ESERCITAZIONE TIGA: TCP Client, TCP Server e multithreading

Un host (*BufferHost*) offre il servizio di *Buffer di stringhe,* per host produttori (*ProducerHost*) e consumatori (*ConsumerHost*), interagenti secondo un modello a scambio di messaggi, mediante l'uso del protocollo di trasporto TCP (Fig.1).

Ciascun host produttore invia un numero P (parametro utente) di stringhe, ad intervalli random di 3÷5 secondi l'una dall'altra, sulla porta 8080 del *BufferHost*. Ciascun host consumatore richiede un numero C (parametro utente) di stringhe, ad intervalli random di 3÷5 secondi tra una precedente ricezione e la successiva richiesta, sulla porta 8080 del *BufferHost*.

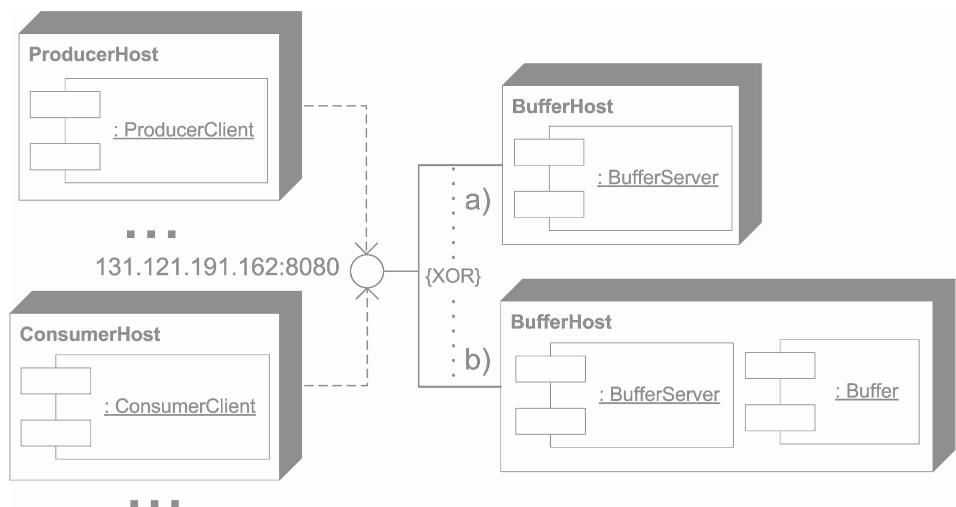

Fig.1 – Distribuzione dei componenti nel caso di Server senza thread (a) e con thread (b)

Sviluppare in Java le classi `ProducerClient`, `ConsumerClient` e `BufferServer` che, distribuite sui nodi della rete come raffigurato in Fig.1, realizzino quanto suddetto. Per le attese del consumatore e produttore, usare il metodo (java.lang.Thread)

```
public static void sleep(long millis) throws InterruptedException
```

Durante l'attesa di un `ConsumerClient` dovuta a buffer vuoto, si possono implementare due diverse gestioni delle risorse di rete, a seconda dei tempi medi di attesa e della disponibilità di risorse del sistema. Nella prima gestione, il canale di comunicazione viene chiuso, ed il `ConsumerClient` si pone in ascolto su una porta concordata, in attesa della stringa. Nella seconda modalità, di più semplice realizzazione, il canale di comunicazione viene mantenuto in attesa della risposta del `BufferServer`.

Si realizzi `BufferServer` in due diverse modalità: (a) senza l'uso di thread e (b) con i thread, secondo le seguenti indicazioni.

a) TCP Server

Si supponga per semplicità che il buffer sia *illimitato*, e che le richieste sopraggiunte a coda vuota siano poste in una coda FIFO[1] e servite appena possibile. Per gestire richieste ravvicinate sulla porta di ascolto del `BufferServer` si usi seguente costruttore (java.net.ServerSocket), dove l'intero `backlog`[2] indica il numero massimo di richieste TCP che possono essere automaticamente accettate ed accodate senza che la `accept()` sia ancora stata invocata.

```
public ServerSocket(int port, int backlog) throws IOException
```

Se la coda di backlog diventa piena, un tentativo di connessione sulla corrispondente porta di ascolto del BufferHost viene rifiutato. Catturare l'eccezione generata e gestirla provando a riconnettersi ad intervalli random di 5÷7 secondi; al terzo tentativo consecutivo fallito, terminare l'esecuzione del produttore/consumatore relativo.

b) TCP Server multi-thread

Si supponga che il buffer sia *limitato*, e si riprogetti il sottosistema *BufferHost* adoperando i thread per assegnare, ad ogni richiesta di servizio, un flusso sequenziale di controllo indipendente, provvedendo a gestire in mutua esclusione gli accessi multipli al buffer e sincronizzando tra loro gli accessi di produttori e consumatori.

In particolare, la classe `Buffer` implementa il buffer FIFO di dimensione limitata, con i metodi `put/get` che garantiscono ai thread un accesso mutuamente esclusivo e sincronizzato, evitando il blocco reciproco di tutte le attività del sistema (deadlock) o l'attesa indefinita di una di esse (starvation). Nella classe `BufferServer`, il metodo main esegue un loop infinito in cui accetta richieste di connessione su una porta e – per ognuna di esse – crea un thread per gestirle. In tal caso non è necessaria la gestione della coda di backlog, in quanto richieste ravvicinate sulla porta di ascolto sono immediatamente gestite mediante thread autonomi, pertanto il flusso di controllo relativo a `BufferServer` é quasi sempre bloccato sulla `accept`.

Mentre i consumatori rimangono in attesa del messaggio prima di inviare la prossima richiesta, i produttori inviano incessantemente messaggi, causando un aumento dei thread sospesi, nel caso di buffer pieno. Visualizzare, ad ogni richiesta di connessione accettata, il numero di thread sospesi, includendo in un `ThreadGroup` ogni thread creato, quindi adoperando il metodo `activeCount()` per conoscere il numero di thread attivi del gruppo. Sebbene i thread abbiano un costo inferiore ai processi, un loro aumento <u>indefinito</u> può esaurire le risorse di sistema. Quali grandezze dell'applicazione influenzano tale aumento ?

---

[1] *First In, First Out*, ossia il primo elemento ad entrare sarà il primo ad uscire.

[2] *Backlog* è, letteralmente, il cumulo di lavoro arretrato. È possibile che, dopo aver eseguito la accept(), il server debba eseguire altre operazioni e che, durante queste operazioni, sopraggiungano altre richieste prima che il server possa rieseguire la accept().