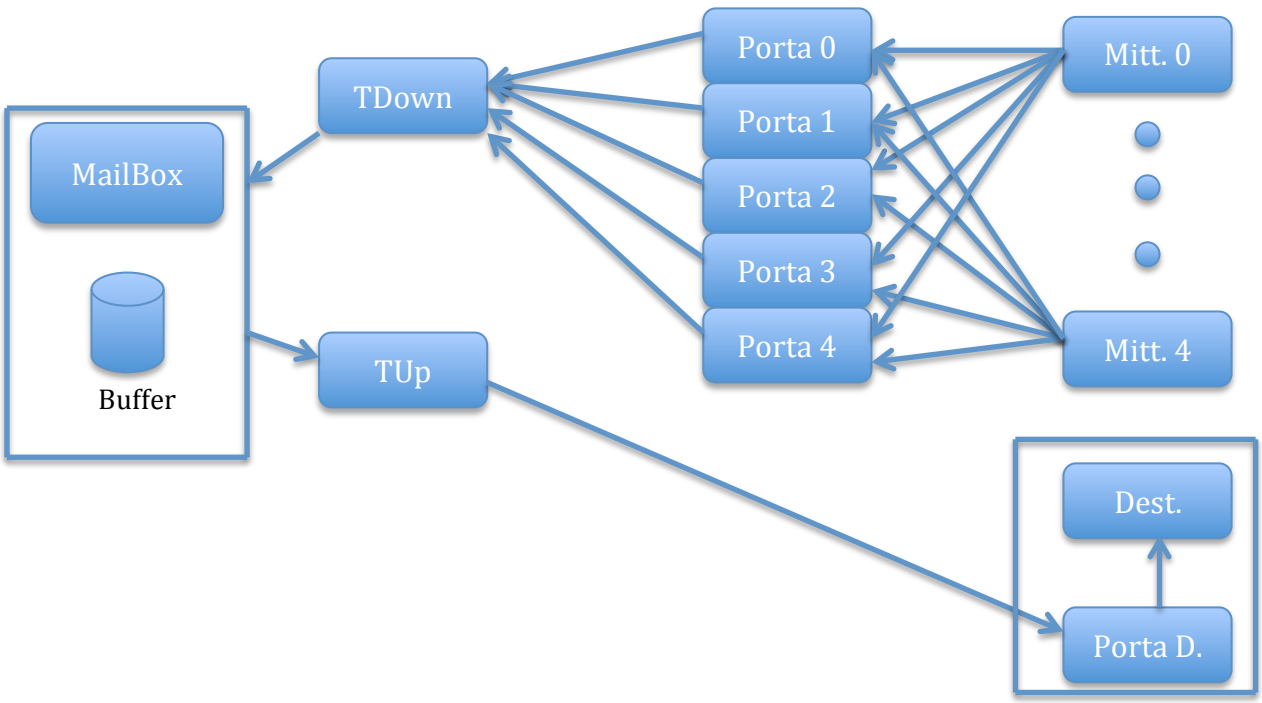

Figura 2

Come da specifiche la MailBox dovrà lavorare con 5 mittenti e un destinatario. Il tutto potrebbe funzionare con una singola porta sincronizzata (oggetto di classe "PortVector" avente array di oggetti di classe "SynchPort" di dimensione unitaria) in grado di supportare 5 mittenti simultaneamente. Nello specifico si è scelto di creare tante porte quante i mittenti (5), in modo da dare un maggiore grado di imprevedibilità nel comportamento dei threads mittenti. Questi ultimi utilizzeranno il metodo "Send()" sull'oggetto di classe "PortVector" dichiarato pubblico nella classe "MailBox". La mailbox di conseguenza utilizzerà il metodo "Receive()" sempre sullo stesso oggetto per ricevere i messaggi dai mittenti.

Per quanto concerne il destinatario, invece, il suo thread corrispondente avrà un oggetto pubblico di classe "SynchPort", tramite il quale riceverà i messaggi che la mailbox gli invierà nella porta medesima.

Come da specifiche, ciascun mittente deve inviare 4 messaggi, per un totale di 20 messaggi. Ogni mittente, perciò, effettuerà 4 iterazioni, in ciascuna delle quali sceglierà casualmente la porta sulla quale inviare il messaggio. Il destinatario quindi, effettuerà 20 iterazioni sulla propria porta sincronizzata di ricezione. Per facilitare il debug, ad ogni interazione viene inviato un messaggio pari al messaggio inviato nell'iterazione precedente, incrementato di 1. Nella prima interazione viene inviato il messaggio che viene passato come parametro al costruttore del thread mittente.

Nel file "Test.java" infine è implementata la classe "Test" avente un unico metodo "main()", che definisce il numero di porte. Subito dopo crea un oggetto di classe "MailBox", 5 threads di classe "Mittente" e uno di classe "Ricevente" e li manda in esecuzione. Per la compilazione, è sufficiente digitare da terminale il comando "javac Test.java", mentre per l'esecuzione si utilizza il comando "java Test". Data la considerevole quantità di righe di output (di solito oltre 400 righe), si consiglia di "direzionarlo" verso un file, piuttosto che stamparlo a monitor, con il comando "java Test > output.txt". Il programma non termina da solo in quanto, nonostante terminino i threads mittenti e ricevente, i threads "TUp" e "TDown" restano in esecuzione. Pertanto occorrerà terminare il programma con la combinazione di tasti "Ctrl+C".