PCAD
Programmazione Concorrente
Algoritmi Distribuiti

Arnaud Sangnier
arnaud.sangnier@unige.it

Programmazione Rete in Java

Due modi principali di comunicare

- Comunicazione **per flussi** (streaming) : come TCP
  - Informazione sono trasmesse in ordine
  - Non c'è perdita dei messaggi
  - Inconvenienti:
    - Necessità una connessione
    - Necessità di risorse per gestire la correttezza della comunicazione
- Comunicazione **per pacchetti** (datagram): come UDP
  - L'ordine nella trasmissione dei pacchetti non è garantito
    - Un pacchetto mandato per prima può arrivare dopo un pacchetto mandato in un secondo tempo
  - Non ci sono garanzie
    - Un pacchetto mandato può essere perso

Paradigma di funzionamento

- Comunicazione **per flussi** (streaming) : come TCP
  - Modello Client-Server
  - Un server aspetta connessione
  - Poi comunica con il processo che ha richiesto la connessione

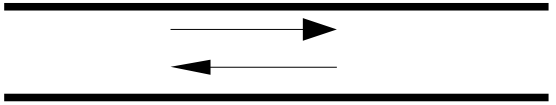
Avere un server multi-threaded permette di

gestire più connessioni in parallelo

- Comunicazione **per pacchetti** (datagram): come UDP
  - Un processo aspetta messaggi su un port
  - I messaggi possono arrivare da diversi processi
  - Secondo i messaggi ricevuti fa azioni

Comment vedere TCP

TCP -> comunicazione con i flussi, i.e. comunicazione connessa



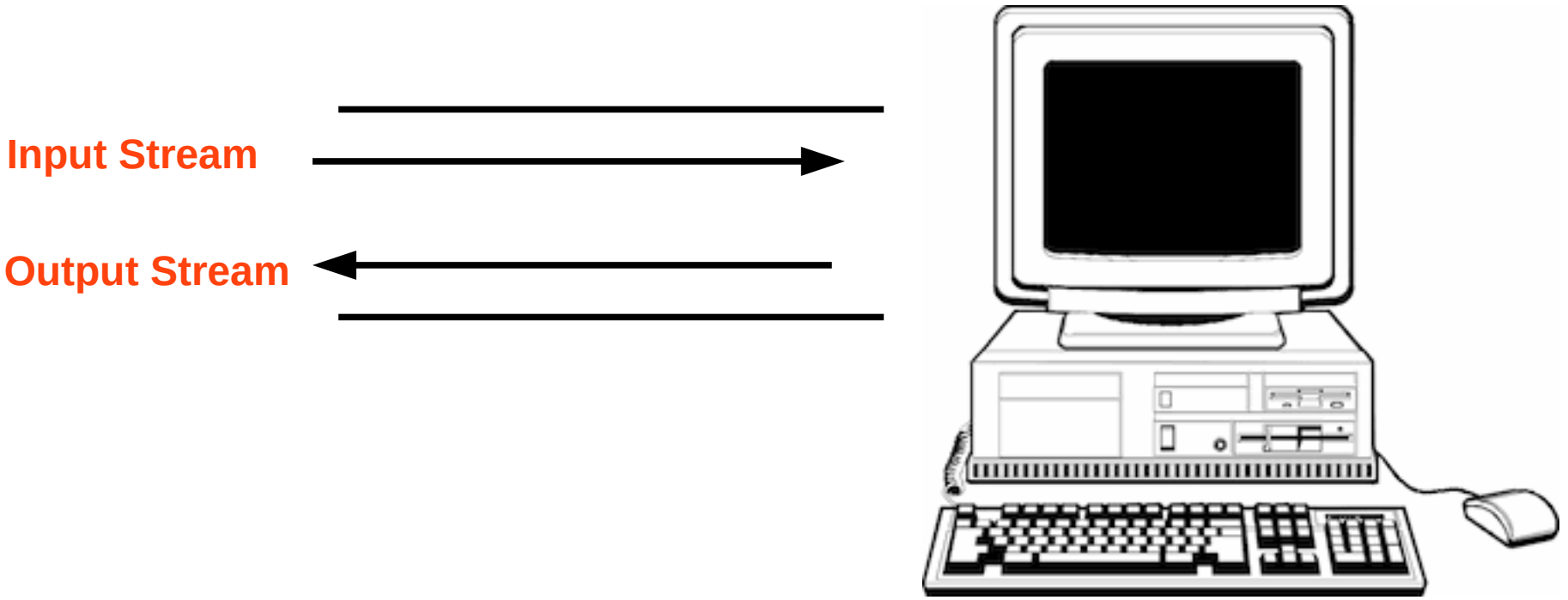
Client

Server

Per creare un sistema TCP

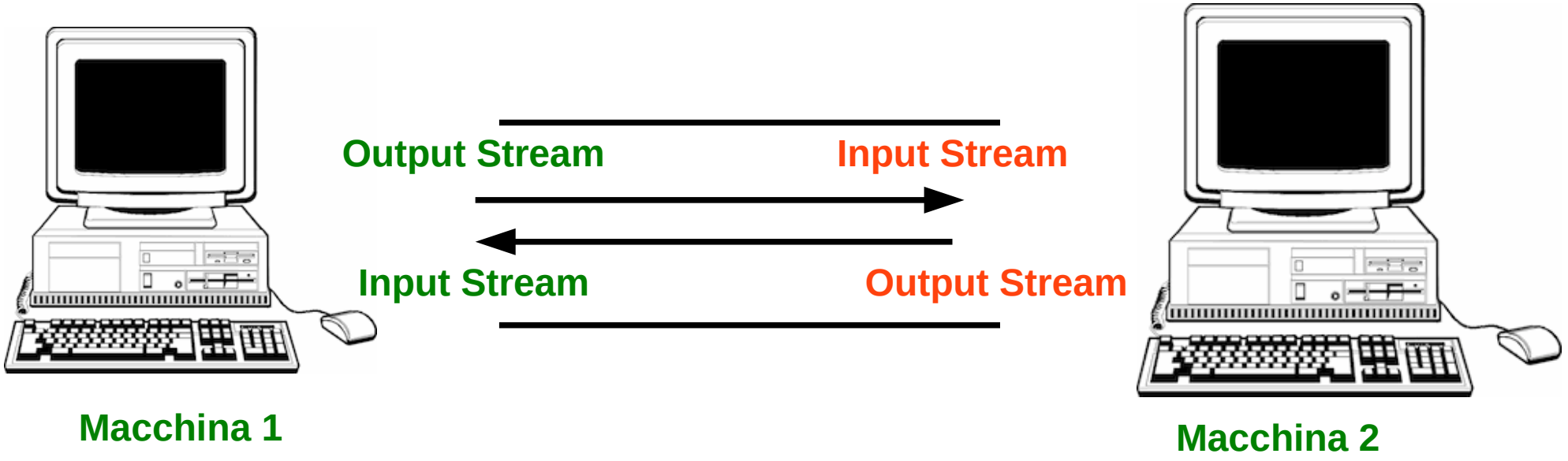
- Come comunicare in TCP in Java
- Come connettersi ad un server (presente su un computer e ascoltando su un port)
  - lato **client**
- Come ricevere dei messaggi
- Come mandare dei messaggi
- Come creare un servizio che ascolta su un port d
  - lato **server**

I flussi nella rete



Dal punto di vista della macchina, il flusso d'Input sono i dati che arrivano e il flusso d'Output, i dati che escono

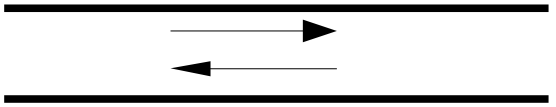
I flussi nella rete



I data che escono sul flusso di output della macchina 1 arrivano sul flusso d'input della macchina 2!

Per il nostro problema

- **TCP -> modalità connessa**



Client

Server

- Come connettersi ad un servizio?
- Come mandare dei dati ad un servizio?
- Come ricevere i dati mandati?

Legame tra rete e flussi

- Gli **sockets** permettono di connettersi a delle macchine distante e di comunicare tramite flussi
- Quindi e grazie ai flussi che mandiamo e riceviamo dei dati

Le socket in una slide

- Una socket rappresenta una connessione fra due macchine
- Passa fra due port
- Operazione basiche realizzate con una socket
  - Connettersi ad una macchina distante ad un port specifico
  - Mandare dei dati
  - Ricevere dei dati
  - Chiudere una connessione
  - Connettersi ad un port della macchina dove si trova la socket
  - Accettare delle connessione
- Le due ultime operazione sono necessari per il lato server (cf la classe **ServerSocket**)

Esempio semplice

- Programmare un server ed un cliente per un servizio echo
- Dopo avere accettato la connessione, Il server aspetta una string che finisce con ‘\n’ e la rimanda al client dopo avere inserito “ECHO” al inizio
- Assumiamo che il server ascolta sul port 4242
- Per l’esempio, il server sarà localhost, cioè il server sarà al indirizzo IP 127.0.0.1

Le socket in una slide

- Una socket rappresenta una connessione fra due macchine
- Passa fra due port
- Operazione basiche realizzate con una socket
  - Connettersi ad una macchina distante ad un port specifico
  - Mandare dei dati
  - Ricevere dei dati
  - Chiudere una connessione
  - Connettersi ad un port della macchina dove si trova la socket
  - Accettare delle connessione
- Le due ultime operazione sono necessari per il lato server (cf la classe **ServerSocket**)

Cliene per il servizio echo

- Domande da porsi prima di programmare il client:
  - Su quale macchina sta il servizio con il quale vogliamo comunicare o quale il suo indirizzo IP
    - **localhost** o **127.0.0.1**
  - Su qual port è connesso questo service
    - **port 4242**
  - Qual è la forma della comunicazione?
    - Il client manda string finendo con ‘\n’
    - Il client inizia la conversazione
    - Il server risponde con la stessa string completata con ‘ECHO ’

Creazione di socket TCP

- Come creare una socket che si connette al port 4242 di localhost
- Uso della classe **java.net.Socket**
  - Ci sono più costruttore possibile
  - Usiamo: :
    - **public Socket(String host, int port) throws UnknownHostException, IOException**
    - **host** è il nome della macchina distante
    - **port** è il numero di port

```
Socket socket=new Socket("localhost",4242);
```

Creazione di socket TCP

- Cosa succede quando facciamo :

```
Socket socket=new Socket("localhost",4242);
```

- Una socket è creata fra la macchine locale e **localhost** (per l'esempio faciamo tutto il local, ma qua 'localhost' è da prendere come una maccchina distante)
- La socket è attaccata localmente ad un port temporaneo (scelto dalla JVM)
- Alla creazione della socket, una richiesta di connessione è mandata verso **localhost** sul **port 4242**
- Sella connessione viene accettata, il costruttore finisce normalmente
- Dopo di che si può usare la socket per comunicare

Essere attento alle eccezione

- Il prototipo per creare una socket TCP è:
  - **public Socket(String host, int port) throws UnknownHostException, IOException**
- È importante ricuperare le eccezione sollevate per evitare che un programma fallisce in un modo brutale
- È ancora più importante in programmazione rete perché i problemi che possono accadere sono **numerosi**:
  - numero di port sbagliato, host sconosciuto, problemi con la rete, con i flussi, etc

```
try{...}
catch(Exception e){
    System.out.println(e);
    e.printStackTrace();
}
```

Esempio di connessione

```
import java.net.*;
import java.io.*;
public class ClientEcho{
   public static void main(String[] args){
     try{
      Socket socket=new Socket("localhost",4242);
      socket.close();
     }
     catch(Exception e){
       System.out.println(e);
       e.printStackTrace();
     }
   }
}
```

Uso della socket creata

- Per mettere fine alla connessione:
  - **void close()**
- Per comunicare tramite la socket:
  - **public InputStream getInputStream() throws IOException**
    - per ricuperare il flusso d'input per i dati che riceviamo
  - **public OutputStream getOutputStream() throws IOException**
    - per ricuperare il flusso d'output per i dati che mandiamo
- **Osservazione:**
  - Le comunicazione sulle sockets sono **bi-direzionale**
  - Si usa la stessa socket per ricevere e mandare dei dati
  - **Ma i flussi sono diversi**

Ricuperare i flussi

- Ecco come ricuperare i flussi :

```
Socket socket=new Socket("localhost",4242);
InputStream is=socket.getInputStream();
OutputStream os=socket.getOutputStream();
```

- E per usare i filtri che rendono l'uso dei flussi più semplice

```
Socket socket=new Socket("localhost",4242);
BufferedReader br=new BufferedReader(
    new InputStreamReader(socket.getInputStream()));
PrintWriter pw=new PrintWriter(
   new OutputStreamWriter(socket.getOutputStream()));
```

Funzionamento del nostro client

1) Creare una socket per connettersi
2) Ricuperare i flussi di input/output
3) Mandare la string `"Blalbla"`
4) Aspettare di ricevere la string di rispostate
5) Stampare il messagio ricevuto
6) Chiudere i flussi e la socket

Creazione di un client TCP

```
import java.net.*;
import java.io.*;
public class ClientEcho{
    public static void main(String[] args){
        try{
            Socket socket=new Socket("localhost",4242);
            BufferedReader br=new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter pw=new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
            pw.println("Blabla");
            pw.flush();
            String mess=br.readLine();
            System.out.println("Message from the server:"+mess);
            pw.close();
            br.close();
            socket.close();
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Punti importanti

- Non dimenticare di ricuperare le eccezione
- Usare la funzione flush() per svuotare il buffer legato al flusso di output e mandare il messaggio
- La lettura con readLine() blocca finché un messaggio sia ricevuto

E ora ?

- Si usa la classe **ServerSocket**
- La cosa pratica
  - a parte l’attesa di connessione
  - la manipolazione dei socket sarà simile allo client

Creazione di un server

- Usiamo la classe **java.net.ServerSocket**
- Esistono anche qua diversi costruttori
- Due funzione ci interessano
  - **public ServerSocket(int port) throws IOException**
    - Crea un socket che ascolta sul port e serve solo ad aspettare le domande di connessione

```
ServerSocket server=new ServerSocket(4242);
```

  - **public Socket accept() throws IOException**
    - Aspetta una richiesta di connessione
    - Ritorna una socket che servirà alla comunicazione

Et il nome della macchina ?


```
ServerSocket server=new ServerSocket(4242);
```

- Perché il server si trova sulla macchina dove è eseguito il programma
- Il **server** non ha bisogno di conoscere il nome della macchina dove si trova

Funzionamento del nostro server

1) Creare un **ServerSocket** legato al port
2) Attendere una richiesta di connessione con **accept**
3) Ricuperare la socket per la comunicazione
4) Ricuperare i flussi di input/output
5) Comunicare con il client
6) Chiudere i flussi e la socket
7) Andare al punto 2)

Creazione di un server TCP

```
import java.net.*;
import java.io.*;
public class ServerEcho{
    public static void main(String[] args){
        try{
            ServerSocket server=new ServerSocket(4242);
            while(true){
                Socket socket=server.accept();
                BufferedReader br=new BufferedReader(new InputStreamReader(socket.getInputStream()));
                PrintWriter pw=new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
                String mess=br.readLine();
                System.out.println("Receive message:"+mess);
                pw.println("ECHO "+mess);
                pw.flush();
                br.close();
                pw.close();
                socket.close();
            } }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Osservazione

- Il nostro server non accetta più connessione allo stesso tempo
  - Quando fa un **accept**, comunica con un client specifico e deve avere finito la comunicazione per potere comunicare con un nuovo client
- Come risolvere questo problema?
  - Il serve deve essere **multi-threaded**
  - Deve nello ‘stesso’ tempo potere:

    1) comunicare con un client

    2) accettare nuove connessione
- Il thread principale sarà responsabile per 2)
- Ad ogni nuova connessione si creerà un nuovo thread per comunicare con il nuovo client

Service per il server multi-threaded

```
import java.net.*;
import java.io.*;
import java.lang.*;
public class ServiceEcho implements Runnable{
    public Socket socket;
    public ServiceEcho(Socket s){
        this.socket=s;
    }
    public void run(){
        try{
            BufferedReader br=new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter pw=new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
            String mess=br.readLine();
            System.out.println("Receive message:"+mess);
            pw.println("ECHO "+mess);
            pw.flush();
            br.close();
            socket.close();
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
}}}
```

Server multi-threaded

```
import java.net.*;
import java.io.*;
public class ServerEchoConcur{
    public static void main(String[] args){
        try{
            ServerSocket server=new ServerSocket(4242);
            while(true){
                Socket socket=server.accept();
                ServiceEcho serv=new ServiceEcho(socket);
                Thread t=new Thread(serv);
                t.start();
            }
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```