Reti e Laboratorio
Modulo Laboratorio III
AA. 2026-2027

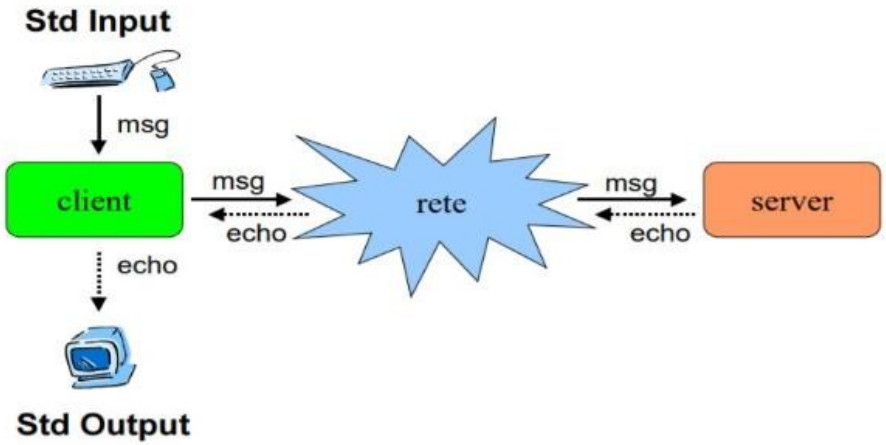
docente: Laura Ricci

laura.ricci@unipi.it

Correzione Assignment 7
"NIO Echo Server"
26/11/2026

ASSIGNMENT 7: NIO ECHO SERVER

- scrivere un programma echo server usando la libreria java NIO e, in particolare, il Selector e canali in modalità non bloccante, e un programma echo client, usando NIO (va bene anche con modalità bloccante).
- Il server accetta richieste di connessioni dai client, riceve messaggi inviati dai client e li rispedisce (eventualmente aggiungendo "echoed by server" al messaggio ricevuto).
- Il client legge il messaggio da inviare da console, lo invia al server e visualizza quanto ricevuto dal server



SERVER PROPERTIES

```
# Porta di ascolto del server.
port=12000
# Dimensione del buffer di lettura.
bufSize=4096
# Messaggio di terminazione.
exitMessage=exit
# Stringa da aggiungere al messaggio di risposta.
echoString=echoed by server
```

ECHO SERVER

```
/**
 * Reti e Laboratorio III - A.A. 2023/2024
 * Soluzione del nono assignment
 *
 * Questa classe contiene l'implementazione del server echo basato su NIO.
 * Una volta accettata la connessione con un client, il server riceve
 * una sequenza di messaggi (aventi una lunghezza massima prefissata).
 *
 * Per ogni messaggio ricevuto, il server costruisce un messaggio di risposta
 * costituito dal messaggio originale seguito dalla stringa "(echoed by server)".
 * Le operazioni di lettura e scrittura vengono effettuate in modalita' non bloccante.
 *
 */
```

SERVER - READSTATE

```
/**
 * Questa classe contiene lo stato della comunicazione fra client e server
 * durante la ricezione del messaggio da parte di quest'ultimo.
 * Le variabili in essa contenute vengono usate per tenere traccia del numero
 * di byte letti dal server e della dimensione del messaggio inviato dal client.
 */
public class ReadState {
    // Numero totale di byte letti.
    public int count;
    // Dimensione del messaggio da ricevere.
    public int length;
    // Buffer per memorizzare il messaggio e la sua lunghezza.
    public ByteBuffer buffer;
    public ReadState(int bufSize) {
        this.count = 0;
        this.length = 0;
        this.buffer = ByteBuffer.allocate(bufSize);
    }
}
```

ECHO SERVER

```
public class Server {
    // Percorso del file di configurazione.
    public static final String configFile = "server.properties";
    // Porta di ascolto del server.
    public static int port;
    // Dimensione del buffer di risposta (in byte).
    public static int bufSize;
    // Messaggio di terminazione. Se ricevuto dal client, chiude la connessione.
    public static String exitMessage;
    // Stringa da aggiungere ai messaggi di risposta inviati al client.
    public static String echoString;
```

ECHO SERVER

```
public static void main(String[] args) {
    try {readConfig();}
    catch (Exception e) {
        System.err.println("Errore durante la lettura del file di configurazione.");
        e.printStackTrace();
        System.exit(1);
    }
    // Quindi apro il ServerSocketChannel e il selettore per ricevere e monitorare
    // le connessioni da parte dei vari client.
    try (
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        Selector selector = Selector.open();
    ) {
        serverSocketChannel.bind(new InetSocketAddress(port));
        // Configuro il canale in modalita' non bloccante.
        serverSocketChannel.configureBlocking(false);
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
        System.out.printf("[SERVER] In ascolto su porta %d\n", port);
```

ECHO SERVER

```
// Il server entra in un ciclo infinito nel quale:
// 1)Attende (e accetta) richieste di connessione da parte dei client.
// 2)Controlla (tramite il selettore) se ci sono canali
// pronti per essere letti o scritti.
while (true) {
    selector.select();
    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator<SelectionKey> iter = selectedKeys.iterator();
    while (iter.hasNext()) {
        SelectionKey key = iter.next();
        // Controllo se sul canale associato alla chiave esiste la
           possibilita'
        // di accettare una nuova connessione. Nel caso, la accetto
        // e registro il canale sul selettore.
```

ECHO SERVER

```
        if (key.isAcceptable()) {
            SocketChannel client = serverSocketChannel.accept();
            System.out.println("[SERVER] Nuova connessione ricevuta");
            client.configureBlocking(false);
            client.register(selector, SelectionKey.OP_READ, new ReadState(bufSize));
        }
        // Se sul canale ci sono dati pronti per essere letti, procedo con la lettura.
        else if (key.isReadable()) {
            System.out.printf("[SERVER] Dati pronti per la lettura\n");
            handleRead(selector, key);
        }
        // Se il canale e' pronto per la scrittura, posso inviare la risposta.
        else if (key.isWritable()) {
            System.out.printf("[SERVER] Dati pronti per la scrittura\n");
            handleWrite(selector, key);
        }
        iter.remove();
    }
```

SERVER - LETTURA

```
public static void handleRead(Selector selector,SelectionKey key) throws IOException {
    SocketChannel channel = (SocketChannel) key.channel();
    ReadState state = (ReadState) key.attachment();
    // Leggo dati dal canale.
    state.count += channel.read(state.buffer);
    // Se non ho (ancora) letto i 4 byte della lunghezza, termino.
    if (state.count < Integer.BYTES) return;
    // Altrimenti, controllo se ho gia' estratto la lunghezza del messaggio
    // dal buffer. Nel caso in cui non lo abbia fatto, la memorizzo nello stato.
    if (state.length == 0) {
        state.buffer.flip();
        state.length = state.buffer.getInt();
        System.out.printf("[SERVER] Ricevuta lunghezza: %d\n", state.length);
        // compact() sposta gli eventuali byte rimanenti all'inizio del buffer.
        state.buffer.compact();
    }
```

SERVER - LETTURA

```
public static void handleRead(Selector selector,SelectionKey key) throws IOException {
    SocketChannel channel = (SocketChannel) key.channel();
    ReadState state = (ReadState) key.attachment();
    // Leggo dati dal canale.
    state.count += channel.read(state.buffer);
    // Se non ho (ancora) letto i 4 byte della lunghezza, termino.
    if (state.count < Integer.BYTES) return;
    // Altrimenti, controllo se ho gia' estratto la lunghezza del messaggio
    // dal buffer. Nel caso in cui non lo abbia fatto, la memorizzo nello stato.
    if (state.length == 0) {
        state.buffer.flip();
        state.length = state.buffer.getInt();
        System.out.printf("[SERVER] Ricevuta lunghezza: %d\n", state.length);
        // compact() sposta gli eventuali byte rimanenti all'inizio del buffer.
        state.buffer.compact();
    }
```

SERVER - LETTURA

```
    if (state.count < Integer.BYTES + state.length) return;
        // Se sono qui, invece, posso estrarre il messaggio dal buffer.
        // Conosco gia' state.length poiche' l'ho letta in precedenza.
        state.buffer.flip();
        byte[] messageBytes = new byte[state.length];
        state.buffer.get(messageBytes);
        String messageString = new String(messageBytes);
        System.out.printf("[SERVER] Ricevuto messaggio: %s\n", messageString);
        // Controllo se il messaggio che ho ricevuto corrisponde alla stringa di
           terminazione. In caso affermativo, chiudo la connessione con il client.
        if (messageString.equalsIgnoreCase(exitMessage)) {
            channel.close(); // La chiusura del canale cancella anche la chiave.
            System.out.printf("[SERVER] Connessione con il client chiusa.\n");
            return;
        }
    // Preparo la risposta e registro il canale per la scrittura.
    ByteBuffer buffer = buildReplyBuffer(messageString);
    channel.register(selector, SelectionKey.OP_WRITE, buffer);
}
```

SERVER - SCRITTURA

```
 * Metodo che gestisce le operazioni di scrittura sul canale.
 * @param selector il selettore
 * @param key chiave associata al canale di comunicazione
 * @throws IOException in caso di errori di scrittura


public static void handleWrite(Selector selector, SelectionKey key) throws
IOException {
    SocketChannel channel = (SocketChannel) key.channel();
    ByteBuffer buffer = (ByteBuffer) key.attachment();
    // Provo a scrivere i dati sul canale.
    channel.write(buffer);
    // Se dopo la write() i dati nel buffer non sono stati consumati completamente,
    // allora termino (e riprovo piu' tardi).
    if (buffer.hasRemaining()) return;
        // Altrimenti, sono riuscito a mandare tutto il messaggio di risposta
        // e posso nuovamente registrare il canale in lettura.
        System.out.printf("[SERVER] Risposta inviata.\n");
        channel.register(selector, SelectionKey.OP_READ, new ReadState(bufSize));
    }
```

SERVER - RISPOSTA

```
/**
 * Costruisce la stringa di risposta e la inserisce in un ByteBuffer
 * pronto per la scrittura sul canale.
 * @param message la stringa ricevuta dal client
 * @return un buffer contenente la lunghezza (in byte) e i byte della stringa di risposta
 */
public static ByteBuffer buildReplyBuffer(String message) {
    String replyString = String.format("%s (%s)", message, echoString);
    byte[] replyBytes = replyString.getBytes();
    ByteBuffer buffer = ByteBuffer.allocate(Integer.BYTES + replyBytes.length);
    // NOTA: la flip() rende il buffer pronto per essere scritto sul canale.
    buffer.putInt(replyBytes.length).put(replyBytes).flip();
    return buffer;
}
```

SERVER - CONFIGURAZIONE

```
/**
 * Legge il file di configurazione del server.
 * @throws FileNotFoundException se il file non esiste
 * @throws IOException se si verifica un errore durante la lettura
 */
private static void readConfig() throws FileNotFoundException, IOException {
    InputStream input = new FileInputStream(configFile);
    Properties prop = new Properties();
    prop.load(input);
    port = Integer.parseInt(prop.getProperty("port"));
    bufSize = Integer.parseInt(prop.getProperty("bufSize"));
    exitMessage = prop.getProperty("exitMessage");
    echoString = prop.getProperty("echoString");
    input.close();
}
```

CLIENT - PROPERTIES

```
# Nome dell'host su cui risiede il server.
hostname=localhost
# Porta di ascolto del server.
port=12000
# Dimensione del buffer per il client.
bufSize=4096
# Messaggio di terminazione.
exitMessage=exit
```

ECHO CLIENT

```
/**
 * Questa classe contiene l'implementazione del client NIO che:
 * 1) Richiede all'utente di inserire un messaggio.
 * 2) Invia il messaggio al server.
 * 3) Legge la risposta del server e la stampa su schermo.
 * 4) Chiude la connessione se riceve la stringa di terminazione
 */
public class Client {
    // Nome del file di configurazione.
    public static final String configFile = "client.properties";
    // Nome dell'host del server.
    public static String hostname;
    // Porta di ascolto del server.
    public static int port;
    // Dimensione buffer.
    public static int bufSize;
    // Messaggio di terminazione. Se letto da input provoca la chiusura della
       connessione.
    public static String exitMessage;
```

ECHO CLIENT

```
public static void main(String[] args) {
    // Leggo il file di configurazione.
    try {readConfig();}
    catch (Exception e) {
        System.err.println("[CLIENT] Errore durante la lettura del file di
configurazione.");
        e.printStackTrace();
        System.exit(1);
    }
    // Alloco un buffer con la dimensione fissata per invio/ricezione messaggi
    ByteBuffer buffer = ByteBuffer.allocate(bufSize);
    // Apro uno Scanner per leggere l'input da tastiera e
    // un SocketChannel per collegarmi al server.
    try (
        Scanner scanner = new Scanner(System.in);
        SocketChannel sc=SocketChannel.open(new InetSocketAddress(hostname,port));
        ) {
```

ECHO CLIENT

```
while (true) {
    // Chiedo all'utente di inserire un messaggio da tastiera.
    System.out.printf("[CLIENT] Inserisci il messaggio: ");
    String inputStr = scanner.nextLine();
    byte[] message = inputStr.getBytes();
    // Preparo il buffer per la scrittura (inserisco dati nel buffer).
    buffer.clear();
    // Inserisco la lunghezza del messaggio e il messaggio vero e proprio.
    buffer.putInt(message.length);
    buffer.put(message);
    // Preparo il buffer per la lettura e poi li scrivo sul canale).
    buffer.flip();
    sc.write(buffer);
    // Se il messaggio letto da tastiera corrisponde alla stringa
    // di terminazione, esco dal ciclo.
    // NOTA: il SocketChannel si chiudera' automaticamente.
    if (inputStr.equalsIgnoreCase(exitMessage)) break;
```

ECHO CLIENT

```
        // A questo punto, attendo la risposta del server.
        // La risposta e' costituita dalla coppia (lunghezza, messaggio).
        buffer.clear();
        sc.read(buffer);
        buffer.flip();
        // Estraggo la lunghezza e quindi leggo il contenuto del messaggio.
        int replyLength = buffer.getInt();
        byte[] replyBytes = new byte[replyLength];
        buffer.get(replyBytes);
        System.out.printf("[CLIENT] Ricevuto: %s\n", new String(replyBytes));
        }
}
```

ECHO CLIENT

```
 /**
 * Legge il file di configurazione del client.
 * @throws FileNotFoundException se il file non esiste
 * @throws IOException se si verifica un errore durante la lettura
 */
private static void readConfig() throws FileNotFoundException, IOException {
    InputStream input = new FileInputStream(configFile);
    Properties prop = new Properties();
    prop.load(input);
    hostname = prop.getProperty("hostname");
    port = Integer.parseInt(prop.getProperty("port"));
    bufSize = Integer.parseInt(prop.getProperty("bufSize"));
    exitMessage = prop.getProperty("exitMessage");
    input.close();
}
```