Esercitazione 2020-10-15

Nella prima parte della lezione si vede l'esercizio di TicTacToe.

Funzione select() - continuo

Come abbiamo visto, la funzione select() restituisce un interno, e questo valore indica **quante delle socket sono state interessate da un'evento**; possiamo quindi usare questo valore per effettuare dei controlli. Inoltre, se la select() restituisce un valore <0, allora si è verificato un errore nell'operazione.

Nel secondo tipo di esercizio visto:

```
int fd;
int newfd[10];
int next;

fd_set readfds;
FD_ZERO(&readfds);

while(1){
    FD_SET(fd, &readfds);
    select(maxfd + 1, &readfds, 0, 0, 0); // attendiamo per un evento di lettura

    if(FD_ISSET(fd, &readfds)){
        newfd[++next] = accept(fd, ...);
        FD_ZERO(&readfds);
        FD_SET(newfd[next], &readfds);

        if(newfd[next] > maxfd)
            maxfd = newfd[next];
    }

    if(FD_ISSET(newfd[i], &readfds)){
        read(newfd[i], buf, sizeof(buf));
        // processiamo il buffer
    }
}
```

> Questo codice è abbastanza di fantasia, infatti non vediamo nessuna inizializzazione di i, nessun ciclo su di essa e nessuna inizializzazione di next.

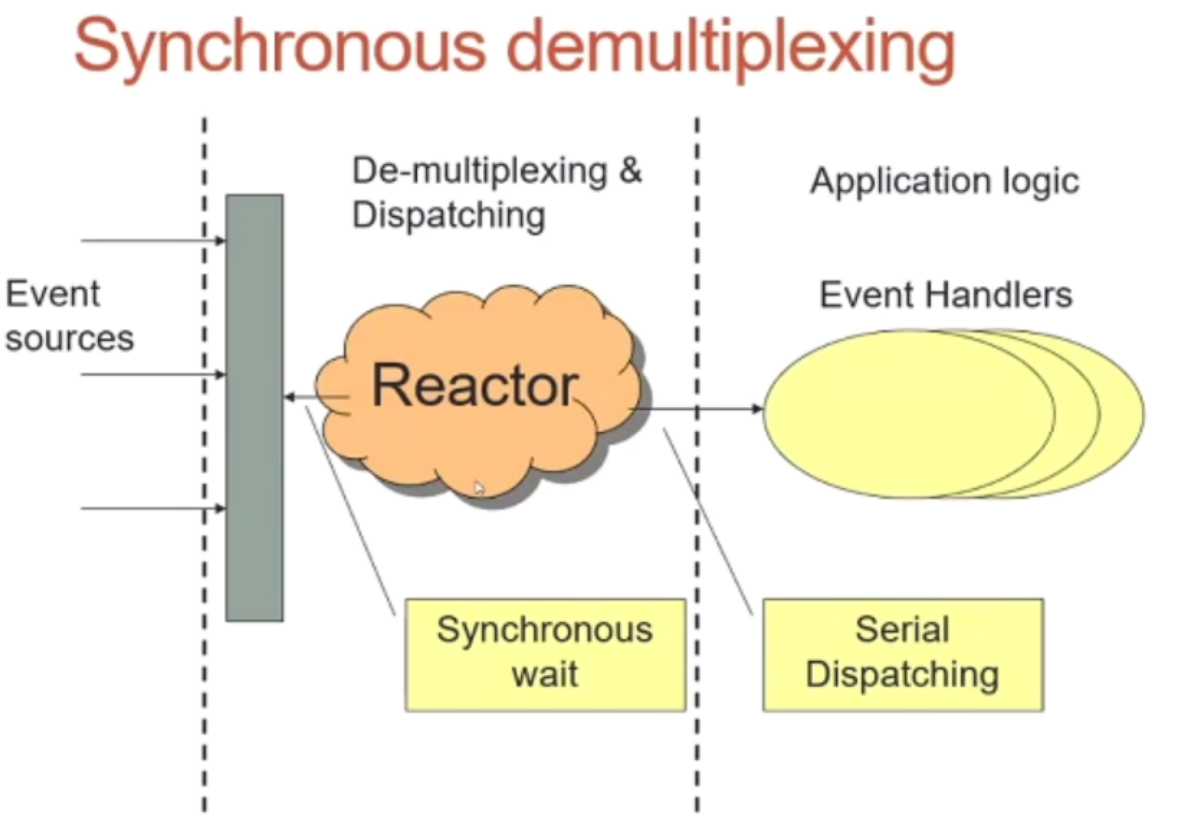
utilizziamo `newfd[10]` per mantenere le richieste di connessione da parte dei client. Inizialmente mettiamo all'interno dell'insieme degli eventi di lettura e con FD_SET andiamo a collocare fd all'interno dell'insieme, in modo da monitorare quell'insieme.

Quando invochiamo la select() dobbiamo fare attenzione a `maxfd`, che deve essere opportunamente inizializzato, in modo che comprenda tutte le socket che vogliamo monitorare.

Dopo la select() andiamo a verificare se si è **verificato un evento di lettura su fd**; possiamo quindi invocare l'accept() in maniera **non bloccante**; ci restituisce un descrittore che andiamo a porre all'interno dell'array contenente tutti i descrittori delle socket. Poniamo quindi il descrittore all'interno di **FD_SET**, in modo tale che la **select()** possa monitorare anche questa nuova socket.

Quando si verifica un evento su uno di questi nuovi descrittori, sarà gestito dalla porzione di codice `if(FD_ISSET(newfd[i], &readfds)){ ... }`.

Demultiplexing Sincrono - Pattern Reactor



Il pattern **Reactor**, attraverso una select() che abbiamo appena visto, permette di realizzare il **demultiplexing sincrono**:

La funzione che abbiamo visto blocca il processo invocante, ma quando il processo "si risveglia", possiamo scegliere tra le diverse operazioni possibili quella consentita da mandare in esecuzione; una volta che la select restituisce il controllo al processo, abbiamo con la FD_SET se la socket s1/s2 era pronta (aveva avuto un evento di lettura).

Possiamo quindi pensare di incapsulare il codice che vogliamo eseguire all'interno di **handlers**; di conseguenza non sappiamo con che ordine il codice verrà eseguito, e quindi l'ordine dipenderà dagli eventi.

Come usare la select() con l'esercizio 2.2

In questo esercizio abbiamo due operazioni bloccanti: la lettura da stdin e la lettura da socket; possiamo quindi pensare di usare la select() anche per la lettura da tastiera: dobbiamo osservare il descrittore di file (SO) associato allo stdin (0), e poi il descrittore della socket connessa.

In questo esercizio avevamo una variabile di stato che ci diceva di chi fosse il turno per scrivere; per usare la select() non dobbiamo eliminare quella variabile, ma modificarne il comportamento. Prima veniva modificata tramite l'invio di caratteri speciali, ora possiamo modificarla nel momento in cui avviene un evento.

🏁 1:10

Codice Interessante

```
while(status != EXIT){ // nulla di nuovo qui
    FD_ZERO(&readfds);
    FD_SET(sd2, &readfds);          // aggiungo al set la socket
    FD_SET(STDIN_FILENO, &readfds); // aggiungo al set il canale di lettura da
tastiera

    maxfds = MAX(STDIN_FILENO, &readfds);   // trovo l'identificatore più grande
    select(maxfds, &readfds, NULL, NULL, NULL); // la select ascolta entrambi i
descrittori: file(stdin) e socket.

    // se il descrittore è quello da file (tastiera) lo stato è "YOU", ovvero
posso scrivere; se il descrittore è un altro, sicuramente la socket, allora
leggo.
    if(FD_ISSET(STDIN_FILENO, &readfds)) status = YOU;
    else status = PEER;

    memset(buf, 0, sizeof(buf));    // pulisco il buffer

    switch(status){ ... } // molto simile
}
```

MAX() è una macro: `#define MAX(x, y) ((x) > (y) ? (x) : (y))` ci restituisce l'elemento maggiore.

Dallo **switch()** in poi il codice è uguale; l'unica cosa che cambia è lo switching dello status, che non è più dettato da un carattere speciale, ma dalla select().

Programmazione socket in Java

🏁 1:25 2020-10-15

Socket orientate ai datagram

Nel momento in cui passiamo al linguaggio java non passiamo semplicemente ad un altro linguaggio, ma anche ad un altro paradigma: invece di procedurale, dobbiamo programmare ad oggetti.

Abbiamo la classe **DatagramSocket**; in java si è preferito differenziare le socket a seconda del tipo di servizio (datagram/stream), per rendere più espressivo il codice scritto. Le socket orientate ai datagram sono delgi endpoint liberi, possono quindi comunicare con chiunque, e non devono essere **accoppiati**. Abbiamo un modello di comunicazione **punto-multipunto**.

Abbiamo quindi un'istanza di questa classe in un processo, ed un'istanza di questa classe in un altro processo; un'istanza rappresenta un **endpoint**, e con due istanze potremo comunicare.

L'altra classe che useremo è **DatagramPacket**, che astrae un concetto già visto, ovvero la **costruzione del pacchetto da spedire**; nel codice scritto in C questo veniva fatto tramite la **preparazione di un buffer**, inviato con **sendto()**.
Ci aspettiamo che in **un'istanza di DatagramPacket** si possano inserire i **dati da spedire** e si possa prevedere un **intestazione (header)**.

Nell'intestazione ci saranno le informazioni utili affinchè il pacchetto arrivi a destinazione; siccome il modello non prevede un accoppiamento, ogni volta va specificato l'endpoint a cui vogliamo spedire: specifichiamo l'indirizzo del destinatario, ovvero **indirizzo IP : porta**.

Immaginiamo di voler scrivere un server DNS

Usiamo come esempio il server DNS perchè si basa sul modello di comunicazione UDP ed utilizza quindi la comunicazione orientata ai datagram.

Il numero di porta che usiamo è il **numero di porta 53**.

Per scrivere il server come prima operazione dobbiamo preparare una socket di tipo datagram: `DatagramSocket serverSocket = new DatagramSocket(53)`; il costruttore ci consente di specificare alcune informazioni utili: stiamo creando una socket da usare lato server, di conseguenza passiamo il numero di porta; se non passiamo l'indirizzo IP assumiamo la costante INADDR_ANY, ovvero qualunque indirizzo presente sulla macchina.

All'interno di questo costruttore viene "costruito" il canale di comunicazione, equivalente alle operazioni precedenti alla **bind()**. Inoltre un oggetto di tipo **DatagramSocket** mantiene **il descrittore** della socket (intero); quando invochiamo su quell'oggetto una send(), stiamo dicendo che vogliamo spedire con quella socket (descrittore contenuto al suo interno).

Non è necessario prevedere una funzione di conversione, questo perchè al di sotto di Java gira una macchina virtuale **uguale per tutti i sistemi**, e di conseguenza la rappresentazione degli interi è la stessa.

Eccezioni

A differenza del C possiamo prevedere delle eccezioni, che sono gestite dalla classe **SocketException**.

receive()

Una volta creata la socket, la socket è **pronta per essere usata**, e non dobbiamo quindi invocare la bind() esplicitamente. Andiamo quindi a preparare un **DatagramPacket** per ricevere dei dati: `DatagramPacket receivePacket = new DatagramPacket(receiveData, receivedata.length);`
receiveData non è altro che un "buffer" (come in c), rappresentato da un **array di bytes**.

Una volta preparato il datagrampacket possiamo invocare il metodo `serverSocket.receive(receivePacket);`. Quando invochiamo la receive(), se non sono presenti dati da ricevere, il processo **viene sospeso**. Se invece i dati sono disponibili, ci aspettiamo che **receivePacket** venga inizializzato con i dati ricevuti.

Come scoprire l'indirizzo del mittente?

Non essendo queste socket connesse, potremmo avere bisogno di rispondere ad un eventuale messaggio; dove prendiamo le informazioni utili a costruire il **datagram di risposta?** Dobbiamo infatti necessariamente indicare le informazioni del ricevente al momento in cui costruiamo un datagram. Di conseguenza, nel **receivePacket**, oltre ai **dati** ricevuti dal canale, sono presenti anche delle **informazioni di controllo** utili a costruire il datagram di risposta.

Abbiamo dei metodi che ci permettono di estrarre le informazioni che ci interessano:

- `InetAddress cliAddr = receivePacket.getAddress();` L'indirizzo del mittente è di tipo **InetAddress**.
- `int cliPort= receivePacket.getPort();` L'indirizzo del mittente è di tipo **intero**.

Successivamente viene utilizzata la **send()** per rispondere al client.

Client DNS

send()

Dal lato client, per preparare il canale di comunicazione si deve procedere allo stesso modo:

1. Istanziare una **DatagramSocket** - stessa procedura del server - non passiamo alcun argomento
2. Allochiamo il buffer - stessa procedura del server
3. Prepariamo il pacchetto da spedire ed usiamo il metodo send()

Per preparare il pacchetto usiamo la classe **DatagramPacket** vista durante il server, con un costruttore diverso:

`DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, hostname, 53);`

A differenza della costruzione nel server, dove avevamo semplicemente preparato il buffer per estrarre i dati dalla rete, in questo caso dobbiamo specificare:

1. Area di memoria che contiene il contenuto da spedire
2. Lunghezza dei dati che vogliamo spedire
3. L'indirizzo di trasporto del server, quindi **IP:porta**

Inviamo il pacchetto usando la funzione **send()**: `clientSocket.send(sendPacket);`

🏁 1:49 2020-15-09

Un Esempio

I due processi, client e server, si scambiano delle stringhe: il client invia una stringa al server ed il server la trasforma in uppercase e la restituisce al client.

Server

```
class datagramServer{
    public static void main(String args[]) throws Exception{
        DatagramSocket serverSocket = new DatagramSocket(1200);

        byte[] receiveData = new byte[1024];
        byte[] sendData;

        while(true){
            DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length); // creo il pacchetto da ricevere sulla base di receiveData
e la sua lunghezza
            serverSocket.receive(receivePacket);    // leggo il pacchetto
inviato dal client e lo salvo in receivePacket

            // processo i dati ricevuti
```

```
            String receivedSentence = new String(receivePacket.getData(), 0,
receivePacket.getLength());
            String capitalizedSentence = receivedSentence.toUpperCase();
            sendData = capitalizedSentence.getBytes();  // trasformo la stringa
in array di bytes

            InetAddress IPAddress = receivePacket.getAddress(); // prendo
l'indirizzo del client dal pacchetto ricevuto
            int port = receivePacket.getPort();

            DatagramPacket sendPacket = new DatagramPacket(sendData,
sendData.length, IPAddress, port);

            serverSocket.send(sendPacket);
        }
    }
}
```

Client

```
class DatagramClient{
    public static void main(String args[]) throws Exception{
        Scanner inFromUser = new Scanner(System.in);
        DatagramSocket clientSocket = new DatagramSocket(); // non serve
specificare la porta

        InetAddress IPAddress = InetAddress.getLocalHost(); // usiamo
l'indirizzo di loopback

        byte[] sendData;
        byte[] receiveData = new byta[1024];

        String sentenceToSend = inFromUser.nextLine();
        sendData = sentenceToSend.getBytes();   // convertiamo la stringa in
array di bytes

        // prepariamo il pacchetto da spedire
        DatagramPacket sendPacket = new DatagramPacket(sendData,
sendData.length, IPAddress, 1200); // porta hardwired

        // spediamo il pacchetto
        clientSocket.send(sendPacket);

        // ricevo i dati

        // preparo il pacchetto dei dati da ricevere
        DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);
        // leggo i dati dal canale di trasmissione
        clientSocket.receive(receivePacket);

        // trasformo i dati ricevuti in stringa
        String modifiedSentence = new String(receivePacket.getData());
```

```
        // stampo
        Sout.println("Dal server: "+modifiedSentence);

        clientSocket.close();
    }
}
```

---

fine lezione 10