Correzioni LPR
e serializzazione di oggetti

S. Campa
LPRb A.A. 2007-2008

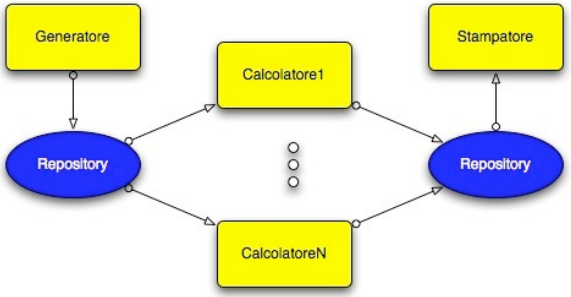
miniTalk

- Il programma principale pubblica tramite due thread un ServerSocket per la ricezione da socket e un Socket per l'invio su socket
- **GestoreInput**: si limita ad accettare la connessione e creare una classe che legge da socket e scrive a video
- **GestoreOutput**: tenta per 5 volte una connessione remota e crea una classe che legge tastiera e scrive sul socket
- Classe di gestione **Copiatore**: copia da inputstream ad outputstream
- Pro: immediato da progettare
- Cons: inefficiente, consuma risorse (doppia connessione unidirezionale)

instantMessanger

- La classe principale **Messanger** si propone come server o come client, in modo alternato
- La connessione, una volta stabilita, e' unica ma bidirezionale
- La gestione degli stream di input/output e' demandata ad una coppia di thread concorrenti (**CopyThread**)
- Molto piu' efficiente del precedente

threadPoolConcurrency



threadPoolConcurrency

- Soluzione fornita da java.util.concurrent
- La creazione/gestione dei thread esecutori e' affidata ad un oggetto **ExecutorService** che crea un thread pool di dimensione costante
- Ogni nuovo task (oggetto **Runnable**) creato viene assegnato da ES ad un thread disponibile tramite il metodo **execute**(<Runnable>)
- La terminazione e' affidata al metodo **shutdown()** dell'ES
- Il metodo **awaitTermination** blocca il flusso di esecuzione finche' tutti i task sono stati calcolati (o e' scattato un timeout)

threadPoolConcurrencyDue

- Come threadPoolConcurrency solo che il Repository e' realizzato tramite la classe **LinkedBlockingQueue** di java.util.concurrent
- Implementazione dell'interfaccia **BlockingQueue**
- I metodi **take()** e **put()** permettono di estrarre/inserire elementi
- Tutte le implementazioni di **BlockingQueue** sono thread-safe, ossia i metodi offrono funzionalita' atomiche.
- Dunque, LinkedBlockingQueue e' l'implementazione di una coda FIFO con operazioni di estrazione/inserimento *bloccanti*

Serializzazione (e deserializzazione)

- Lo stato di un oggetto viene trasformato in un flusso di bit (stream)
  - Per salvare il contenuto degli oggetti su disco (persistenza)
  - Per trasferire oggetti in deep-copy da un jvm ad un'altra (e.g. su rete)
- Deep-copy -> La copia riguarda lo stato di tutta la rete di oggetti raggiungibili dall'oggetto principale

Accortezze

- Una classe serializzabile deve implementare **java.lang.Serializable** (e trattare le opportune NotSerializableException)
- E' richiesto il versionamento delle classi (warning)
- Programmando, bisogna sempre tenere in mente che la deserializzazione si comporta come una copia:
  - allocazione di memoria per l'oggetto e inizializzazione a NULL
  - i campi delle classi serializzabili vengono ripristinati dallo stream
  - non c'e' relazione tra copia locale e copia remota!

Un esempio

- **ObjectInputStream** e **ObjectOutputStream** sono classi che permettono di deserializzare/serializzare oggetti “leggendo”/”scrivendo” sullo stream rappresentato da su oggetto ObjectOutputStream/ObjectInputStream
- Alla base di ObjectInputStream/ObjectOuputStream e' possibile avere diverse classi di InputStream (OutputStream) come da costruttore

public ObjectInputStream(InputStream in) throws IOException

public ObjectOutputStream(OutputStream in)throws IOException

- ... e dunque anche InputStream/OutputStream su socket!

Java API (ObjectInputStream)

- Usato per leggere (deserializzare) oggetti da uno stream

```
Socket s = ...;
ObjectInputStream ois;
ois = new ObjectInputStream(s.getInputStream());
```

- L'oggetto letto deve essere castato opportunamente

```
MyObject obj = (MyObject) ois.readObject();
obj.mymethod();
```

- Stringhe ed array sono considerati oggetti serializzabili

Java API (ObjectOutputStream)

- Usato per scrivere (serializzare) oggetti su uno stream

```
Socket s = ...;
ObjectOutputStream oos;
oos = new ObjectOutputStream(s.getOutputStream());
```

- La scrittura segue il metodo opportuno

```
oos.writeObject(new MyObject());
```

- L'oggetto verra' inviato alla readObject() corrispondente

Java API

- Inoltre, vi sono alcuni metodi pre-costuiti per tipi built-in:
  - readBoolean() / writeBoolean(boolean b)
  - readByte() / writeByte(byte b)
  - readChar() / writeChar (char c)
  - readFloat() / writeFloat (float f)
  - readDouble() / writeDouble (double d)
  - readInt() / writeInt (int v)
- ecc....

serialization

- Applicazione Client/Server che si scambiano un oggetto
- Il server pubblica un socket e resta in attesa di una connessione, riceve un oggetto **Richiesta**, ne modifica lo stato e lo rimanda indietro
- Il client si connette al server, invia un oggetto di tipo **Richiesta** e attende la versione modificata
- **Richiesta** implementa **Serializable**
- La versione inviata e quella ricevuta non hanno lo stesso stato!

Domande?