Lezione n.3
LPR-Informatica Applicata
sockets UDP

27/2/2006

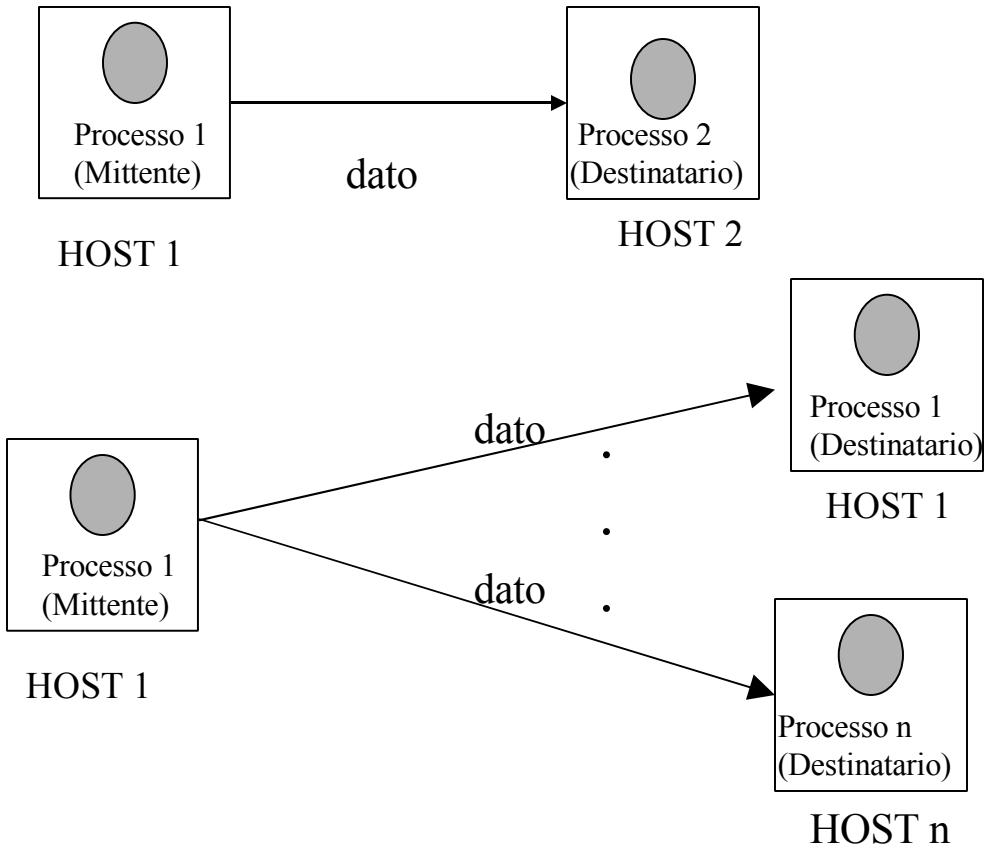
Laura Ricci

SCHEMA DELLA PRESENTAZIONE

- Meccanismi di comunicazione interprocess (IPC)
- Rappresentazione dei dati trasmessi sulla rete
- Sockets
- JAVA:Le classi DatagramSocket e DatagramPacket

MECCANISMI DI COMUNICAZIONE TRA PROCESSI

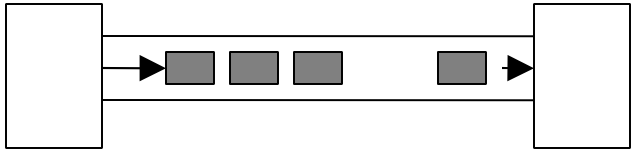
Meccanismi di comunicazione tra processi (IPC): studiare Liu, Capitolo 2



TIPI DI COMUNICAZIONE: CONNECTION ORIENTED VS. CONNECTIONLESS

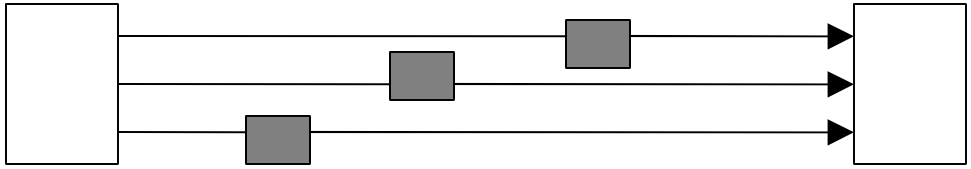
Comunicazione *Connection Oriented*. Prevede le seguenti fasi:

- creazione di una *connessione* tra mittente e destinatario
- invio dei dati sulla connessione
- chiusura della connessione



Comunicazione *Connectionless*.

- Non si stabilisce alcuna connessione
- Mittente e destinatario comunicano mediante lo scambio di pacchetti



TIPI DI COMUNICAZIONE: CONNECTION ORIENTED VS. CONNECTIONLESS.

Connection oriented vs. connectionless: differenze

- *Indirizzamento:*
  - ***Connection Oriented:*** l'indirizzo del destinatario è specificato al momento della connessione
  - ***Connectionless:*** l'indirizzo del destinatario viene specificato in ogni pacchetto (per ogni send)
- *Ordinamento dei dati scambiati:*
  - ***Connection Oriented:*** ordinamento dei messaggi garantito
  - ***Connectionless:*** nessuna garanzia sull'ordinamento dei messaggi
- *Utilizzo:*
  - ***Connection Oriented:*** Grossi streams di dati
  - ***Connectionless:*** invio di un numero limitato di dati

TIPI DI COMUNICAZIONE: CONNECTION ORIENTED VS. CONNECTIONLESS

Protocollo UDP = connectionless, trasmette pacchetti dati = *Datagrams*

- ogni datagram deve contenere l'indirizzo del destinatario
- datagrams spediti dallo stesso processo possono seguire percorsi diversi ed arrivare al destinatario in *ordine diverso* rispetto all'ordine di spedizione

Protocollo TCP = trasmissione connection-oriented o stream Oriented

- viene stabilita una connessione tra mittente e destinatario
- su questa connesione si spedisce una sequenza di dati = stream di dati (per modellare questo tipo di comunicazione in JAVA si possono sfruttare i diversi tipi di stream definiti dal linguaggio)

IPC: MECCANISMI BASE

Una API per la comunicazione tra processi deve garantire almeno le sguenti funzionalità

- *Send* per trasmettere un dato al processo destinatario
- Receive per ricevere un dato dal processo mittente
- *Connect* (solo per comunicazione connection oriented) per stabilire una connessione logica tra mittente e destinatario
- *Disconnect* per eliminare una connessione logica

Possono esistere diversi tipi di send/receive (sincrona/asincrona, simmetrica/asimmetrica

IPC: MECCANISMI BASE

Un esempio: HTTP (1.0)

- Il processo che esegue il Web browser esegue una connect per stabilire una connessione con il processo che esegue il Web Server
- Il Web browser esegue una send, per trasmettere una richiesta al Web Server (operazione GET)
- Il Web server esegue una receive per ricevere la richiesta dal Web Browser, quindi a sua volta esegue una send per inviare la risposta
- I due processi eseguono una disconnect per terminare la connessione

HTTP 1.1: Più richieste su una connessione (più send e receive).

IPC: MECCANISMI BASE

Comunicazione sincrona (o bloccante): il processo che esegue la send o la receive si *sospende* fino al momento in cui la comunicazione è completata.

send sincrona = completata quando i dati spediti sono stati ricevuti dal destinatario (è stato ricevuto un ack da parte del destinatario)

receive sincrona =completata quando i dati richiesti sono stati ricevuti

send asincrona (non bloccante) = il destinatario invia i dati e prosegue la sua esecuzione senza attendere un ack dal destinatario

receive asincrona = il destinatario non si blocca se i dati non sono arrivati. Possibile diverse implementazioni

IPC: MECCANISMI BASE

Receive Asincrona.

- se il dato richiesto è arrivato, viene reso disponibile al processo che ha eseguito la receive

- se il dato richiesto non è arrivato:

  - il destinatario esegue nuovamente la receive, dopo un certo intervallo di tempo (*polling)*

  - il supporto a tempo di esecuzione notifica al destinatario l'arrivo del dato (richiesta l'attivazione di un *event listener*)

IPC: MECCANISMI BASE

Comunicazione sincrona: per non bloccarsi indefinitamente

- *Timeout* – meccanismo che consente di bloccarsi per un intervallo di tempo prestabilito, poi di proseguire comunque l'esecuzione

- *Threads* – l'operazione sincrona può essere effettuate in un thread. Se il thread si blocca su una send/receive sincrona, l'applicazione può eseguire altri thread.

  Nel caso di receive sincrona, gli altri threads non devono ovviamente richiedere per l'esecuzione il valore restituito dalla receive

RAPPRESENTAZIONE DEI DATI

Caratteristica principale di un ambiente distribuito: gli hosts su cui sono in esecuzione i processi sono *eterogenei* ⇒ possono utilizzare rappresentazioni diverse per lo stesso tipo di dato.

Esempio1: l'host A (mittente) è una macchina a 64 bits che utilizza la rappresentazione *big –endian.* L'host B (destinatario) è una macchina a 32 bits, che utilizza la rappresentazione *little-endian.*

⇒

E' necessario effettuare un troncamento del valore spedito da A + scambiare l'ordine dei bytes, quando il dato viene memorizzato da B.

Esempio2: l'host A spedisce un carattere all'host B. A utilizza la rappresentazione ASCII, mentre B utilizza l' *unicode*

necessarie procedure di conversione (effettuate dal mittente o dal destinatario) oppure la definizione di una *notazione esterna* (esempio ASNN, XDR,XML)

RAPPRESENTAZIONE DEI DATI: MARSHALLING

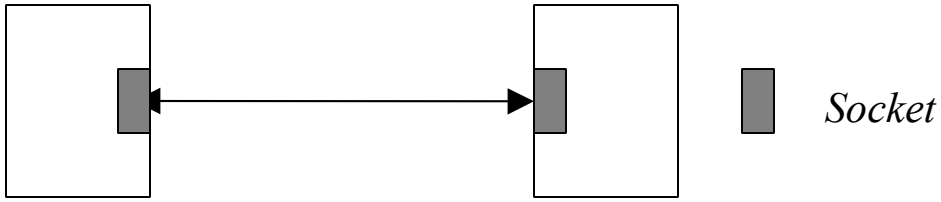
- Invio di strutture dati (esempio liste, …) richiede :
  - Il mittente deve effettuare il flattening (*serializzazione*) delle strutture dati (eliminazione dei puntatori)
  - Il destinatario deve ricostruire la struttura dati nella sua memoria
- *Data Marshalling:* procedimento necessario per spedire valori e strutture dati
  - serializzazione della struttura dati
  - conversione dei valori ad una rappresentazione esterna
- In JAVA : marshalling degli oggetti.

JAVA IPC: LA SOCKET API

Socket = presa di corrente

Termine utilizzato in tempi remoti in *telefonia.* La connessione tra due utenti veniva stabilita tramite un operatore che inseriva fisicamente i due estremi di un cavo in due ricettacoli (*sockets*), ognuno dei quali era assegnato ai due utenti.

Socket è una *astrazione* che indica una "*presa*" a cui un processo si può collegare per spedire dati sulla rete. Al momento della creazione un socket viene collegato ad una porta.



JAVA IPC: LA SOCKET API

Studiare capitolo 13 Elliotte Rusty Harold
Socket Application Program Interface = Definisce un insieme di meccanismi che supportano la comunicazione di processi in ambiente distribuito.

- JAVA socket API: definisce interfacce diverse per UDP e TCP
  - UDP = Datagram Sockets
  - TCP = Stream Sockets
- Datagram Sockets.

  Vengono utilizzate le classi
  - *DatagramPacket:* gestione dei pacchetti spediti su UDP
  - *DatagramSocket:* creazione di sockets, spedizione di pacchetti tramite il socket creato, etc…

JAVA : COMUNICAZIONE UDP

Trasmissione di pacchetti UDP:

- mittente e destinatario devono creare i sockets attraverso i quali avviene la comunicazione.
- *Ipotesi:* il mittente collega il suo socket ad una porta *PM*, il destinatario collega il suo socket ad una porta *PD*

Invio di pacchetti UDP, operazioni eseguite dal *mittente*

- creazione di un datagram socket *SM* collegato a *PM*
- creazione del *pacchetto DP (datagram).*
- invio del pacchetto *DP* sul socket *SM*

Ogni pacchetto UDP spedito dal mittente deve contenere:

- indirizzo del destinatario= indirizzo IP su cui è in esecuzione il destinatario + porta PD
- Riferimento ad un *vettore di bytes* che contiene il valore del messaggio che deve essere spedito.

JAVA : COMUNICAZIONE UDP

Ricezione di pacchetti DP, operazioni svolte dal *destinatario*

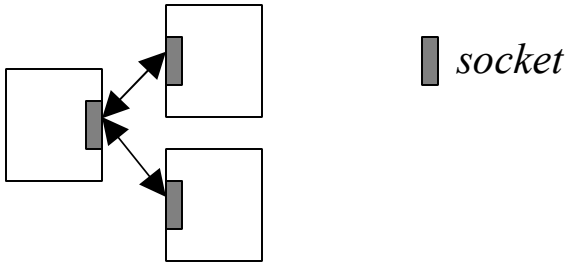
- creazione di un datagram socket *SD* collegato a *PD*
- creazione di una struttura adatta a memorizzare il pacchetto ricevuto
- ricezione di un pacchetto dal *socket SD* e sua memorizzazione nella struttura

I dati inviati mediante UDP devono essere rappresentati come vettori di bytes, ma JAVA offre diversi tipi di filtri per spedire dati di tipo primitivo e dati strutturati (oggetti)

JAVA : COMUNICAZIONE UDP

Caratteristiche dei sockets UDP

- il destinatario deve "*pubblicare*" la porta a cui è collegato il socket di ricezione, affinchè il mittente possa spedire pacchetti su quella porta
- non è in genere necessario pubblicare la porta a cui collegato il socket del mittente
- un processo può utilizzare lo stesso socket per spedire pacchetti verso destinatari diversi
- Processi diversi possono spedire pacchetti sullo stesso socket allocato da un processo destinatario



JAVA : LA CLASSE DATAGRAM SOCKET

Public class DatagramSocket ***extends Object***
Costruttori:

public DatagramSocket ( ) *throws SocketException*

- crea un socket e lo collega ad una porta *anonima* (o *effimera),* il sistema sceglie una porta *non utilizzata* e la assegna al socket. Per reperire la porta allocata utilizzare il metodo *getLocalPort().*

- utilizzato generalmente da chi inizia la trasmissione.

- *Esempio:* un client si connette ad un server mediante un socket collegato ad una porta anonima. Il server invia la risposta sullo stesso socket, ⇒ preleva l'indirizzo del mittente (IP+porta) dal pacchetto ricevuto. Quando il client termina la porta viene utilizzata per altre connessioni.

JAVA : LA CLASSE DATAGRAM SOCKET

Public class DatagramSocket ***extends Object***
Costruttori:

public DatagramSocket (***int*** port) ***throws SocketException***

- crea un socket su una porta specificata .
- viene sollevata un'eccezione quando la porta è già utilizzata, oppure se si tenta di connettere il socket ad una porta su cui non si hanno diritti.
- utilizzato da chi attende una comunicazione.
- *Esempio*: il server crea un socket collegato ad una porta resa nota ai client. Di solito la porta viene allocata permanentemente a quel servizio (porta non effimera)

INDIVIDUAZIONE PORTE LIBERE

Un programma per *individuare le porte libere* su un host:

```
import java.io.*;
import java.net.*;
public class porte-udp
 {public static void main(String args[ ])
        { for (int i=1; i<65535; i++)
          {try {
                  DatagramSocket s =new DatagramSocket(i);
                  System.out.println ("Porta libera"+i);
                }
      catch (BindException e) {System.out.println (e + "porta già in uso") ;}
      catch (Exception e) {System.out.println (e);}
        }
 }
```

JAVA : LA CLASSE DATAGRAMPACKET

public final class DatagramPacket extends Object

Costruttori:

public DatagramPacket(byte [ ] data, ***int*** length, ***InetAddress*** destination, ***int*** port)

- utilizzato dal mittente
- il messaggio deve essere trasformato in una *sequenza di bytes* e memorizzato nel vettore data (strumenti necessari per la traduzione, es: metodo *getBytes*( ), la classe java.io.ByteArrayOutputStream)
- length indica il numero di bytes da prelevare dal vettore data per costruire il pacchetto
- il pacchetto contiene un riferimento al vettore data: posso modificare i dati da spedire dopo la creazione del pacchetto
- Destination+port individuano il destinatario

JAVA : LA CLASSE DATAGRAMPACKET

public final class DatagramPacket extends Object

Costruttori:

public DatagramPacket(*byte*[ ] buffer, ***int*** length)

- utilizzato dal destinatario

- definisce la struttura che deve essere utilizzata per memorizzare il pacchetto ricevuto. Il payload del pacchetto (la parte che contiene i dati) viene copiata in buffer. Si copiano al massimo length bytes del pacchetto.

  *Esempio:*byte [] buffer = *new byte*[8192]
  DatagramPacket dp = *new DatagramPacket* (buffer, buffer.length);

JAVA : INVIARE E RICEVERE PACCHETTI

Invio di pacchetti

- Sock.*send* (dp)
- dove: *sock* è il socket attraverso il quale voglio spedire il pacchetto *dp*

Ricezione di pacchetti

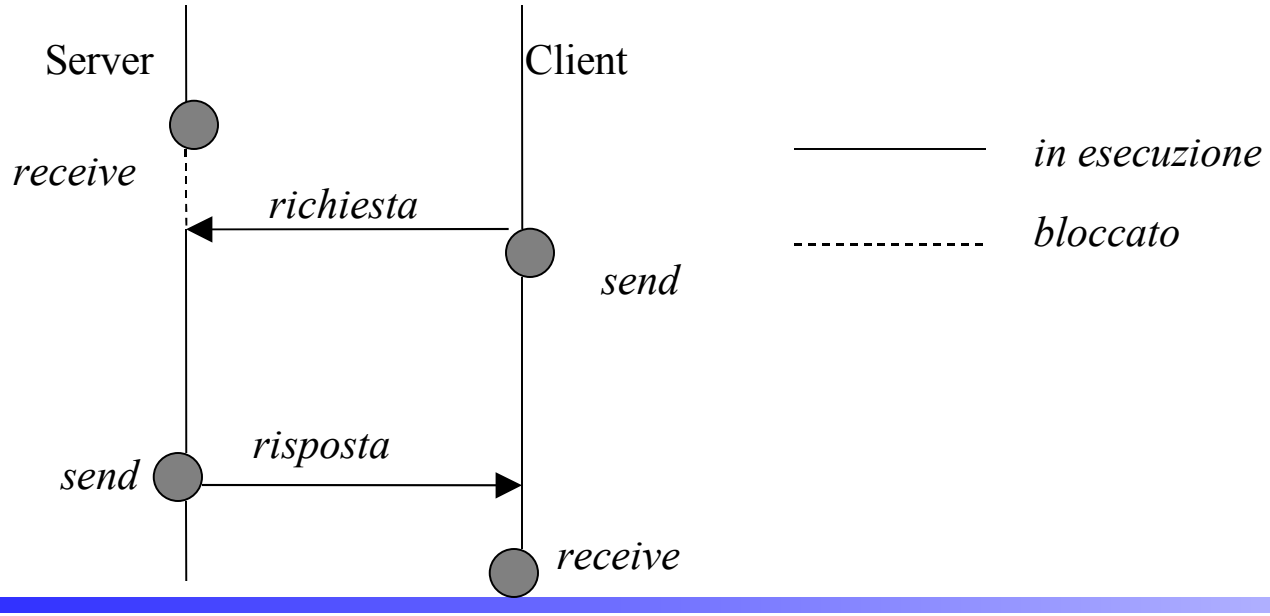
- sock.*receive*(buffer)
- Dove sock è il socket attraverso il quale ricevo il pacchetto e buffer è la struttura in cui memorizzo il pacchetto ricevuto

COMUNICAZIONE TRAMITE SOCKETS: CARATTERISTICHE

La send è non *bloccante* = il processo che esegue la send prosegue la sua esecuzione, senza attendere che il destinatario abbia ricevuto il pacchetto

receive *bloccante* = il processo che esegue la receive si blocca fino al momento in cui viene ricevuto un pacchetto.

per evitare attese indefinite è possibile associare *al socket un timeout.* Quando il timeout scade, viene sollevata una *InterruptedIOException*



PER ESEGUIRE IL PROGRAMMA SU UN UNICO HOST

- Attivare il client ed il server in due diverse shell
- Se l'host è connesso in rete: utilizzare come indirizzo IP del mittente/destinatario l'indirizzo dell'host su cui sono in esecuzione i due processi (reperibile con getLocalHost())
- Se l'host non è connesso in rete utilizzare l'indirizzo di *loopback*
- Tenere presente che mittente e destinatario sono in esecuzione sulla stessa macchina ⇒ devono utilizzare porte diverse
- Mandare in esecuzione per primo il server, poi il client

ESERCIZIO

Svolgere l'esercizio n.5, pag 128 del testo Liu, Distributed Computing.

Sintesi dell'esercizio:

Scrivere un applicazione composta da un processo Sender ed un processo Receiver, in esecuzione su hosts diversi. Il Sender riceve a linea di comando una stringa, l'indirizzo del receiver (indirizzo IP+porta) ed invia al Receiver la stringa. Il Receiver riceve il messaggio e lo visualizza.

Considerare poi i seguenti punti

- Cosa accade se mando in esecuzione prima il Receveir, poi il Sender?
- E se mando in esecuzione prima il Sender, poi il receiver?
- Nel processo Receiver, aggiungere un time-out sulla receive, in modo che la receive non si bocchi per più di 5 secondi. Cosa accade se attivo il receiver, ma non il sender?

ESERCIZIO

- Modificare il codice del Sender in modo che esso usi lo stesso socket per inviare lo stesso messaggio a due diversi Receivers. Mandare in esecuzione prima i due Receivers, poi il Sender. Cosa accade?
- Modificare il codice del Sender in modo che esso usi due sockets diversi per inviare lo stesso messaggio a due diversi Receivers.Mandare in esecuzione prima i due Receivers, poi il Sender. Cosa accade?
- Modificare il codice ottenuto al passo precedente in modo che il Sender invii una sequenza di diversi messaggi ai Receivers. Ogni messaggio contiene il valore della sua posizione nella sequenza. Il Sender si sospende per 3 secondi tra un invio ed il successivi. Ogni receiver deve essere modificato in modo che esso esegua la receive in un ciclo infinito. Cosa accade?
- Modificare il codice ottenuto al passo precedente in modo che il Sender non si sospenda tra un invio e l'altro. Cosa accade?
- Modificare il codice iniziale in modo che il Receiver invii al Sender un ack quando riceve il messaggio. Il Sender visualizza l'ack ricevuto.