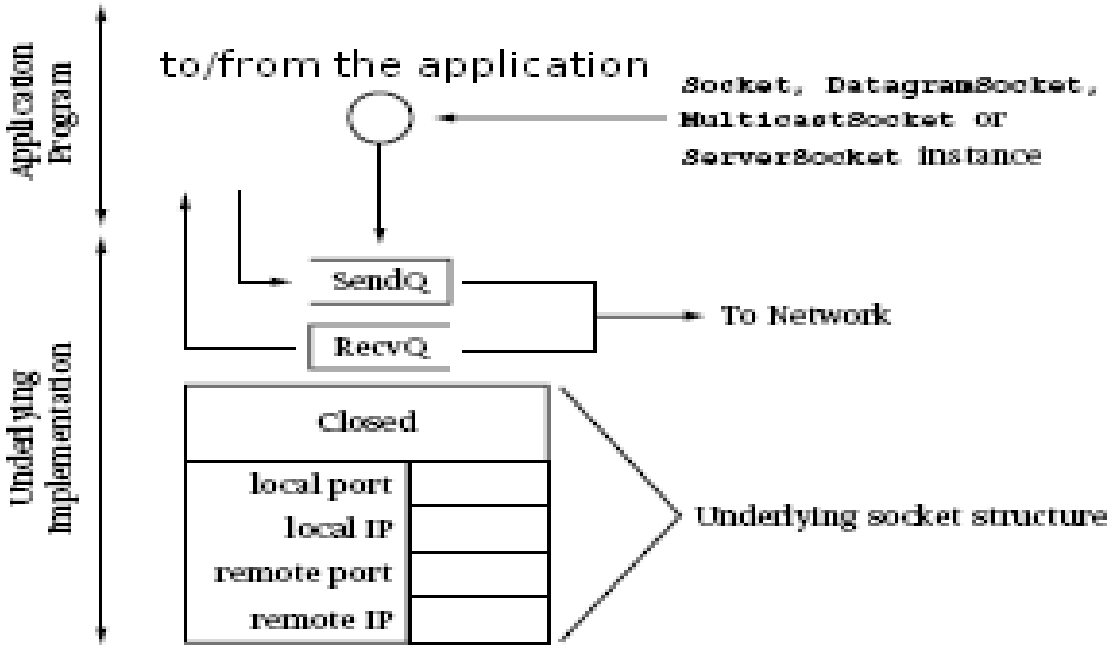
Lezione n.8
LPR Informatica Applicata
Interazione
JVM- TCP Layer

27/04/2009

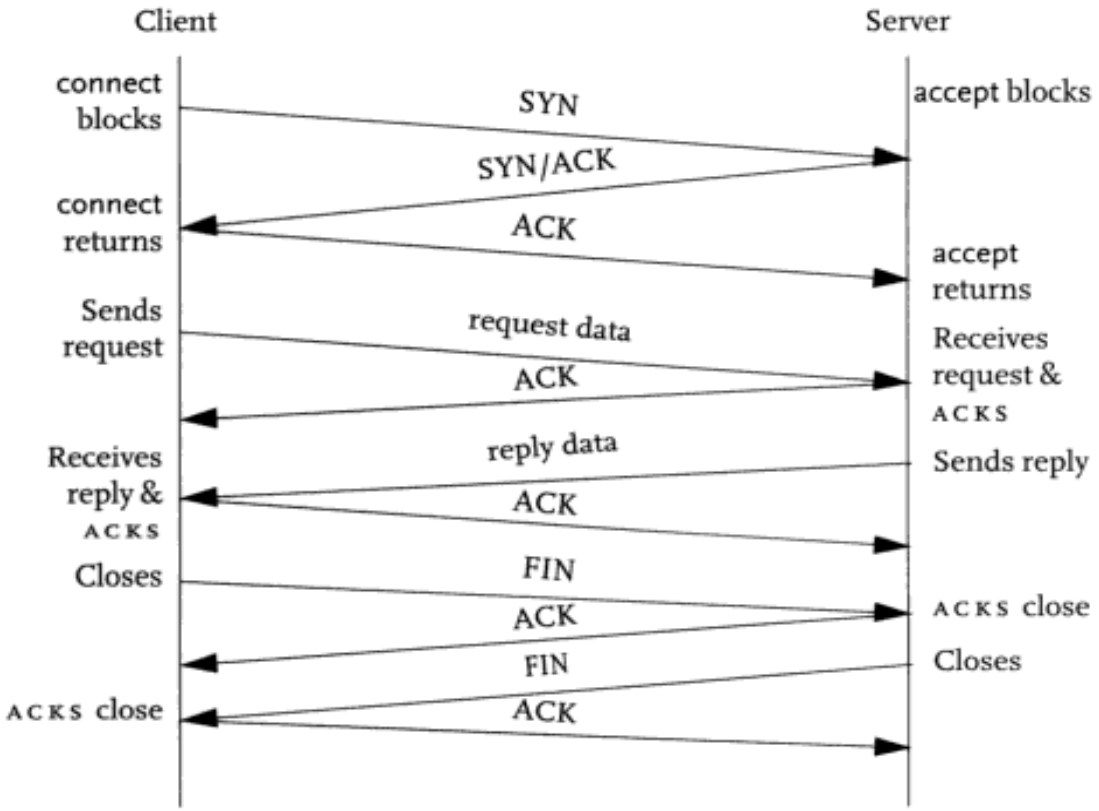
Laura Ricci

STRUTTURA GENERALE DI UN SOCKET



- remote port ed host significative solo per socket TCP
- SendQ, RecQ: buffer di invio/ricezione
- ogni socket è caratterizzato da informazioni sul suo stato (ad esempio closed). Lo stato del socket è visibile tramite il comando netstat

TCP: GESTIONE DELLE CONNESSIONI



TCP:GESTIONE DELLE CONNESSIONI

- Apertura della connessione implementata mediante il three way handshake:
  - Il client invia un SYN
  - Il server risponde con un SYN/ACK
  - Il client risponde con un ACK
- Se non si riceve una risposta al primo SYN, il SYN viene inviato nuovamente un certo numero di volte, incrementando via via l'intervallo di tempo tra un tentativo ed il successivo
- La prima ritrasmissione dopo 3-6 secondi, di solito viene raddoppiato l'intervallo di tempo per ogni ritrasmissione (generalmente massimo 3-4 ritrasmissioni)
- La chiusura di una connessione di solito comporta lo scambio di quattro pacchetti (FIN-ACK). Il socket risulta chiuso quando entrambe i partner hanno eseguito la close

TCP: BACKLOG QUEUE

- Supponiamo di aver sviluppato un server TCP in JAVA. La JVM interagisce con il livello TCP per l'esecuzione dei comandi che riguardano la rete

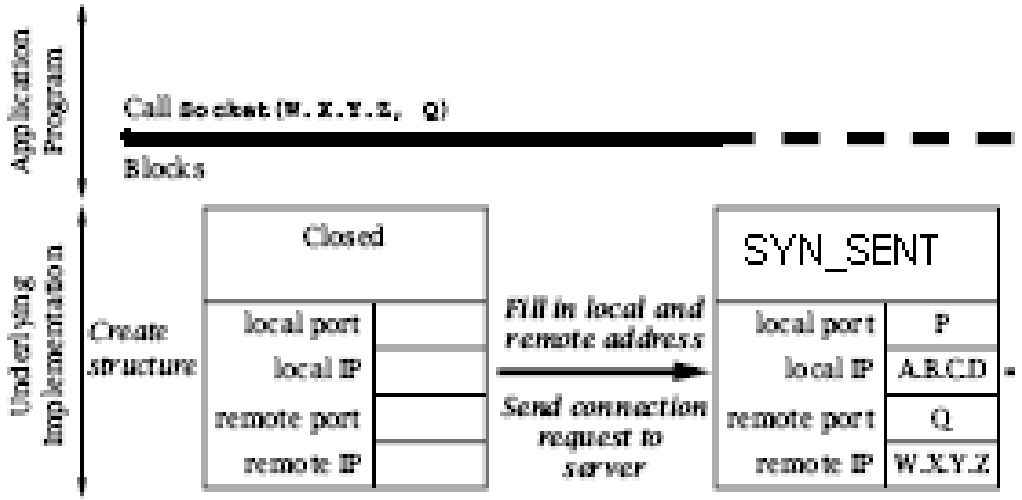
```
ServerSocket (int port, int backlog) throws IOException
```

- Backlog Queue:
  - contiene le richieste che sono state accettate dal supporto TCP, ma non sono ancora state accettate dalla applicazione
  - è una coda esistente tra il livello TCP e l'applicazione che ha creato il ServerSocket (Listening Socket)
  - Il parametro backlog specifica la lunghezza massima della coda. Il valore di default=50
  - Il valore specificato può essere ridotto dal supporto sottostante conoscere. Non esistono metodi per conoscere il valore effettivo della lunghezza della coda

TCP: BACKLOG QUEUE

- la accept( ) preleva le richieste di connessione dalla backlog queue e restituisce il socket dedicato a quella connessione
- Se la coda non è piena, al momento della ricezione della richiesta di connessione,
  - il TCP completa il there-way-handshake e aggiunge un riferimento alla connessione nella backlog queue
  - il client, a questo punto, risulta connesso
  - Il server, non ha ancora creato il socket relativo a quella connessione, che verrà creato solo al completamento della accept()
- altrimenti
  - il livello TCP lato server non invia il SYN/ACK al client.
  - Il client effettua ulteriori tentativi di stabilire la connessione dopo un certo intervallo di tempo

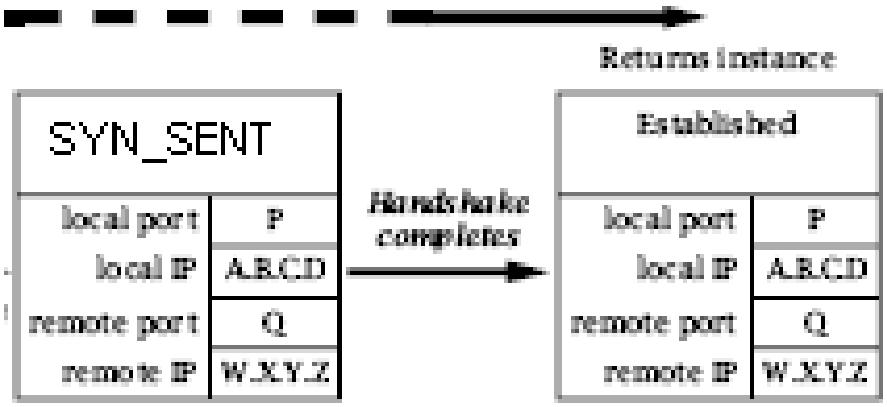
CONNESSIONE LATO CLIENT: STATO DEL SOCKET



Quando il client invoca il costruttore Socket( ).

- lo stato iniziale del socket viene impostato a Closed, la porta (P) e l'indirizzo locale (A.B.C.D) sono impostate dal supporto
- dopo aver inviato il messaggio iniziale di handshake, lo stato del socket passa a SYN_SENT (inviato segmento SYN)
- il client rimane bloccato fino a che il server riscontra il messaggio di handshake mediante un ack

CONNESSIONE LATO CLIENT: STATO DEL SOCKET



- il messaggio di handshake può venire trasmesso più volte
  il client può rimanere bloccato per un lungo periodo.

il costruttore può sollevare una eccezione se,

- non esiste il servizio richiesto sulla porta selezionata
- il messaggio di handshake non viene riscontrato entro un certo intervallo di tempo(timeout)

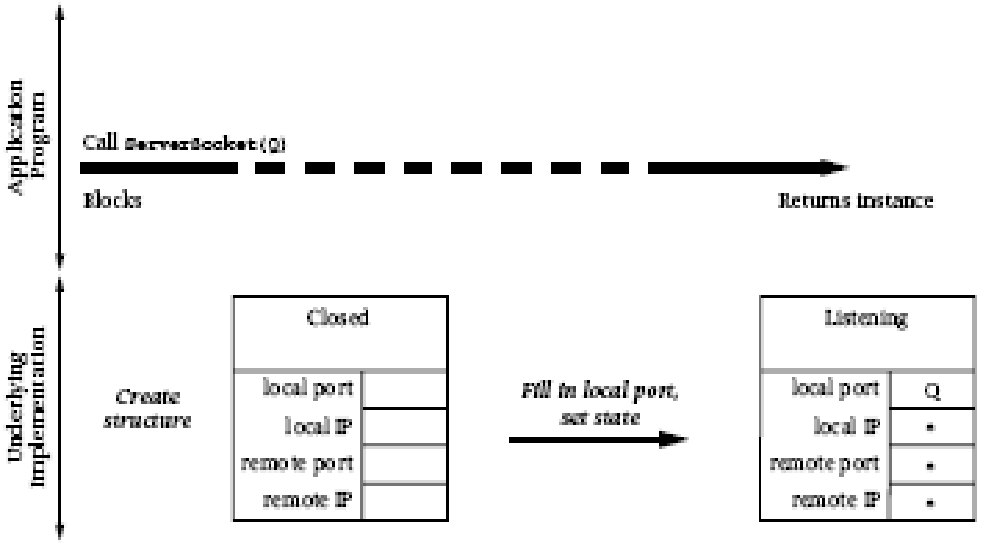
TCP: STATO DELLA CONNESSIONE LATO CLIENT

- Lo stato della connessione associata ad una porta TCP può essere ''osservato'' mediante il comando netstat
- Stati possibili lato client
  - CLOSED Il socket non è ancora stato inizializzato
  - SYN-SENT il client ha iniziato il three-way-handshake ed ha inviato il SYN, ma il server non ha ancora accettato la connessione
  - Si può rilevare questo stato mediante il metodo

    boolean IsConnected( )

    Che restituisce in questo caso il valore false
  - ESTABLISHED il client ha effettuato la connessione . Il metodo isConnected() restituisce il valore true, il valore di isOutputShutdown è false

CONNESSIONE LATO SERVER: STATO DEL SOCKET



Il Server crea un server socket sulla porta P

- se non viene specificato alcun indirizzo IP (wildcard = *), il server può ricevere connessioni da una qualsiasi delle sue interfacce
- lo stato del socket viene posto a Listening: questo indica che il server sta attendendo connessioni da una qualsiasi interfaccia, sulla porta P

CONNESSIONE LATO SERVER: STATO DEL SOCKET

- il server si sospende sul metodo accept( ) in attesa di una nuova connessione
- quando riceve una richiesta di connessione dal client, crea una nuova struttura che implementa il nuovo socket creato. In tale struttura
  - indirizzo e porta remoti vengono inizializzati con l'indirizzo IP e la porta ricevuti dal client che ha richiesto la connessione
  - L'indirizzo locale viene settato con l'indirizzo dell'interfaccia da cui è stata ricevuta la connessione.
  - La porta locale viene inizializzata con quella a cui associata al serversocket
  - Lo stato del nuovo socket è SYN_RCVD

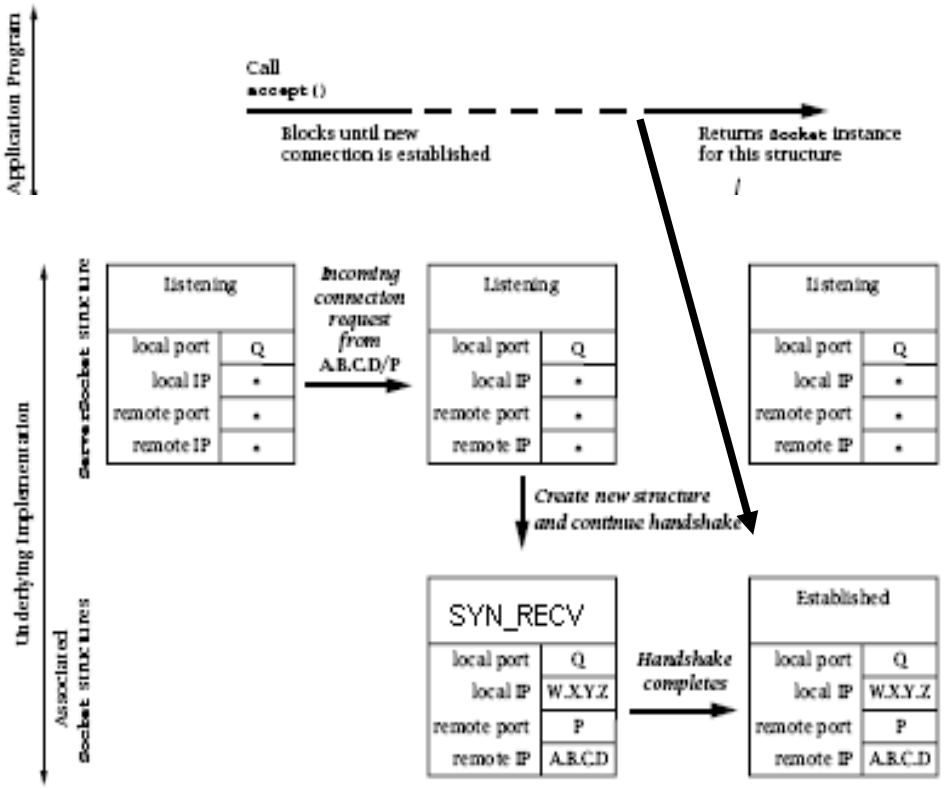
    è stato inviato il SYN/ACK al client e che si sta attendendo l'ACK dal client, per terminare il 3-way handshake

CONNESSIONE LATO SERVER: STATO DEL SOCKET

Dopo aver creato il nuovo socket, il server

- riscontra il SYN inviato dal client mediante un ack
- quando riceve, a sua volta, il riscontro dal client (terzo messaggio del 3-way handshake)
  - imposta lo stato del socket ad ESTABLISHED
  - inserisce il socket creato in una lista di socket associata al ServerSocket da cui è stata ricevuta la richiesta di connessione
  - solo a questo punto, l'esecuzione del metodo accept( ) termina e restituisce un puntatore alla struttura creata
- Anche il client imposta lo stato del proprio socket ad ESTABLISHED dopo aver terminato il 3-way handshake

CREAZIONE DI CONNESSIONI LATO SERVER



DEMULTIPLEXING DEI SEGMENTI TCP

- Tutti i sockets associati allo stesso ServerSocket 'ereditano' da esso
  - la porta di ascolto
  - l' indirizzo IP da cui è stata ricevuta la richiesta di connessione
- Questo implica che sullo stesso host possano esistere più sockets associati allo stesso indirizzo IP ed alla stessa porta locale (il Serversocket e tutti i sockets associati,.....)
- Meccanismo di demultiplexing: utilizzato per decidere a quale socket è destinato un segmento TCP ricevuto su quella interfaccia e su quella porta
- La conoscenza dell'indirizzo e porta destinazione non risulta più sufficiente per individuare il socket a cui è destinato il segmento

DEMULTIPLEXING DEI SEGMENTI TCP

Definizione del meccanismo di demultiplexing:

- La porta locale riferita nel socket deve coincidere con quella contenuta nel segmento TCP
- Ogni campo del socket contenente una wildcard (*), può essere messo in corrispondenza con qualsiasi valore corrispondente contenuto nel segmento
- Se esiste più di un socket che corrisponde al segmento in input , viene scelto il socket

  che contiene il minor numero di wildcards.

- in questo modo un segmento spedito dal client viene ricevuto sul socket S associato alla connessione con quel client, piuttosto che sul serversocket perchè S risulta 'più specifico' per quelsegmento

TCP: STATO DELLA CONNESSIONE LATO SERVER

- Lo stato della connessione associata ad una porta TCP può essere ''osservato'' mediante il comando netstat
- Stati possibili per il Server
  - LISTEN: corrisponde ad un ServerSocket in attesa di connessioni
  - SYN-RECEIVED: TCP ha accettato una connessione e ne ha inserito un riferimento nella backlog queue. Si attende l'ACK dal client, dopo cui è possibile costruire il socket associato a quella connessione e che verrà restituito poi dalla l'accept( )

CHIUSURA DI SOCKETS TCP

- Metodi per la chiusura di un socket: close, shutdownOutput, shutdownInput
- In generale, il mittente M , dopo aver inviato tutti i dati al destinatario D, chiude il socket mediante
  - close, se M non attende risposte dal destinatario D
  - shutdownOutput, in caso contrario
- Supponiamo che M invochi shutdownOutput/close,
  - invio di tutti i dati bufferizzati e relativo riscontro (a meno di impostazioni diverse della proprietà linger())
  - successivamente si invia il segmento di FIN, che deve essere riscontrato dal destinatario D
  - l'attesa che D chiuda, a sua volta, il socket e ne segnali la chiusura ad M, mediante il corrispondente segmento di FIN
  - L'invio da parte di M a D del riscontro del segmento di FIN ricevuto

CHIUSURA DI SOCKET TCP

- Ipotesi Semplificativa:
  - uno dei due partner della connessione completa l'handshake per la chiusura del socket, prima che l'altro parte inizi la stessa procedura
  - Linger= false, il metodo close/shutdownOutput ritorna immediatamente il controllo al chiamante e il protocollo di chiusura viene eseguito in background
- Protocollo di chiusura del socket: M invoca una close/shutdownOutput( )
  - i dati presenti nel send buffer di M vengono inviati al destinatario
  - il supporto in esecuzione su M invia quindi il segmento di FIN e lo stato del socket passa a FIN_WAIT1
  - quando il supporto del destinatario riceve il messaggio di FIN, trasforma la segnalazione di chiusura ricevuta in un messaggio di fine sequenza (valore = -1) da inviare all'applicazione (continua pag.succ.....)

CHIUSURA DI SOCKET TCP

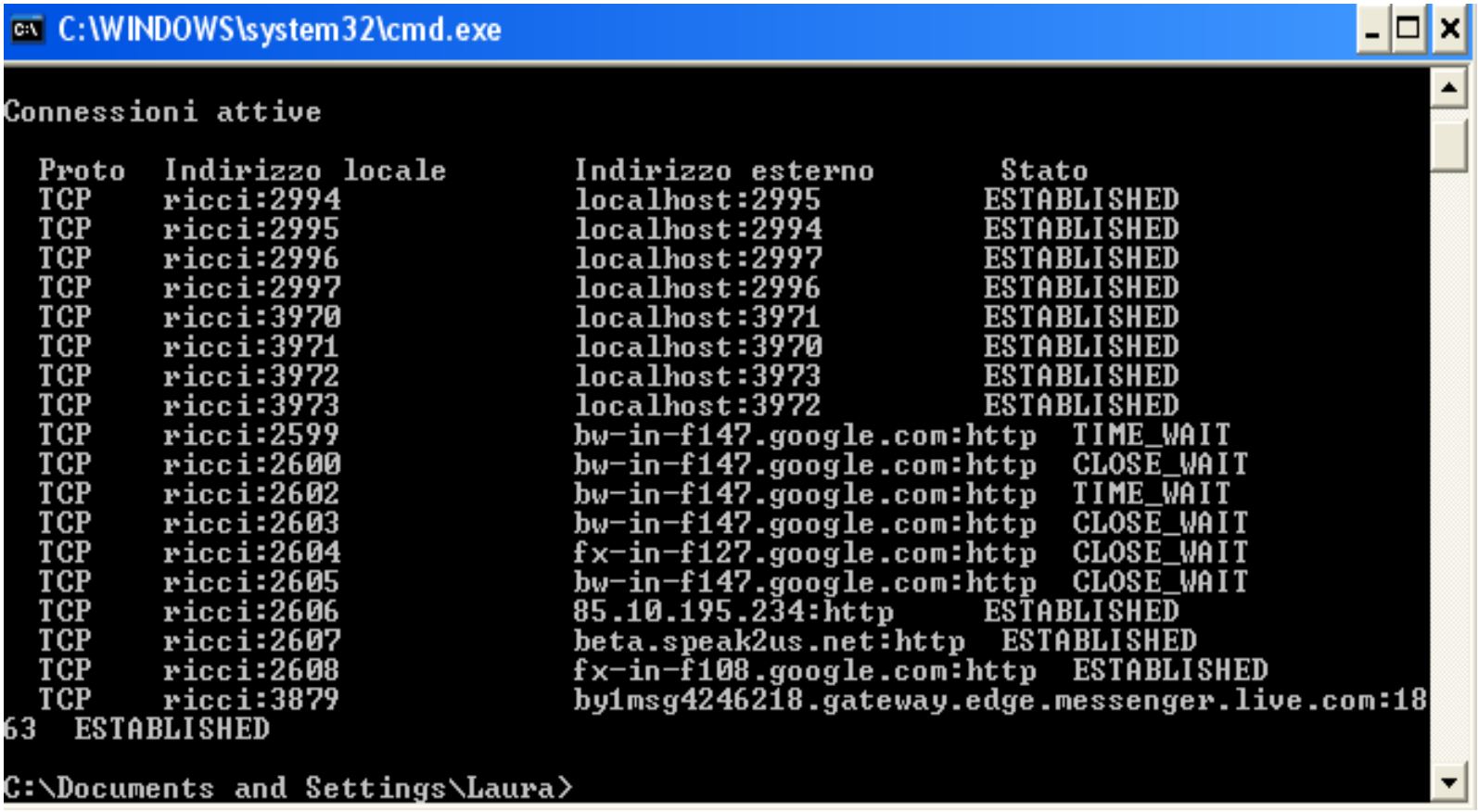
Protocollo di chiusura del socket: M invoca una close( )/shutdownOutput( )

- Quando M riceve da D il riscontro del FIN inviato (ricezione dell'ack), il socket passa nello stato di half closed (FIN_WAIT2).
  - Se D fallisce prima di aver completato la procedura di chiusura, il socket può rimanere indefinitamente in questo stato
- Il socket nel destinatario D passa a questo punto in uno stato di CLOSE_WAIT, in attesa che l'applicazione in esecuzione su D chiuda a sua volta il socket
- Quando l'applicazione chiude il socket., il supporto invierà a M il segmento di FIN
- A questo punto anche se la connessione risulta completamente chiusa, il socket, passa nello stato di TIME_WAIT

TCP: STATO DELLA CONNESSIONI

- FIN-WAIT1 – Il Socket è stato chiuso dalla applicazione locale, ma non è ancora stato ricevuto l'ACK alla richiesta di chiusura da parte della applicazione remota
- FIN-WAIT-2 - Il socket è stato chiuso dalla applicazione locale ed è stato ricevuto l'ACK dalla applicazione remota. L'applicazione remota però non ha ancora a sua volta chiuso il socket corrispondente
- CLOSE-WAIT - Il socket è stato chisuo dalla applicazione remota, ma non da quella locale. Il socket associato all'altro estremo della applicazione è quindi in stato FIN-WAIT1 o FIN-WAIT2
- TIME-WAIT Il socket è stato chiuso sia dalla applicaqzione locale che da quella remota. Le strutture dati viene mantenuta per un certo intervallo di tempo per essere sicuri che l'applicazione remota abbia ricevuto l'ack()

NETSTAT: ANALISI DELLO STATO DEI SOCKET



TCP: SHUTDOWNOUTPUT

Se si esegue una shutDownOuput( )

- localmente il socket rimane aperto solo in input
- una eventuale successiva write sullo stream di output da parte della applicazione locale genera una IOException( )
- il metodo viene può essere utilizzato per inviare un EOF sulla connessione in modo da segnalare la terminazione dello stream
- se l'host remoto effettua una read() su uno stream che è stato chiuso mediante una shutdownOuput( ), all'altro capo della comunicazione, a seconda del metodo invocato, la read può
  - restituire un numero di caratteri letti = -1
  - sollevare una EOF Exception( )

SHUTDOWNOUTPUT: ESEMPI DI UTILIZZO

```
import java.net.*;
import java.io.*;
public class closer
    {
    public static void main(String args[])throws Exception
        {
        InetAddress ia = InetAddress.getByName("localhost");
        Socket out =new Socket(ia,2600);
        OutputStream outs= out.getOutputStream();
        byte [] msg = new byte[100];
        msg= "ciaomondo".getBytes();
        outs.write(msg);
        out.shutdownOutput();
        }}
```

SHUTDOWNOUTPUT: ESEMPI DI UTILIZZO

```
import java.net.*;
import java.io.*;
public class server  {
public static void main (String args[]) throws Exception
    {ServerSocket ss = new ServerSocket(2600);
    Socket s=ss.accept();
    InputStream is = s.getInputStream();
    boolean go=true; int x;
    byte msg[]= new byte[100];
    do
    { x=is.read(msg);
      if (x!=-1){ String str=new String(msg);
                  System.out.println(x+str);}
    }
   while (x!=-1);   } }
```

SHUTDOWNOUTPUT: ESEMPI DI UTILIZZO

```
import java.net.*; import java.io.*; import java.util.*;
public class closer
{
    public static void main(String args[])throws Exception{
        InetAddress ia = InetAddress.getByName("localhost");
        Socket out =new Socket(ia,2500);
        OutputStream outs= out.getOutputStream();
        ObjectOutputStream oos = new  ObjectOutputStream(outs);
        Date d= new Date();
        oos.writeObject(d);
        out.shutdownOutput();
        }
    }
```

SHUTDOWNOUTPUT:ESEMPI DI UTILIZZO

```
import java.net.*; import java.io.*; import java.util.*;
public class server {
public static void main (String args[]) throws Exception
    {ServerSocket ss = new ServerSocket(2500);
    Socket s=ss.accept();
    InputStream is = s.getInputStream();
    ObjectInputStream ois = new ObjectInputStream(is);
    boolean go=true;
    while (go)
    {try{
        Date d =(Date) ois.readObject();
       System.out.println(d);}
   catch (IOException e) {System.out.println(e);
   go=false;
   }}}}
```

SHUTDOWNOUTPUT: ESEMPIO DI UTILIZZO

- Si consideri un proxy server = un programma che si interpone tra un client ed un server, inoltrando le richieste e le risposte dall'uno all'altro.
- un proxy server semplicemente copis tutto ciò che riceve sul suo input sul sull' output, in entrambe le direzioni
- supponiamo riceva un EOF dal client, deve propagarloal Server
- non può però chiudere il socket con mediante cui è collegato al Server, perchè il server potrebbe inviare ulteriori dati
- Oppure: può essere utilizzato per sovrapporre parte del protocollo di chiusura del socket con la ricezione di alcuni dati
- Esempio: il client invia una singola richiesta al server ed esegue lo shutdownOutput non appena ha inviato la richiesta. Il protocollo di chiusura del socket viene sovrapposto alla ricezione della risposta.

TCP: SHUTDOWNINPUT

- Dopo aver eseguito una shutDownInput su un socket, tutte le read successive su quel socket da parte dell'applicazione locale restituiscono una condizione di EOF (un read count=-1 oppure una EOFExecption)
- Il comportamento dell'applicazione remota in corrispondenza di un socket chiuso in input varia a seconda della piattaforma
- Soluzione 1:
  - l'applicazione remota può continuare a scrivere dati all'altro estremo della connessione, può inviare altri dati che vengono scartati dal destinatario
  - il mittente non si accorge che i dati vengono scartati
- Soluzione 2:
  - in alternativa: ogni invio su un socket chiuso in input genera un reset della comunicazione e causa una SocketException()

IMPOSTAZIONE LINGER TIME

- SetSoLinger( )=false, timeout=non significativo. L'applicazione non attende il completamento del protocollo di chiusura che viene eseguito in background
- SetSolinger( )=true, timeout=0,
  - si effettua una procedura di 'hard closure', i dati vengono scartati, non si esegue il protocollo FIN-ACK. Viene invece inviato un segmento RST (reset connection) . Il destinatario solleva una SocketException ( ) .
- SetSoLinger( )= true, timeout≠0,
  - l'applicazione si blocca fino allo scadere del timeout, oppure termina prima se la procedura di chiusura viene completata regolarmente prima dello scattare del time out
  - allo scadere del time out, a seconda della piattaforma su cui si esegue la JVM
    - si effettua una procedura di 'hard closure'
    - si inviano i dati rimanenti e si esegue il protocollo FIN/ACK

SOCKET I/O

```
Socket socket;
OutputStream out=socket.getOutputStream();
byte[] buffer = new byte[8192];
int offset = 0;
int count = buffer.length;
out.write(buffer,offset, count);
```

- Tutte le operazioni di scrittura su un socket TCP sono sincrone rispetto al buffer locale, asincrone rispetto al ricevente
- Questo implica che
  - se il buffer di output associato al socket è pieno la write si blocca fino a che vi è spazio nel buffer
  - se c'è spazio sufficiente nel buffer, la write scrive i dati nel buffer e restituisce il controllo al chiamante, senza attendere che i dati vengano ricevuti dal destinatario

SOCKET I/O

- una write bloccata perchè non vi è spazio nel buffer di spedizione, può sbloccarsi perchè alcuni dati vengono eliminati dal buffer quando si è ricevuto il loro riscontro da parte del destinatario
- se esiste spazio per una parte dei dati, questi vengono bufferizzati e l'applicazione si blocca in attesa che ci sia spazio per i rimanenti
- quando si scrive un dato sul socket, non vi è nessuna garanzia ne' che i dati scritti precedentemente siano stati ricevuti dal destinatario ne' che siano stati inviati sulla rete
- E' spesso opportuno concatenatre un BufferedOutputStream ad un OutputStream per minimizzare il numero di cambiamenti di contesto tra la JVM ed il supporto sottostante
  - i dati vengono bufferizzati dalla JVM ed inviati in blocco al supporto
  - per scaricare i dati bufferizzati dalla JVM nel buffer TCP utilizzare l'operazione flush() in punti opportuni del programma

SOCKET I/O

```
Socket socket;
InputStream in=socket.getInputStream();
byte[] buffer = new byte[8192];
int offset = 0;
int size = buffer.length;
int count=in.read(buffer,offset,size);
```

- una operazione di input da un socket si blocca se il buffer di ricezione è vuoto
- Il numero di dati letti dall'input buffer può però essere inferiore al numero di dati richiesti
  - count può essere inferiore a size() dopo l'operazione di read()

SOCKET I/O

- se si utilizza un DataInputStream oppure un ObjectInputStream il comportamento della read è diverso
- Viene invocato il metodo interno DataInput.readFully che restituisce il controllo al chiamante solo quando
  - il dato richiesto è stato letto ed assemblato completamente
  - è stato letto l'EOF
- Ad esempio, se sto leggendo un oggetto, la readFully( ) attende che tutti i byte che contengono la rappresentazione serializzata dell'oggetto siano ricevuti, prima di restituire il controllo al chiamante
- Inputstream.available( ) restuisce il numero dei byte disponibili nell'input buffer.

INVIO DI OGGETTI SU CONNESSIONI TCP

- Per inviare oggetti su una connessione TCP occorre definire l'oggetto come istanza di una classe che implementa l'interfaccia Serializable
- E' possibile associare i filtri ObjectInputStream/ ObjectOutputStream agli stream di bytes associati al socket e restituiti da getInputStream/getOutputStream
- Quando creao un ObjectOutputstream viene scritto lo stream header sullo stream. In seguito scrivo gli oggetti che voglio inviare sullo stream
- L'header viene scritto una sola volta quando lo stream viene creato e viene letto quando viene creato il corrispondente ObjectInputStream
- L'invio/ ricezioni degli oggetti sullo/dallo stream avviene mediante scritture/letture sullo stream (writeObject, readObject)

OBJECT INPUT/OUTPUT STREAM: DEADLOCK

- Supponiamo che un'applicazione A1 apra una connessione verso A2 ed invii ad A2 uno stream di oggetti
- A1 associa alla connessione un ObjectOutputStream, mentre A2 associa alla medesima connessione un ObjectInputStream
- Quando A1 crea l'ObjectOutputStream, l'header dello stream viene registrato ed inviato sulla connessione
- Quando A2 crea l' ObjectInputStream
  - la JVM accede tenta di recuperare l'header dello stream dal socket associato alla connessione
  - Se l'header non è presente, la JVM si blocca in attesa di ricevere l'header sul socket
  - ATTENZIONE: per prevenire situazioni di deadlock occorre porre attenzione sull'ordine con cui vengono creati gli stream di Input/Output

OBJECT INPUT/OUTPUT STREAM: DEADLOCK

Se i due partners della connessione eseguono entrambi il seguente frammento di codice (s è il socket associato alla connessione)

```
ObjectInputStream in = new ObjectInputStream(s.getInputStream( ));
ObjectOutputStream out = new ObjectOutputStream(s.getOutputStream( ));
```

si verifica una situazione di deadlock..

Infatti,

- entrambi tentano di leggere l'header dello stream dal socket
- l'header viene generato quando viene viene creato l'ObjectOutputStream
- nessuno dei due è in grado di generare l'ObjectOutputStream, perchè bloccato
- E' sufficiente invertire l'ordine di creazione degli stream in uno dei partner

INVIO DI OGGETTI SU UNA CONNESSIONE TCP

```
import java.io.*;
   public class Studente implements Serializable {
   private int matricola;
   private String nome, cognome, corsoDiLaurea;
   public Studente (int matricola, String nome, String cognome,
                    String corsoDiLaurea)
   {this.matricola = matricola; this.nome = nome;
     this.cognome = cognome; this.corsoDiLaurea = corsoDiLaurea;}
   public int getMatricola () { return matricola; }
   public String getNome () { return nome; }
   public String getCognome () { return cognome; }
   public String getCorsoDiLaurea () { return corsoDiLaurea; }
```

INVIO DI OGGETTI SU UNA CONNESSIONE TCP- LATO SERVER

```
import java.io.*; import java.net.*;
public class Server {
public static void main (String args[]) {
try { ServerSocket server = new ServerSocket (3575);
      Socket clientsocket = server.accept();
      ObjectOutputStream output =
       new ObjectOutputStream (clientsocket.getOutputStream ());
      output.writeObject("<Welcome>");
      Studente studente = new Studente
                  (14520,"Mario","Rosso","Informatica");
      output.writeObject(studente);
      output.writeObject("<Goodbye>");
      clientsocket.close();
      server.close();} catch (Exception e) {   }   } }
```

INVIO DI OGGETTI SU UNA CONNESSIONE TCP-LATO CLIENT

```
import java.io.*; import java.net.*;
public class Client { public static void main (String args[  ]) {
try {Socket socket = new Socket ("localhost",3575);
    ObjectInputStream input=new ObjectInputStream(socket.getInputStream());
    String beginMessage = (String) input.readObject();
    System.out.println (beginMessage);
    Studente studente = (Studente) input.readObject();
    System.out.print (studente.getMatricola()+" - ");
    System.out.print (studente.getNome()+""+studente.getCognome()
        +"- ");
    System.out.print (studente.getCorsoDiLaurea()+"\n");
    String endMessage = (String)input.readObject();
    System.out.println (endMessage); socket.close();}
catch (Exception e) { System.out.println (e); }  } }
```

INVIO DI OGGETTI SU UNA CONNESSIONE TCP- LATO CLIENT

Stampa prodotta lato Client

<Welcome>

14520 - Mario Rossi - Informatica

<Goodbye>

ESERCIZIO: ASTA ELETTRONICA

Sviluppare un programma client server per il supporto di un'asta elettronica. Ogni client possiede un budget massimo B da investire.
Il client può richiedere al server il valore V della migliore offerta pervenuta fino ad un certo istante e decidere se abbandonare l'asta, oppure rilanciare. Se il valore ricevuto dal server supera B, l'utente abbandona l'asta, dopo aver avvertito il server. Altrimenti, il client rilancia, inviando al server un valore maggiore di V.
Il server invia ai client che lo richiedono il valore della migliore offerta ricevuta fino ad un certo momento e riceve dai client le richieste di rilancio. Per ogni richiesta di rilancio, il server notifica al client se tale offerta può essere accettata (nessuno ha offerto di più nel frattempo), oppure è rifiutata.

ESERCIZIO: ASTA ELETTRONICA

Il server deve attivare un thread diverso per ogni client che intende partecipare all'asta.

La comunicazione tra clients e server deve avvenire mediante socket TCP. Sviluppare due diverse versioni del programma che utilizzino, rispettivamente:

- la serializzazione offerta da JAVA in modo da scambiare oggetti tramite le connessione TCP
- una codifica testuale dei messaggi spediti tra client e sever

PROGRAMMAZIONE CONCORRENTE: IL PROBLEMA DEI 5 FILOSOFI

- cinque filosofi siedono ad una tavola rotonda con un piatto di spaghetti davanti, una forchetta (o bacchette cinesi, a seconda della versione)a destra e una forchetta a sinistra

- la vita di un filosofo alterna periodi in cui mangia a quelli in cui pensa

- ciascun filosofo ha bisogno di due forchette per mangiare, ma che le forchette vengano prese una per volta. Dopo essere riuscito a prendere due forchette il filosofo mangia per un pò, poi lascia le forchette e ricomincia a pensare

IL PROBLEMA DEI 5 FILOSOFI

- Problema proposto da E.Dijkstra nel '65: sviluppo di un algoritmo che impedisca lo stallo (deadlock) o la morte d'inedia (starvation).

- Il deadlock può verificarsi se ciascuno dei filosofi tiene in mano una forchetta senza mai riuscire a prendere l'altra. Il filosofo F1 aspetta di prendere la forchetta che ha in mano il filosofo F2, che aspetta la forchetta che ha in mano il filosofo F3, e così via in un circolo vizioso.

IL PROBLEMA DEI 5 FILOSOFI

- problema proposto da E.Dijkstra nel '65 come problema di sincronizzazione tra più attività concorrenti

- Starvation si verifica quando un processo non riesce mai ad acquisire le risorse di cui ha bisogno

- la situazione di starvation può verificarsi indipendentemente dal deadlock se uno dei filosofi non riesce mai a prendere entrambe le forchette.

FILOSOFI A PRANZO: LE BACCHETTE

```
public class bacchetta {
        private boolean taken;
        private int bacid;
        public bacchetta (int bacid)
            {this.bacid=bacid;
            this.taken = false; }
```

FILOSOFI A PRANZO: LE BACCHETTE

```
public synchronized void take(int filid) throws
                                        InterruptedException
        { while (taken)
            wait();
                taken=true;
                System.out.println(filid+"ho preso la
                                        racchetta"+bacid);
            }
public synchronized void drop(int filid) {
            taken = false;
            System.out.println(filid+"ho lasciato la
                                        racchetta"+bacid);
            notifyAll();   } }
```

FILOSOFI A PRANZO: IL FILOSOFO

```
import java.util.*;
public class filosofo implements Runnable {
        private bacchetta left;
        private bacchetta right;
        private int filid;
        public filosofo (bacchetta left, bacchetta right,
          int filid)     {
                this.filid=filid;
                this.left=left;
                this.right=right;
                }
```

FILOSOFI A PRANZO: IL FILOSOFO

```
public void run( )  {
      try {  for (int i=0; i<2; i++){
            System.out.println (filid+""+"thinking");
            Thread.sleep (500);
            right.take(filid);
            left.take(filid);
            System.out.println (filid+""+"Mangio");
            Thread.sleep(1000);
            right.drop(filid);
            left.drop(filid);
         }}catch (InterruptedException e){} }
}
```

FILOSOFI A PRANZO: INIZIALIZZAZIONE

```
import java.util.concurrent.*;
public class filosoficondeadlock {
public static void main (String [ ] args) throws Exception
{int size =5;
if (args.length >1)  size = Integer.parseInt(args[1]);
ExecutorService exec = Executors.newFixedThreadPool(size);
bacchetta [  ] vectbacchetta = new bacchetta[size];
for (int i=0; i < size; i++)   vectbacchetta[i]= new
  bacchetta(i);
for (int i= 0; i< size; i++)
    {exec.execute(new filosofo(vectbacchetta[i],
                          vectbacchetta[(i +1)%size],i));} }
}
```

FILOSOFI A PRANZO: ESEMPIO DI ESECUZIONE

1thinking

0thinking

2thinking

4thinking

3thinking

1ho preso la racchetta2

1ho preso la racchetta1

1Mangio

2ho preso la racchetta3

4ho preso la racchetta0

4ho preso la racchetta4

4Mangio

4ho lasciato la racchetta0

4ho lasciato la racchetta4

4thinking

FILOSOFI A PRANZO: ESEMPIO DI ESECUZIONE

3ho preso la racchetta4

1ho lasciato la racchetta2

1ho lasciato la racchetta1

2ho preso la racchetta2

2Mangio

2ho preso la racchetta2

2Mangio

```
0ho preso la racchetta1
0ho preso la racchetta0
0Mangio
1thinking
2ho lasciato la racchetta3
0ho lasciato la racchetta1
0ho lasciato la racchetta0
0thinking.................
```

In questo caso il deadlock non si verifica...

FILOSOFI A PRANZO: ESECUZIONE CON DEADLOCK

Per provocare il deadlock modificare il codice in modo che un thread si sospenda dopo che ha preso la bacchetta di destra

```
public void run(){
try { for (int i=0; i<2; i++){
    System.out.println (filid+""+"thinking");
    Thread.sleep(500);
    right.take(filid);
    Thread.sleep(2500);
    left.take(filid);
    System.out.println (filid+"  "+"Mangio");
    Thread.sleep(1000);
    right.drop(filid);  left.drop(filid);  }}catch
                              (InterruptedException e){ } } }
```

FILOSOFI A PRANZO: ESECUZIONE CON DEADLOCK

1thinking

2thinking

0thinking

3thinking

4thinking

1ho preso la racchetta2

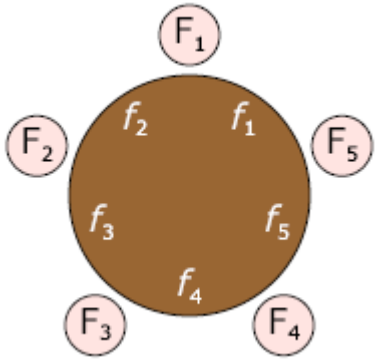
2ho preso la racchetta3

0ho preso la racchetta1

4ho preso la racchetta0

3ho preso la racchetta4

FILOSOFI A PRANZO:EVITARE IL DEADLOCK



Possibile soluzione: I filosofi prendono le bacchette in ordine numerico crescente.

F_1 deve prendere la bacchetta f_1 prima di poter prendere la seconda bacchetta f_2,i filosofi F_2, F_3 e F_4 si comportano in modo analogo, prendendo sempre la bacchetta f_i prima della bacchetta f_{i+1}

Invece il filosofo F_5 deve prendere prima la bacchetta b_1 e poi la bacchetta b_5. In questo modo si crea un'asimmetria che evita il deadlock.

FILOSOFI A PRANZO:EVITARE IL DEADLOCK

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class filsofisenzadeadlock {
public static void main (String [] args) throws Exception
{int size =5;
if (args.length >1)
size = Integer.parseInt(args[1]);
ExecutorService exec = Executors.newFixedThreadPool(5);
```

FILOSOFI A PRANZO: EVITARE IL DEADLOCK

```
Bacchetta [ ] vectbacchetta = new bacchetta[size];
for (int i=0; i < size; i++)
vectbacchetta[i]= new bacchetta(i);
for (int i= 0; i< size-1; i++)
   {exec.execute(
       new filosofo(vectbacchetta[i],vectbacchetta[(i+1)],i));}
   exec.execute(
     new filosofo(vectbacchetta[0],vectbacchetta[size],size-1));
    }
}
```