SISTEMI OPERATIVI E PROGRAMMAZIONE DISTRIBUITA
PROGETTO PARTE JAVA

Daniele Giannetti

Questo documento illustra come si è deciso di realizzare il progetto e quali sono state le scelte che hanno condotto ad utilizzare alcuni meccanismi del linguaggio Java, inoltre si spiega il funzionamento del codice, nei minimi dettagli.

Il progetto era diviso in due parti, dunque le analizzeremo una ad una, alla fine di questo elaborato verrà illustrato l'utilizzo degli script ausiliari che sono stati inseriti per facilitare la esecuzione e compilazione dei file del progetto.

Tutto il codice che verrà mostrato nel seguito è preso direttamente dai file del progetto, ma è stato ripulito da una certa quantità di commenti.
Andando a vedere i file sorgente si potrebbero scorgere statement del tipo:

```
@SuppressWarnings("unchecked")
```

Questi servono solamente per evitare fastidiosi messaggi di warning in fase di compilazione, dovuti al fatto che java nella sua versione attuale non consente la definizione di array di oggetti generici. Per ovviare al problema si deve ricorrere ad array di oggetti non generici ai cui elementi si assegnano riferimenti di oggetti generici, questa operazione non è rischiosa in quanto tale, ma giustamente viene segnalata dal compilatore come operazione di casting non controllata (unchecked).

1) Implementazione del meccanismo di comunicazione sincrono

Veniva chiesto di implementare un meccanismo che emulasse la comunicazione sincrona tipica di un kernel a scambio di messaggi, mentre l'ambiente della Java Virtual Machine è per definizione a memoria comune.
In Java il concetto fondamentale per la concorrenza è infatti il Thread.
Veniva suggerito al progettista di implementare un link sincrono (canale da uno ad uno) oppure una porta sincrona (canale da molti ad uno), in entrambi i casi i canali dovevano essere generici rispetto al tipo di informazione scambiata.

Implementato un link o una porta, veniva poi chiesto di realizzare un array di link o di porte che permettesse di eseguire anche la receiveany() sull'intero array, la quale consente al chiamante di ricevere da una qualsiasi delle porte o link facenti parte dell'array, in modo non deterministico.

Tutto ciò è stato fatto ed incluso nel package SyncSupport, di cui spiegheremo adesso ogni componente.

Nel caso presente si è deciso di implementare porte (canali asimmetrici) e non link.

1.1) SyncSupport.SafeOutput.OutputQueue

Come si vede la classe OutputQueue fa parte di un sotto-package di SyncSupport, in particolare del package SyncSupport.SafeOutput.
Obiettivo di questa classe è quello di consentire ai programmi che useranno il package SyncSupport o anche agli elementi del package stesso di stampare qualcosa a video per l'utente senza che questo modifichi pesantemente l'ordine in cui i Thread vengono schedulati dalla JVM.

Il codice relativo a questa classe è banale ed è il seguente:

```
publicclass OutputQueue {
   privatestatic LinkedList<String> messages = new LinkedList<String>();

   publicstaticsynchronizedvoid insert(String m) {
      messages.addLast(m);
   }

   publicstaticsynchronizedvoid print() {
      Iterator i = messages.iterator();
      while(i.hasNext()) {
         System.out.println(i.next());
      }
   }
}
```

Come si vede la classe è pubblica per utilizzo all'esterno del package, i metodi di tale classe sono stati dichiarati statici per evitare di dover definire un oggetto di tipo OutputQueue per poter invocare i metodi stessi.
La lista dove i messaggi vengono accordati è stata realizzata con una Java LinkedList, presente nel package java.util e che oramai fa parte dello standard (è stata introdotta nella versione 1.4.2).

- Il metodo insert() come si vede non fa altro che aggiungere alla lista statica un nuovo elemento in fondo alla lista, dopo aver acquisito la mutua esclusione sulla lista stessa.
- Il metodo print() invece serve a scorrere tutti gli elementi della lista stampando uno ad uno le stringhe contenute in essa, a partire dalla testa, anche questo metodo è synchronized e quindi viene eseguito in mutua esclusione.

Il metodo print() deve essere eseguito come ultima azione del main() sia che si termini correttamente, sia che la computazione venga arrestata a causa di un errore, dunque per far si che effettivamente il comportamento sia quello desiderato si userà uno shutdown hook che verrà installato all'inizio del main(), e non fa altro che specificare che all'uscita deve essere invocato il metodo print della classe OutputQueue

1.2) SyncSupport.Message

Questa classe rappresenta un messaggio che viene scambiato in modo sincrono tra i Thread, il codice relativo a tale classe è il seguente:

```
publicclass Message<T> {
   public SyncThread sender;
   public T inf;
}
```

Dunque la classe è pubblica per l'utilizzo all'esterno del package, e non contiene metodi.

È una classe generica, ossia il suo membro pubblico "inf" non è di un tipo prefissato, ma di un tipo generico, e questo significa che è possibile creare messaggi che contengano informazioni di qualunque tipo.
Il membro pubblico "sender" rappresenta invece il mittente del messaggio, che viene appunto visto come un riferimento ad un oggetto di tipo SyncThread.
Gli oggetti SyncThread rappresentano i thread che oltre ad avere le caratteristiche fornite di default dalla JVM sono anche in grado di comunicare con altri SyncThread per mezzo del meccanismo di comunicazione a scambio di messaggi sincrono che si è implementato.

1.3) SyncSupport.Port

Questa classe rappresenta una singola porta.
Per l'implementazione del meccanismo di comunicazione a scambio di messaggi sincrono si è deciso di operare in modo più simile possibile a quanto visto a lezione, questo significa che una porta (oggetto di classe Port), deve essere privata del thread proprietario che la ha definita, ed inoltre (per come visto nella implementazione del kernel a scambio di messaggi sincrono) farebbe comodo avere una sorta di "descrittore" del thread, contenente tra le altre cose un array di porte dove si individua una porta tramite il suo indice.

Questo meccanismo lo vedremo più in dettaglio nella sezione 1.4, si noti comunque che ogni oggetto di tipo SyncThread contiene un array di porte di nome thread_ports, e che rappresenta appunto l'array di cui sopra.

Il codice relativo alla classe Port è il seguente

```
publicclass Port<T> {
   staticfinalintN = 10;
   intindex;
   SyncThread owner;
   Message<T>[] messagequeue;
   intfirst;
   intlast;
   intcont;
   finalstatic Lock lock = new ReentrantLock();
   Condition[] blockedsenderqueue;

   public Port(SyncThread t) {
      first = 0; cont = 0; last = 0;
      messagequeue = new Message[N];
      blockedsenderqueue = new Condition[N];
      for(int i = 0; i < N; i++) {
         blockedsenderqueue[i] = lock.newCondition();
         messagequeue[i] = new Message<T>();
      }
      owner = t;
      for(int i = 0; i < SyncThread.M; i++)
         if(t.thread_ports[i] == null) {
            t.thread_ports[i]=this;
            index = i;
            return;
         }
      OutputQueue.insert("Maximum port definitions number exceeded");
      System.exit(0);
   }
}
```

Come si vede ogni porta ha un tipo T (anche la porta è una classe generica) che corrisponde al tipo di messaggi che possono transitare lungo il canale di comunicazione asimmetrico.
I membri sono:

- N: costante statica che definisce il numero massimo di mittenti per ogni singola porta (il valore 10 che le è stato assegnato è del tutto arbitrario).
- index: intero corrispondente all'indice della porta nell'array di porte nel SyncThread proprietario della porta stessa.
- owner: riferimento al SyncThread proprietario.
- messagequeue: coda di messaggi di tipo T presenti nella porta, essendo che stiamo parlando di comunicazione sincrona, i messaggi accodati sono quelli relativi ai mittenti in attesa di

ricezione, e sono al massimo N (essendo N i massimi mittenti).
- first, last, cont: interi per la gestione della coda di messaggi.
- blockedsenderqueue: è un array di N variabili condition, la possibilità di poter dichiarare variabili condition e lock arbitrariamente in Java è stata introdotta con la versione 5.0 (pertanto richiesta). Quando un mittente si accorge che il ricevente non è ancora pronto a ricevere, si deve bloccare attendendo l'arrivo del ricevente... nel caso visto a lezione, ciò si faceva semplicemente con la funzione di sistema assegnazione_CPU, ma non esiste una cosa analoga in Java (se non la suspend(), che però è stata deprecata) e per questo si deve proprio bloccare il mittente, per farlo si usa una variabile condition. Il fatto di dover utilizzare N diverse variabili condition per N diversi mittenti è che dobbiamo essere sicuri che quando il ricevente effettivamente preleva un messaggio, non si riattivi un mittente a caso ma bensì quello in attesa da più tempo.
- lock: è un membro statico di tipo Lock, ciò significa che vi è solo un oggetto lock in tutto il programma, essendo appunto statico. Questo viene usato per rappresentare la mutua esclusione tipica delle primitive di nucleo, nel nostro caso le funzioni che andremo a vedere nel seguito operano anche su più porte contemporaneamente, e quindi questo lock viene usato per assicurare che la consistenza non vada persa per operazioni eseguite non in mutua esclusione.

Per la classe porta viene poi definito il costruttore, questo prende come argomento un riferimento ad oggetto di tipo SyncThread che diventa il proprietario della porta.
Prima di tutto il costruttore inizializza gli array e le code descritti sopra, poi fissa il proprietario della porta con il riferimento al SyncThread passato al costruttore, e infine inizializza l'indice "index" della porta nell'array di porte del proprietario (quest'ultima operazione può fallire nel caso in cui per il thread proprietario abbia definito sufficienti porte da colmare l'array, in tal caso l'esecuzione del programma termina segnalando l'errore all'utente).

Una porta dovrebbe allora essere definita da un SyncThread come segue:
Port<T> nome_porta = new Port<T>(this);

Questa definizione dovrebbe stare nella sezione public della classe relativa al SyncThread, in modo tale che altri SyncThread possano mandare messaggi a questa porta semplicemente indicandola con una sintassi del tipo:
nome_thread.nome_porta

1.4) SyncSupport.SyncThread

Questa classe rappresenta un Thread capace di comunicare con altri SyncThread tramite il meccanismo emulato di comunicazione sincrona.
Per spiegare il funzionamento cominciamo illustrando i membri di ogni oggetto di tipo SyncThread.

```
publicabstractclass SyncThread extends Thread {
   staticfinalintM = 10;
   Port[] thread_ports = new Port[M];
   boolean[] recvblock = newboolean[M];
   Condition blockedreceiver;
   intavailableport;
   ... (vedi oltre) ...
```

Come si vede la classe SyncThread è astratta, e questo significa che non può essere istanziata direttamente, infatti la classe SyncThread estende la classe Thread lasciando tuttavia il metodo run() non definito.
L'utente che volesse usare un SyncThread allora dovrà derivare una nuova classe che rappresenta un particolare tipo di thread, estendendo la classe SyncThread, ad esempio:

```
class Writer extends SyncThread {...}
```

E dovrebbe inoltre definire il metodo run() per fissare il corpo del SyncThread.
Descriviamo adesso i membri di cui sopra.

- M: Costante statica che definisce il numero massimo di porte per ogni SyncThread (il valore 10 a cui è stata fissata è del tutto arbitrario).
- thread_ports: Array di porte relativo al SyncThread, consente di associare un indice ad una porta del thread.
- recvblock: Array di booleani che indicano se il SyncThread è bloccato in attesa di ricevere, ed indicano anche da quali porte sta attendendo un messaggio.
- blockedreceiver: Variabile condition che consente di bloccare il SyncThread qualora si cercasse di ricevere un messaggio da una o più porte quando il mittente non è ancora arrivato al punto di rendez-vous.
- availableport: questo membro non viene inizializzato dal costruttore, e viene infatti utilizzato esclusivamente dalla funzione receiveany(), il suo significato verrà pertanto spiegato nel seguito

Mostriamo adesso il costruttore per la classe SyncThread

```
...
public SyncThread() {
   super();
   blockedreceiver = Port.lock.newCondition();
   for(int i = 0; i < M; i++) {
      recvblock[i] = false;
      thread_ports[i] = null;
   }
}
... (vedi oltre) ...
```

Questo non fa altro che invocare il costruttore della classe Thread da cui il SyncThread deriva, dopo di che fa alcune operazioni di inizializzazione sull'array di porte thread_ports, sull'array di boolean recvblock ed inoltre costruisce la variabile condition blockedreceiver.

La classe SyncThread, oltre al costruttore, contiene i 3 metodi che consentono la comunicazione sincrona desiderata, ora li analizzeremo uno ad uno.

1.4.1) metodo receive()

il codice è il seguente

```
...
protected <T> void receive(Message<T> m, Port<T> p) {
   if(this != p.owner) {
      OutputQueue.insert("Thread "+this.getId()+
            " tryed receiving from a private port of thread "+
            p.owner.getId());
      System.exit(0);
   }
   int first = -1;
   boolean signal = false;
   Port.lock.lock();
   if(p.cont==0) {
      recvblock[p.index]=true;
      blockedreceiver.awaitUninterruptibly();
   } else {
      signal = true;
      first = p.first;
   }
```

```
      p.cont--;
      Message<T> extr_m = p.messagequeue[p.first];
      p.first = (p.first+1)%Port.N;
      m.inf = extr_m.inf;
      m.sender = extr_m.sender;
      if(signal) {
         p.blockedsenderqueue[first].signal();
      }
      Port.lock.unlock();
      return;
   }
   ... (vedi oltre) ...
```

Come si vede la receive() ha modificatore protected, e questo significa che potrà essere invocata solamente da una classe derivata (quella definita dall'utente) e non da chiunque.
La prima operazione compiuta dalla receive() è capire se in effetti la porta da cui si è detto di voler ricevere è proprietà del thread che ha la ha invocata, e questo è lo scopo del primo if.
Questa operazione non necessita di acquisire la mutua esclusione sulle porte (usando l'oggetto statico lock sopra descritto) e pertanto si può fare immediatamente.
Se il test non causa la terminazione dell'esecuzione del programma (la porta specificata è corretta), allora si acquisisce il lock sulle porte.
A questo punto siamo ad un bivio:

- o non si hanno ancora messaggi accodati sulla porta da parte di mittenti che attendono di essere svegliati
- oppure vi sono messaggi da prelevare dalla coda interna alla porta (e un mittente da svegliare)

La distinzione tra i due casi si fa semplicemente osservando se la coda dei messaggi è o meno vuota, tramite il campo p.cont.
Nel primo caso allora il chiamante deve bloccarsi, e per farlo si segnala prima che ci stiamo bloccando in attesa di messaggi sulla porta di indice specificato, e poi si fa una wait() sulla variabile condition SyncThread.blockedreceiver appositamente prevista.
Deve essere il mittente in questo caso (accorgendosi che il ricevente è bloccato in attesa di un messaggio) a svegliare il ricevente, ma prima di farlo deve essere lui stesso a segnalare che il recevente non è più bloccato in attesa di un messaggio su una data porta.
Se infatti fosse il ricevente stesso a modificare nuovamente il vettore recvblock locale una volta riottenuto il lock e tornato in esecuzione allora dall'istante in cui il mittente ha invocato la signal() a quello in cui il ricevente riacquisisce la mutua esclusione, lo stato continua a segnalare che il ricevente è bloccato in attesa di messaggi su una data porta e quindi altri mittenti non si bloccherebbero in attesa pensando di dover essere loro stessi a svegliare il ricevente e facendo una signal() a vuoto (ma comunque non bloccandosi, come invece si impone in caso di comunicazione sincrona).
Nel secondo caso invece il mittente è già arrivato al punto di rendez-vous e per questo il ricevente può evitare di bloccarsi e continua semplicemente, esso però dovrà fare una signal() sulla variabile condition su cui il mittente del primo messaggio in coda si è bloccato, il mittente viene allora svegliato dopo che il messaggio è stato letto... svegliarlo prima non sarebbe stato un problema perché tanto il mittente non potrà fare altre operazioni sulle porte fintanto che il ricevente non rilascia la mutua esclusione, dopo aver letto effettivamente il messaggio (la scelta che fatta è allora stata guidata da una pura formalità).

Qualunque dei due rami sia stato scelto, all'uscita dall'if-then-else abbiamo un messaggio da prelevare dalla coda, e dunque si fanno le operazioni banali di cui sopra, registrando nei membri dell'oggetto Message<T> riferito dal parametro m i valori ricevuti.
Se il mittente andava svegliato allora a questo punto si fa la signal().
Infine si rilascia la mutua esclusione sulle porte.

1.4.2) metodo receiveany()

il codice è il seguente

```
...
protected <T> int receiveany(Message<T> m, Port[] p) {
   Port<T> port;
   for(int i = 0; i < p.length; i++) {
      Port<T> port = (Port<T>)p[i];
      if(this != port.owner) {
         OutputQueue.insert("Thread "+Thread.currentThread().getId()+
               " tryed receiving from a private port of thread "+
               port.owner.getId());
         System.exit(0);
      }
   }
   boolean signal = true;
   int first = -1;
   Port.lock.lock();
   availableport = -1;
   for(int i = 0; i < p.length; i++) {
      port = (Port<T>)p[i];
      if(port.cont > 0) {
         availableport = port.index;
         first = port.first;
         break;
      }
   }
   if(availableport == -1) {
      signal = false;
      for(int i = 0; i < p.length; i++) {
         port = (Port<T>)p[i];
         recvblock[port.index] = true;
      }
      blockedreceiver.awaitUninterruptibly();
   }
   port = thread_ports[availableport];
   port.cont--;
   Message<T> extr_m = (Message<T>)port.messagequeue[port.first];
   port.first = (port.first+1)%Port.N;
   m.inf = extr_m.inf;
   m.sender = extr_m.sender;
   if(signal) {
      port.blockedsenderqueue[first].signal();
   }
   Port.lock.unlock();
   returnavailableport;
}
... (vedi oltre) ...
```

Questa funzione somiglia vagamente alla receive(), tuttavia vi sono importante differenze che cercheremo di spiegare nel modo più completo possibile.
Notiamo subito che questa funzione non prende come parametro solo una porta, ma bensì un array di porte da cui il chiamante vuole ricevere un messaggio in modo non deterministico (da una qualunque di quelle porte).

Come prima operazione la receiveany() controlla che tutte le porte che sono state passate tramite l'array siano effettivamente proprietà del SyncThread da cui è invocata la receiveany(), in caso contrario si provvede a terminare l’esecuzione del programma segnalando anche un errore all'utente.

Dopo di che si inizializzano le variabili locali signal e first con un valore opportuno.
Tutte le operazioni svolte fino a questo punto non necessitano della mutua esclusione sulle porte, data dal membro statico lock definito in precedenza.
In questa funzione availableport viene utilizzato come indice della porta nel SyncThread da cui si dovrà ricevere dopo aver scoperto che vi sono dei messaggi o aver aspettato che tale situazione si verificasse grazie ad un mittente giunto successivamente.
availableport viene allora inizializzato ad un valore -1 che indica che non è ancora stata trovata una porta valida da cui prelevare un messaggio, questo viene fatto dopo aver ottenuto la mutua esclusione perché (come vedremo successivamente) anche i mittenti possono modificare la variabile availableport locale a questo SyncThread tramite la funzione send().
Dopo la inizializzazione di availableport si scorrono le porte specificate nell'array, e nel caso in cui una delle porte sia non vuota si provvede ad assegnare ad availableport l'indice della porta non vuota e da cui si preleverà subito un messaggio, inoltre si assegna a first la posizione del messaggio da estrarre nella coda di messaggi della porta.
Se alla fine dello scorrimento availableport è rimasto al valore -1 allora nessuna delle porte specificate conteneva già un messaggio da prelevare, quindi il ricevente si deve bloccare.
Il ricevente si bloccherà subito dopo aver specificato che si è bloccato su tutte le porte dell'array passato come parametro (all'interno dell'array di boolean recvblock[]), ed inoltre signal passerà dal valore true al valore false prima del bloccaggio del SyncThread.
La variabile locale signal indica se (trovata una porta valida) il mittente deve o meno essere svegliato: il mittente deve essere svegliato quando è arrivato al rendez-vous prima del ricevente, il che avviene quando una delle porte è stata trovata non vuota dal primo scorrimento dell'array di porte di cui sopra... se dopo il primo scorrimento una porta non è ancora stata trovata allora necessariamente il ricevente non deve risvegliare il mittente quando verrà a sua volta riattivato, perché il mittente non si bloccherà osservando che il ricevente è bloccato in attesa di un messaggio dalla porta dove sta inviando (vedi oltre).

Il mittente inoltre se si trovasse a dover svegliare il ricevente, indicherà all'interno di availableport l'indice della porta dove ha accodato il messaggio. Questa potrebbe sembrare una azione inutile potendo il ricevente che si era ipoteticamente bloccato sulla receiveany() fare un nuovo scorrimento dell'array di porte per trovarne una non vuota, tuttavia non è così...
Il nuovo scorrimento delle porte dell'array può portare a dei problemi, facciamo un esempio:
- Il thread T1 è in ascolto con receiveany sulle porte PA e PB, bloccato.
- Il thread T2 invia un messaggio sulla porta PB, riattivando T1, e continua
- Il thread T3 invia un messaggio sulla porta PA, bloccandosi (giustamente, visto che T2 ha segnalato che T1 non è + in attesa di messaggi su nessuna porta, ma anzi attivo)
- Il thread T1 finalmente riesce a riottenere la mutua esclusione, e fa un nuovo scorrimento dell'array, accorgendosi che la prima (PA) non è vuota.
- Il thread T1 allora continua senza svegliare T3, il vero mittente, credendo infatti da essere stato svegliato da T3 stesso, mentre a svegliarlo è stato T2
- *Dunque T3 ha inviato un messaggio a T1, tuttavia T1 è entrato in rendez-vous con T2.*

Per ovviare al problema di cui sopra è sufficiente che il thread T1 non faccia un nuovo scorrimento dell'array, ma è necessario che availableport sia settato al giusto valore da T2, quando il messaggio viene inviato, e che T1 (risvegliandosi) utilizzi quell'indice come quello della porta da cui ricevere.

Dopo il risveglio, o in caso di primo passaggio con esito positivo, e se signal è ancora al valore true, si sveglia il mittente del messaggio da prelevare, ma solo dopo (per pulizia) aver letto il messaggio dalla porta scelta, e copiato il contenuto all'interno dell'oggetto riferito dal parametro m.

Si rilascia infine il lock sulle porte, restituendo subito dopo al chiamante l'indice della porta da cui si è effettivamente ricevuto, indice che non corrisponde all'ordine con cui le porte sono inserite nel

vettore p passato come parametro, ma bensì corrispondente all'ordine con cui le porte sono state definite all'interno del SyncThread proprietario (chiamante della receiveany), infatti ricordiamo che availableport è un indice dell'array thread_ports, privato del SyncThread.

1.4.3) metodo send()

Il codice è il seguente

```
...
protected <T> void send(Message<T> m, Port<T> p) {
   Port.lock.lock();
   if(p.cont == Port.N) {
      OutputQueue.insert("Too many thread sending messages " +
            "to private port "+p.index+
            " of thread "+p.owner.getId());
      System.exit(0);
   }
   int k = p.last;
   p.cont++;
   p.messagequeue[p.last] = m;
   p.last = (p.last+1)%Port.N;
   if(p.owner.recvblock[p.index]) {
      for(int i = 0; i < SyncThread.M; i++)
         p.owner.recvblock[i]=false;
      p.owner.availableport = p.index;
      p.owner.blockedreceiver.signal();
   } else {
      p.blockedsenderqueue[k].awaitUninterruptibly();
   }
   Port.lock.unlock();
 }
}
```

Come si vede la prima operazione della send() è di acquisire la mutua esclusione sulle porte del nostro kernel a scambio di messaggi sincroni emulato.
Il lock serve infatti in questo caso anche per fare un controllo sul numero si messaggi presenti nella coda della porta in attesa di essere ricevuti, se questo numero supera Port.N, allora l’esecuzione del programma termina segnalando l'errore all'utente.
Subito dopo si salva la posizione dove il messaggio verrà inserito in coda nella variabile locale k.
Inserito il messaggio allora si valuta se il ricevente era o meno già arrivato al punto di rendez-vous in attesa di un mittente.

- In caso il ricevente sia bloccato sulla porta dove si ha appena inserito il messaggio, allora si sveglia il ricevente segnalando inoltre che il suo stato è attivo, e indicando anche l'indice della porta su cui si è inviato il messaggio come nuovo availableport del thread ricevente (questo viene fatto in quanto operazione necessaria se il thread ricevente era bloccato su una receiveany()).
- Nel caso invece in cui il ricevente non sia bloccato in attesa sulla porta dove il messaggio è stato accodato, significa che il mittente è arrivato per primo al punto di rendez-vous e per questo deve bloccarsi in attesa del ricevente. Dunque si blocca in attesa sulla apposita variabile condition presente nell'array blockedsenderqueue[] all'interno della porta stessa (usando l'indice k precedentemente salvato).

Qualunque sia il ramo scelto, l'unica cosa che rimane da fare all'uscita dall'if-then-else è rilasciare la mutua esclusione sulle porte, e così si fa.

2) Implementazione di un Server (gestore di una risorsa) funzionante con il protocollo Lettori/Scrittori usando il meccanismo di comunicazione sincrono appena realizzato

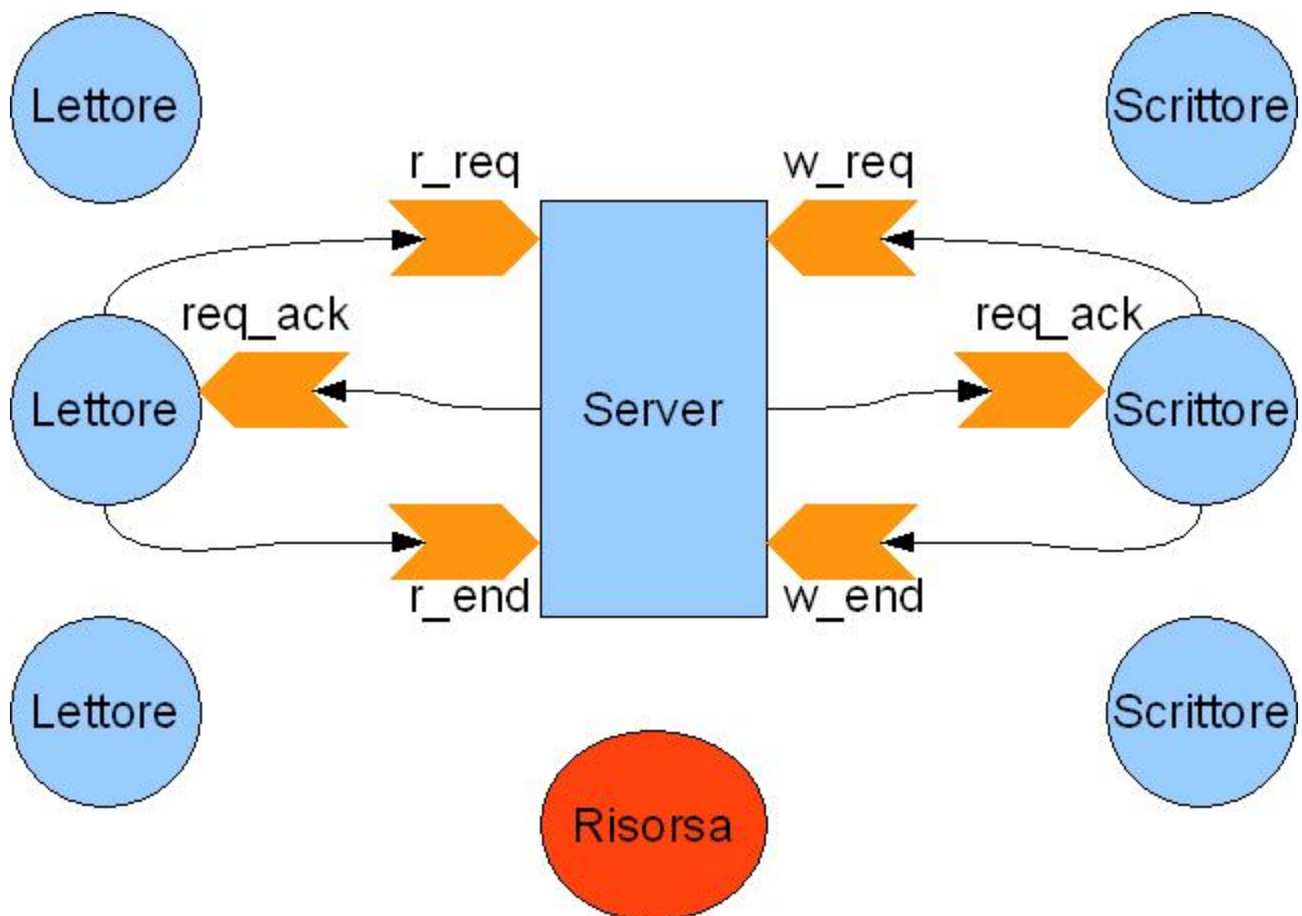
Realizzato il meccanismo di comunicazione sincrono voluto, è facile a questo punto implementare un processo Server che gestisca una ipotetica risorsa a cui devono accedere due gruppi di thread (lettori e scrittori) tramite il protocollo Lettori/Scrittori.
Per realizzare il Server si è fatta l'ipotesi che la risorsa gestita non sia inclusa nel server stesso, bensì si suppone che sia i Lettori che gli Scrittori sappiano come accedere alla risorsa in questione, ma lo facciano solo quando ottengono la autorizzazione dal server gestore che andremo a scrivere.
I requisiti che si vogliono dare alla gestione della risorsa sono i seguenti, sono stati visti anche a lezione, e assicurano che sia impossibile la starvation sia dei thread lettori che dei thread scrittori:

1) Si consentono operazioni di lettura contemporaneamente
2) Le operazioni di scrittura sono mutuamente esclusive sia con altre operazioni di scrittura, sia con operazioni di lettura
3) Una nuova lettura non deve essere consentita se abbiamo dei processi scrittori in attesa di ottenere la risorsa, e alla fine delle letture in corso deve essere privilegiato un processo scrittore rispetto ai lettori in attesa (questo assicura assenza di starvation dei thread scrittori)
4) Alla fine di una scrittura devono essere privilegiati i lettori in attesa rispetto ad ulteriori processi scrittori (questo assicura assenza di starvation dei thread lettori, se ce ne sono in attesa allora si svegliano tutti)

Si capisce subito allora che per sapere se una operazione di lettura deve consentita è necessario sapere se ci sono degli scrittori in attesa di poter accedere alla risorsa, e allo stesso modo quando si finisce una scrittura bisogna sapere se ci sono dei lettori in attesa per capire se posso abilitare nuovi scrittori o rilasciare la risorsa.
Questo significa che per poter prendere le decisioni non posso basarmi solamente sullo stato della risorsa, devo invece ricevere sempre le richieste che mi pervengono e tenere traccia all'interno del Server di eventuali lettori o scrittori che hanno inoltrato delle richieste che non è stato possibile esaudire immediatamente.
Questo implica anche che uno scrittore (o un lettore) non deve interpretare il ritorno dalla send() (ricezione da parte del Server della richiesta) come uno consenso da parte del Server ad accedere alla risorsa, infatti il Server deve rispondere in altro modo e gli scrittori (o i lettori) devono essere pronti a ricevere il consenso su una porta locale.
Dunque avremo che globalmente ci saranno

- 4 porte sul Server:
  - r_req: porta attraverso cui pervengono le richieste dei lettori
  - w_req: porta attraverso cui pervengono le richieste degli scrittori
  - r_end: porta attraverso cui pervengono i rilasci della risorsa da parte dei lettori
  - w_end: porta attraverso cui pervengono i rilasci della risorsa da parte degli scrittori
- 1 porta sui lettori:
  - req_ack: per ricevere la concessione della risorsa da parte del Server
- 1 porta sugli scrittori:
  - req_ack: per ricevere la concessione della risorsa da parte del Server

Nella seguente figura è mostrata la situazione risultante, dove le porte sono rappresentate in arancione e gli scambi di messaggi sono rappresentati tramiti frecce.
Notare che le porte e gli scambi di messaggi per i lettori sono rappresentati solo per uno di essi, e lo stesso vale per gli scrittori.



Possiamo a questo punto scrivere il codice per il server, che sarà il seguente:

```
class Server extends SyncThread {
   Port<Byte> r_req = new Port<Byte>(this);
   Port<Byte> w_req = new Port<Byte>(this);
   Port<Byte> r_end = new Port<Byte>(this);
   Port<Byte> w_end = new Port<Byte>(this);

   publicvoid run() {
      LinkedList<SyncThread> q_readers = new LinkedList<SyncThread>();
      LinkedList<SyncThread> q_writers = new LinkedList<SyncThread>();
      boolean active_writer = false;
      int active_readers = 0;
      Port[] array = new Port[4];
      array[0] = r_req;
      array[1] = w_req;
      array[2] = r_end;
      array[3] = w_end;
      Message<Byte> m = new Message<Byte>();
      Message<Byte> rep = new Message<Byte>();
      rep.sender=this;
      rep.inf=0;
      while(true) {
         int index = receiveany(m, array);
```

```
        switch(index) {
        case 0: // reader request received
          if(!active_writer && q_writers.isEmpty()) {
            active_readers++;
            Reader r = (Reader)m.sender;
            send(rep, r.req_ack);
          } else {
            q_readers.addLast(m.sender);
          }
          break;
        case 1: // writer request received
          if(!active_writer && active_readers==0) {
            active_writer = true;
            Writer w = (Writer)m.sender;
            send(rep, w.req_ack);
          } else {
            q_writers.addLast(m.sender);
          }
          break;
        case 2: // finished reading
          active_readers--;
          if(active_readers==0 && !q_writers.isEmpty()) {
            active_writer = true;
            Writer w = (Writer)q_writers.removeFirst();
            send(rep, w.req_ack);
          }
          break;
        case 3: // finished writing
          active_writer = false;
          if(!q_readers.isEmpty()) {
            while(!q_readers.isEmpty()) {
              active_readers++;
              Reader r = (Reader)q_readers.removeFirst();
              send(rep, r.req_ack);
            }
          } elseif(!q_writers.isEmpty()) {
            active_writer = true;
            Writer w = (Writer)q_writers.removeFirst();
            send(rep, w.req_ack);
          }
          break;
        default:
          OutputQueue.insert("Fatal error on the server");
          System.exit(0);
        }
      }
    }
}
```

Il funzionamento è abbastanza intuitivo, diamo solo alcune spiegazioni:

- q_readers: coda dei processi lettori in attesa di poter accedere alla risorsa
- q_writers: coda dei processi scrittori in attesa di poter accedere alla risorsa
- active_writer: vero se la risorsa è posseduta da un lettore
- active_readers: numero di lettori attivi (che possiedono la risorsa)
- array: vettore delle 4 porte del Server, da passare alla receiveany() per ricevere da una qualunque delle 4 porte in modo non deterministico.
- m: oggetto di tipo Message<Byte> usato per immagazzinare le informazioni restituite da receiveany()
- rep: oggetto di tipo Message<Byte> che rappresenta l'unica forma di messaggio che il server manda a lettori o scrittori

Come si vede in questo problema non è importante il tipo di messaggio che i thread si scambiano, quello che conta è che si scambino un messaggio e che sia contenuto il mittente all'interno dello stesso, per questo si usa il tipo Byte predefinito in Java (avvicinandosi al "signal" visto a lezione).
Il corpo vero e proprio del Server non è altro che un ciclo infinito in cui si fa ogni volta receiveany() e si processa il messaggio arrivato, fino a che non si viene forzatamente interrotti oppure si verifica un errore.
In caso di funzionamento normale l'indice ritornato dalla receiveany() è l'indice della porta da cui è stato ricevuto il messaggio all'interno dell'array thread_ports[] del SyncThread relativo al Server.

A seconda del valore restituito si hanno diversi comportamenti

- index = 0 → il messaggio è stato ricevuto dalla porta r_req, quindi ho che
  - se la risorsa non è posseduta da uno scrittore e non si hanno scrittori in attesa, allora essa viene concessa al lettore che ha fatto richiesta
  - altrimenti la risorsa non può essere concessa immediatamente e il lettore viene inserito nella apposita coda q_readers
- index = 1 → il messaggio è stato ricevuto dalla porta w_req, quindi ho che
  - se non si hanno lettori attivi sulla risorsa ed essa non è posseduta da un altro scrittore allora essa viene concessa allo scrittore che ha fatto richiesta
  - altrimenti la risorsa non può essere concessa immediatamente e lo scrittore viene inserito nella apposita coda q_writers
- index = 2 → il messaggio è stato ricevuto dalla porta r_end, un lettore allora rilascia la risorsa, e se quel lettore era l'ultimo attivo allora si controlla se ci sono scrittori in attesa e in caso positivo si attiva il primo di essi (mandandogli il messaggio di risposta sulla porta req_ack)
- index = 3 → il messaggio è stato ricevuto dalla porta w_end, lo scrittore ha rilasciato la risorsa.
  - Se ci sono lettori in attesa si attivano tutti mandando ad ognuno il messaggio di risposta
  - Altrimenti se ci sono scrittori in attesa si attiva il primo di essi mandandogli il messaggio di risposta
  - Altrimenti si lascia semplicemente la risorsa libera

Visto il codice relativo al Server è semplice capire come si devono comportare i lettori e gli scrittori

- inizialmente mandano la richiesta sulla porta r_req o w_req del Server
- aspettano una risposta dal Server sulla porta locale req_ack
- operano sulla risorsa
- mandano il messaggio di rilascio sulla porta r_end o w_end del Server

A scopo di semplice test, sono state definite due classi per i thread lettori e scrittori, come segue:

```
class Reader extends SyncThread {
   Port<Byte> req_ack = new Port<Byte>(this);
   Server s;
   intid;
   intcycles_number;
   public Reader(Server serv, int i, int k) {
      super();
      s = serv;
      id = i;
      cycles_number = k;
   }
   publicvoid run() {
      Message<Byte> req = new Message<Byte>();
      req.sender=this;
      req.inf=0;
```

```
        Message<Byte> m = new Message<Byte>();
        Random generator = new Random();
        for(int i = 0; i < cycles_number; i++) {
            send(req,s.r_req);
            receive(m,req_ack);
            OutputQueue.insert("Reader "+id+" is now reading");
            try{sleep(1+generator.nextInt(3000));}
            catch(InterruptedException e) {}
            OutputQueue.insert("Reader "+id+" finished reading");
            send(req,s.r_end);
            try{sleep(1+generator.nextInt(3000));}
            catch(InterruptedException e) {}
        }
    }
}

class Writer extends SyncThread {
    Port<Byte> req_ack = new Port<Byte>(this);
    Server s;
    intid;
    intcycles_number;
    public Writer(Server serv, int i, int k) {
        super();
        s = serv;
        id = i;
        cycles_number = k;
    }
    publicvoid run() {
        Message<Byte> req = new Message<Byte>();
        req.sender=this;
        req.inf=0;
        Message<Byte> m = new Message<Byte>();
        Random generator = new Random();
        for(int i = 0; i < cycles_number; i++) {
            send(req,s.w_req);
            receive(m,req_ack);
            OutputQueue.insert("Writer "+id+" is now writing");
            try{sleep(1+generator.nextInt(3000));}
            catch(InterruptedException e) {}
            OutputQueue.insert("Writer "+id+" finished writing");
            send(req,s.w_end);
            try{sleep(1+generator.nextInt(3000));}
            catch(InterruptedException e) {}
        }
    }
}
```

Come si vede queste due classi forniscono funzionalità aggiuntive rispetto a quelle del generico lettore o scrittore appena descritte.

- Ognuna delle due ha un particolare costruttore che consente di
  - Specificare quale è il server a cui lo scrittore o il lettore deve rivolgersi per accedere alla risorsa (passato come riferimento)
  - Specificare un intero che viene utilizzato come numero identificativo del lettore o dello scrittore e verrà stampato come output del programma
  - Specificare il numero di cicli che corrisponde a quante volte il lettore o lo scrittore deve accedere alla risorsa e rilasciarla
- Si hanno dei ritardi di tempo casuali che vanno da 0 a 3 secondi che rappresentano
  - Quanto tempo il lettore o lo scrittore opera sulla risorsa
  - Quanto tempo il lettore o lo scrittore aspetta (una volta completato un ciclo) per cominciarne uno nuovo

Queste tre classi sono state incluse nel file Readers_Writers.java.
Per l'utilizzo corretto del package SyncSupport.SafeOutput si è previsto anche di creare una classe che permetta di eseguire le operazioni di stampa prima dell'uscita dal programma, essa ha la seguente struttura:

```
class PrintBeforeExit extends Thread {
  publicvoid run() {
    OutputQueue.print();
  }
}
```

E rappresenta semplicemente un thread eseguito come shutdown hook.
La classe contenente la funzione main() per questo programma di prova è la seguente

```
publicclass Readers_Writers {
  publicstaticvoid main(String[] args) {
    PrintBeforeExit pbe = new PrintBeforeExit();
    Runtime.getRuntime().addShutdownHook(pbe);

    Server s = new Server();
    Writer w1 = new Writer(s, 1, 5);
    Writer w2 = new Writer(s, 2, 5);
    Reader r1 = new Reader(s, 1, 5);
    Reader r2 = new Reader(s, 2, 5);

    s.start();
    r1.start();
    w1.start();
    r2.start();
    w2.start();

    try{
      r1.join();
      r2.join();
      w1.join();
      w2.join();
    } catch(InterruptedException e) {}
    System.exit(0);
  }
}
```

Si capisce allora che il test si lancia (una volta compilato tutto) con il comando:
java Readers_Writers

Come si vede si è allora previsto:
- Un unico Server s
- Due thread lettori r1 ed r2 aventi numeri identificativi rispettivamente 1 e 2, entrambi eseguono un ciclo di 5 letture con rilascio
- Due thread scrittori w1 e w2 aventi numeri identificativi rispettivamente 1 e 2, entrambi eseguono un ciclo di 5 scritture con rilascio

Il main thread attenderà la terminazione dei due lettori e dei due scrittori, e quando questa arriverà (o anche se venisse interrotta l'attesa da un segnale) si avrà la terminazione dell'esecuzione del programma, con la stampa a video dell'output immagazzinato nella coda della classe OutputQueue.

L'output arriva allora solo al termine della computazione... avendo introdotto dei ritardi piuttosto evidenti, l'utente deve allora aspettare pazientemente la fine della simulazione per avere la stampa dell'output a video.

Un output esemplificativo è il seguente:

Writer 1 is now writing
Writer 1 finished writing
Reader 1 is now reading
Reader 2 is now reading
Reader 2 finished reading
Reader 1 finished reading
Writer 2 is now writing
Writer 2 finished writing
Reader 2 is now reading
Reader 1 is now reading
Reader 2 finished reading
Reader 1 finished reading
Writer 1 is now writing
Writer 1 finished writing
Reader 1 is now reading
Reader 2 is now reading
Reader 2 finished reading
Reader 1 finished reading
Writer 2 is now writing
Writer 2 finished writing
Reader 2 is now reading
Reader 2 finished reading
Writer 1 is now writing
Writer 1 finished writing
Reader 2 is now reading
Reader 1 is now reading
Reader 2 finished reading
Reader 1 finished reading
Writer 2 is now writing
Writer 2 finished writing
Reader 1 is now reading
Reader 1 finished reading
Writer 1 is now writing
Writer 1 finished writing
Writer 2 is now writing
Writer 2 finished writing
Writer 2 is now writing
Writer 2 finished writing
Writer 1 is now writing
Writer 1 finished writing

Che come si nota non evidenza alcuna anomalia rispetto al comportamento atteso.

3) Script ausiliari

Per la corretta compilazione ed esecuzione del progetto è necessaria una versione di Java 5.0 o superiore.

Per l'utente che utilizzi una macchina Linux o Unix, sono stati previsti due script che possono essere particolarmente utili per una rapida compilazione dei file .java che sono stati scritti in questo progetto, e per una loro rapida rimozione qualora si volesse ripulire la cartella base del progetto.

- make.sh: è un piccolo script che semplicemente esegue la compilazione java su tutti i file .java di questo progetto, producendo in particolare anche i file .class necessari per il test di cui sopra.
  Un piccolo output viene mostrato a video quando la compilazione è terminata, fornendo le

istruzioni di base per avviare il test.

- clean.sh: questo piccolo script consente di eliminare tutti i file .class prodotti in fase di compilazione usando o meno lo script make.sh di cui sopra