Azzolini Riccardo

Proxy

1 Problema dei socket

In una tipica applicazione client-server basata su socket, il client e il server devono implementare meccanismi per:

- la connessione;
- la creazione dei dati da trasmettere;
- l'invio e ricezione delle richieste;
- la ricostruzione dei valori dei dati ricevuti.

Il problema è che, in questo modo, si "mischiano" alla logica applicativa molti dettagli relativi alla comunicazione client-server: ciò può creare delle difficoltà, soprattutto in fase di manutenzione e nel porting.

2 Separare logica applicativa e comunicazione

Idealmente, il programmatore che implementa il client dovrebbe potersi concentrare sulla logica applicativa, cioè:

- la richiesta dei servizi al server (sulla base della loro interfaccia);
- l'elaborazione dei dati così ottenuti.

In altre parole, sarebbe conveniente separare la logica applicativa dai dettagli dei meccanismi di interazione con il server.

Analogamente, il programmatore lato server dovrebbe concentrarsi sulla codifica dei servizi da offrire, quindi si vorrebbe separare la realizzazione di tali servizi dai meccanismi per la comunicazione con il client.

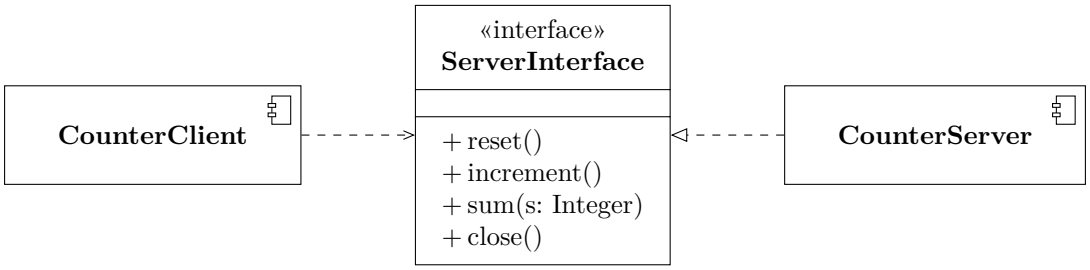
3 Esempio: contatore remoto

Per prima cosa, si introduce un semplice esempio, inizialmente implementato "normalmente" con i socket, che verrà poi modificato al fine di mostrare come separare la parte di comunicazione dalla logica applicativa.

- Il server gestisce un oggetto contatore, che permette sostanzialmente due operazioni:
  - l'inizializzazione o reset, che azzera il contatore;
  - l'incremento del conteggio.
- Il client esegue un po' di operazioni di reset e di incremento, e alla fine calcola il tempo impiegato.

3.1 Vista logica

Dal punto di vista della logica applicativa, l'applicazione è composta dal server, che fornisce i servizi specificati da un'interfaccia, e dal client, che usufruisce di tali servizi:



3.2 Implementazione

In questa prima versione, il server gestisce solo un client alla volta; la gestione della connessione è la solita. La parte di logica applicativa è invece la seguente: per ogni comando ricevuto da un client, il server aggiorna il contatore in base all'operazione richiesta, e poi restituisce il nuovo valore del contatore.

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class CounterServer {
    public static final int PORT = 8888;
    private static int counter = 0;
```

```
public static void main(String[] args) throws IOException {
    try (ServerSocket serverSocket = new ServerSocket(PORT)) {
        while (true) {
            System.out.println("Waiting for a connection...");
            try (
                Socket socket = serverSocket.accept();
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(socket.getInputStream())
                );
                PrintWriter out = new PrintWriter(
                    new BufferedWriter(
                        new OutputStreamWriter(
                            socket.getOutputStream()
                        )
                    ),
                    true
                );
            ) {
                serveClient(in, out);
                System.out.println("Closing...");
            }
        }
    }
}

private static void serveClient(BufferedReader in, PrintWriter out)
        throws IOException {
    String operation;
    while ((operation = in.readLine()) != null) {
        if (operation.equals("<reset>")) {
            counter = 0;
        } else if (operation.equals("<incr>")) {
            counter++;
        } else if (operation.startsWith("<sum>")) {
            int s = Integer.parseInt(operation.split(" ")[1]);
            counter += s;
        } else {
            System.err.println(
                "Operation not recognized: " + operation
            );
        }
        out.println(counter);
    }
```

```
    }
}
```

Anche nel client la gestione della connessione avviene come al solito, mentre nella parte applicativa:

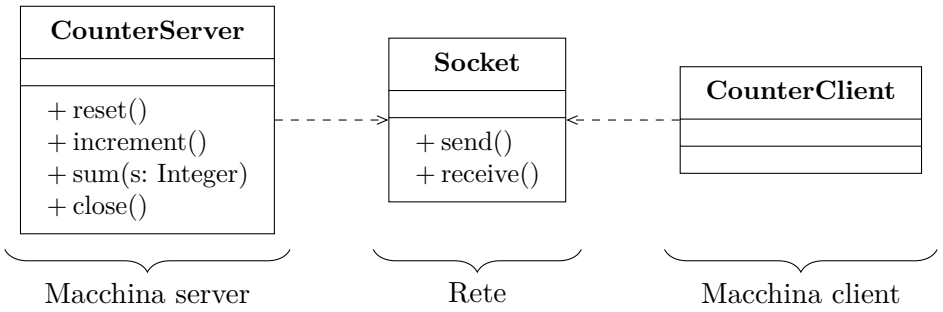
1. si richiede il reset del contatore, e viene visualizzato il risultato restituito dal server;
2. si inviano 1000 richieste di incremento, mostrando a schermo il risultato di ciascuna di esse, e misurando il tempo impiegato in totale.

```
import java.io.*;
import java.net.InetAddress;
import java.net.Socket;

public class CounterClient {
    public static void main(String[] args) throws IOException {
        InetAddress addr = InetAddress.getByName(null);
        System.out.println("addr = " + addr);
        try (
            Socket socket = new Socket(addr, CounterServer.PORT);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream())
            );
            PrintWriter out = new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream())
                ),
                true
            )
        ) {
            System.out.println("socket = " + socket);

            out.println("<reset>");
            System.out.println("reset: " + in.readLine());

            long startTime = System.currentTimeMillis();
            for (int i = 0; i < 1000; i++) {
                out.println("<incr>");
                System.out.println("increment: " + in.readLine());
            }
            long endTime = System.currentTimeMillis();
            System.out.println(
                "Elapsed time: " + (endTime - startTime) + " ms"
            );
```

```
            System.out.println("Closing...");
        }
    }
}
```

3.3 Vista di design

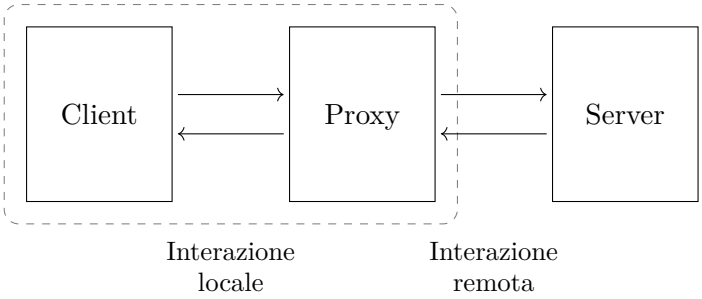
Mentre, a livello logico, il client interagisce con il server mediante l'interfaccia fornita da quest'ultimo, la vista di design mette in evidenza il fatto che, concretamente, la comunicazione avviene tramite un socket:



4 Pattern proxy lato client

Il pattern **proxy** introduce un ulteriore componente, chiamato appunto proxy, nel modello di interazione client-server.

Esso si presenta come un'implementazione del servizio remoto, locale al client (fa parte dello stesso programma), e fornisce un'interfaccia uguale a quella del server reale. I meccanismi di basso livello necessari per l'instaurazione della comunicazione e lo scambio di dati (compresa la gestione del protocollo applicativo) sono implementati e incapsulati all'interno del proxy, che inoltra al server tutte le richieste del client.



5 Esempio: contatore remoto con proxy lato client

L'interfaccia `ServerInterface` descrive i servizi offerti dal server, e quindi anche dal proxy:

```
import java.io.IOException;

public interface ServerInterface extends AutoCloseable {
    static final int PORT = 8888;

    int reset() throws IOException;
    int increment() throws IOException;
    int sum(int s) throws IOException;
    void close() throws IOException;
}
```

Tale interfaccia sarà implementata da una classe `ProxyServer`. Allora, il client non deve più preoccuparsi della comunicazione: invece, si rivolge a un proxy locale, chiamando i metodi corrispondenti ai servizi che vuole richiedere. Così, si ottiene una notevole semplificazione del codice, che adesso riguarda solo la logica applicativa:

```
import java.io.IOException;

public class CounterClient {
    public static void main(String[] args) throws IOException {
        try (ServerInterface server = new ProxyServer()) {
            System.out.println("reset: " + server.reset());

            long startTime = System.currentTimeMillis();
            for (int i = 0; i < 1000; i++) {
                System.out.println("increment: " + server.increment());
            }
            long endTime = System.currentTimeMillis();
            System.out.println(
                "Elapsed time " + (endTime - startTime) + " ms"
            );
        }
    }
}
```

La classe `ProxyServer`, che realizza il proxy, si occupa di stabilire la connessione con il server (nel costruttore) e di inoltrare a quest'ultimo tutte le richieste arrivate dal client:

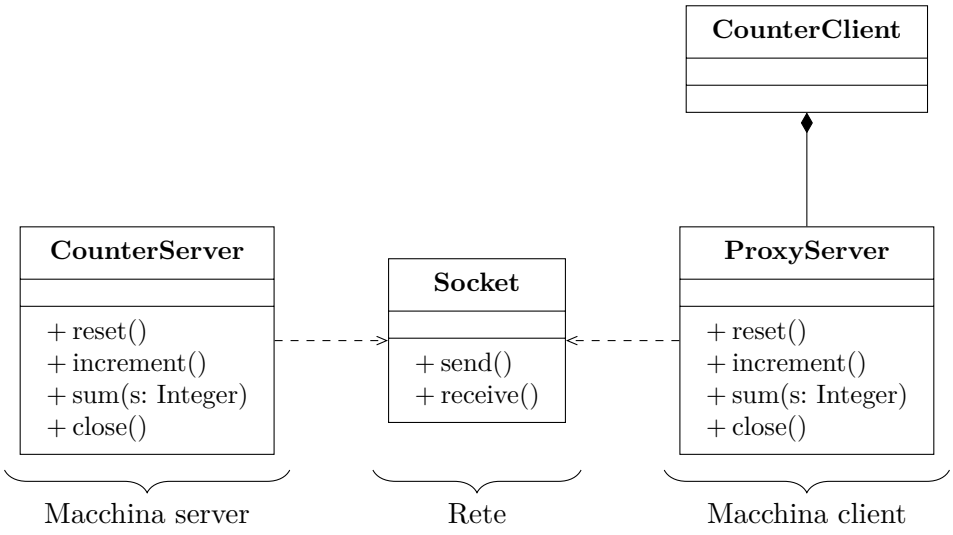
```java
import java.io.*;
import java.net.InetAddress;
import java.net.Socket;

public class ProxyServer implements ServerInterface {
    private Socket socket = null;
    private BufferedReader in = null;
    private PrintWriter out = null;

    public ProxyServer() throws IOException {
        InetAddress addr = InetAddress.getByName(null);
        System.out.println("addr = " + addr);
        socket = new Socket(addr, ServerInterface.PORT);
        System.out.println("socket = " + socket);

        try {
            in = new BufferedReader(
                new InputStreamReader(socket.getInputStream())
            );
            out = new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream())
                ),
                true
            );
        } catch(IOException e) {
            if (out != null) {
                out.close();
            }
            if (in != null) {
                in.close();
            }
            socket.close();
            throw e;
        }
    }

    public int reset() throws IOException {
        out.println("<reset>");
        return result();
    }

    public int increment() throws IOException {
        out.println("<incr>");
```

```
        return result();
    }

    public int sum(int s) throws IOException {
        out.println("<sum> " + s);
        return result();
    }

    private int result() throws IOException {
        return Integer.parseInt(in.readLine());
    }

    public void close() throws IOException {
        System.out.println("Closing...");
        out.close();
        in.close();
        socket.close();
    }
}
```

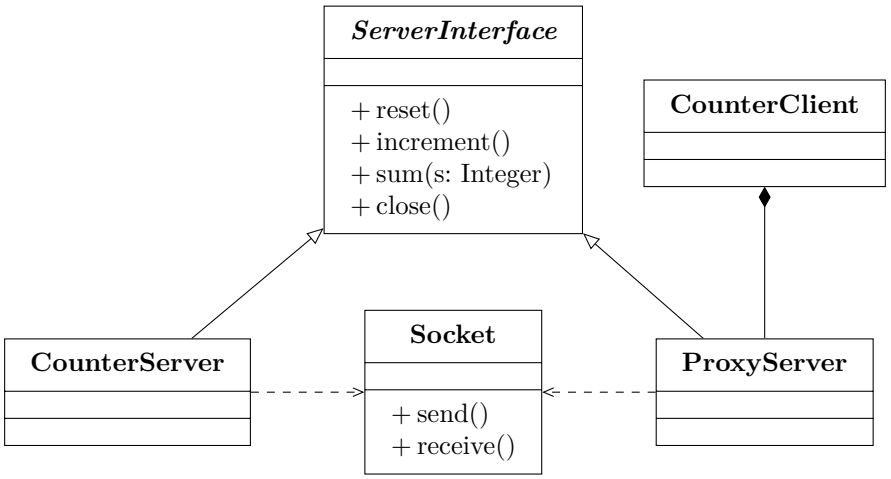
Il server non ha invece bisogno di essere modificato.

5.1 Vista di design

Con l'aggiunta del proxy, il diagramma che rappresenta il design dell'applicazione cambia: sulla macchina client è presente un `ProxyServer` (collegato al client da una relazione di composizione, cioè parte del programma client), che utilizza il socket per comunicare con il server.



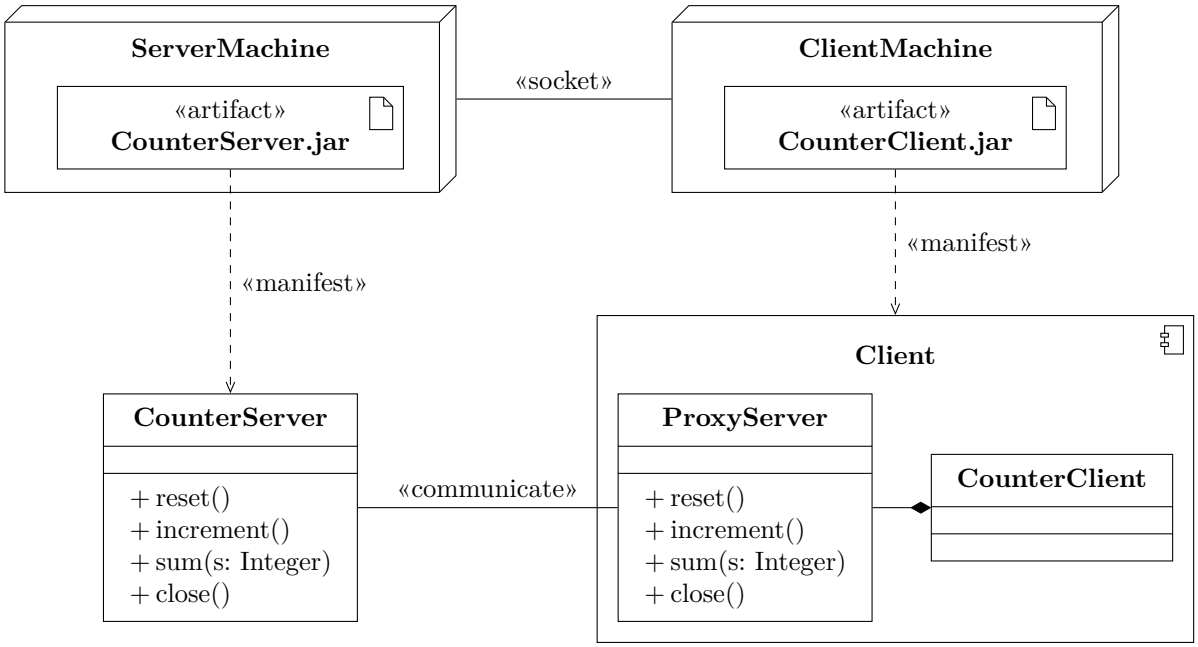
Data l'importanza dell'interfaccia che definisce i servizi forniti dal server, è opportuno metterla in evidenza. Il diagramma di design diventa allora:



Osservazione: Il server e il proxy implementano la stessa interfaccia. Questo è il caso tipico; in generale, il server deve per forza fornire almeno tutti i servizi richiesti dal proxy, ma potrebbe metterne a disposizione anche molti altri.

5.2 Deployment diagram

Dal punto di vista delle macchine fisiche, l'architettura dell'applicazione con il proxy è la seguente:

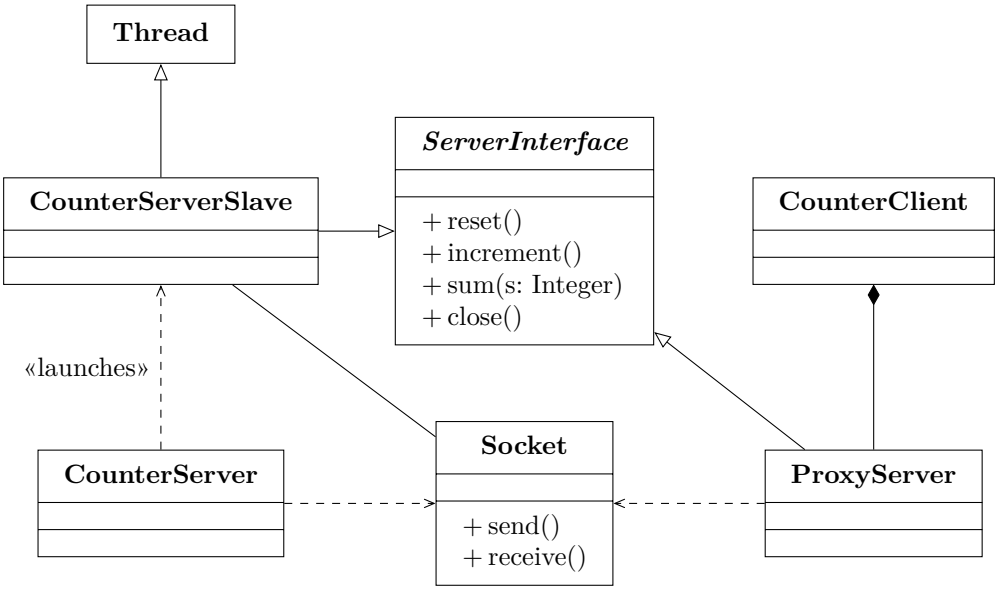


In particolare, le macchine client e server sono connesse via socket, al livello di trasporto, mentre la comunicazione applicativa avviene tra le classi `CounterServer` e `ProxyServer`.

6 Esempio: contatore remoto multi-client

Adesso, si reimplementa il server, rendendolo capace di gestire più client contemporaneamente, usando la tecnica già vista in precedenza, che consiste nella creazione di un nuovo thread per ogni connessione ricevuta.

In questa nuova versione dell'applicazione, saranno allora i thread slave (quelli che gestiscono le singole connessioni) a implementare `ServerInterface`, come mostrato dal design diagram:



Infine, per testare il sistema, verranno generati tanti thread client paralleli, ciascuno con il proprio proxy.

6.1 Implementazione

I servizi forniti dal server e richiesti dal client sono gli stessi, quindi `ServerInterface` non presenta alcuna modifica:

```
import java.io.IOException;

public interface ServerInterface extends AutoCloseable {
    static final int PORT = 8888;

    int reset() throws IOException;
    int increment() throws IOException;
    int sum(int s) throws IOException;
    void close() throws IOException;
}
```

La classe `CounterServerSlave` implementa il thread slave che gestisce la comunicazione con un singolo client:

```
import java.io.*;
import java.net.Socket;

public class CounterServerSlave implements ServerInterface, Runnable {
```

```
private final Socket socket;
private int counter = 0;

public CounterServerSlave(Socket socket) {
    this.socket = socket;
}

public int reset() {
    counter = 0;
    return counter;
}

public int increment() {
    counter++;
    return counter;
}

public int sum(int s) {
    counter += s;
    return counter;
}

public void close() throws IOException {
    System.out.println("Closing...");
    socket.close();
}

public void run() {
    try {
        try (
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream())
            );
            PrintWriter out = new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream())
                ),
                true
            )
        ) {
            serveClient(in, out);
        } finally {
            close();
        }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void serveClient(BufferedReader in, PrintWriter out)
            throws IOException {
        String operation;
        while ((operation = in.readLine()) != null) {
            int result = 0;
            if (operation.equals("<reset>")) {
                result = reset();
            } else if (operation.equals("<incr>")) {
                result = increment();
            } else if (operation.startsWith("<sum>")) {
                int s = Integer.parseInt(operation.split(" ")[1]);
                result = sum(s);
            } else {
                System.err.println(
                    "Operation not recognized: " + operation
                );
            }
            out.println(result);
        }
    }
}
```

Il main (thread master) del server ha semplicemente il compito di accettare connessioni dai client e avviare un nuovo thread slave per ciascuna di esse:

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class CounterServer {
    public static void main(String[] args) throws IOException {
        try (
            ServerSocket serverSocekt =
                new ServerSocket(ServerInterface.PORT)
        ) {
            System.out.println("Started: " + serverSocekt);
            while (true) {
                System.out.println(
                    "Server: waiting for a connection..."
```

```
                );
                Socket socket = serverSocekt.accept();
                System.out.println("Server: new client connected");
                new Thread(new CounterServerSlave(socket)).start();
            }
        }
    }
}
```

Analogamente, il main del programma client crea un certo numero di thread client (in questo esempio, 4), al fine di testare la capacità del server di gestirne più di uno alla volta:

```
import java.io.IOException;

public class CounterClient {
    private static final int NUM_CLIENTS = 4;

    public static void main(String[] args)
            throws InterruptedException, IOException {
        for (int i = 0; i < NUM_CLIENTS; i++) {
            new CounterClientThread(i).start();
            System.out.println(
                "Master client: thread " + i + " created"
            );
            Thread.sleep(2);
        }
    }
}
```

Ciascun thread del client crea una propria istanza di `ProxyServer`, con la quale richiede al server un reset e 100 somme, misurando il tempo necessario per queste ultime:

```
import java.io.IOException;

public class CounterClientThread extends Thread {
    private final int id;

    public CounterClientThread(int id) {
        this.id = id;
    }

    public void run() {
        try (ServerInterface localServer = new ProxyServer()) {
            int init = localServer.reset();
```

```
            log("reset: " + init);

            long startTime = System.currentTimeMillis();
            for (int i = 0; i < 100; i++) {
                int result = localServer.sum(1);
                log("sum: " + result);
            }
            long endTime = System.currentTimeMillis();
            log("elapsed time: " + (endTime - startTime) + " ms");

            log("closing...");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void log(String message) {
        System.out.println("Client " + id + " " + message);
    }
}
```

Infine, la classe `ProxyServer` non ha bisogno di essere alterata, dato che i servizi forniti dal server e il protocollo applicativo sono rimasti gli stessi: come sia implementato il server (ad esempio, con o senza thread slave) non è rilevante dal punto di vista del proxy.

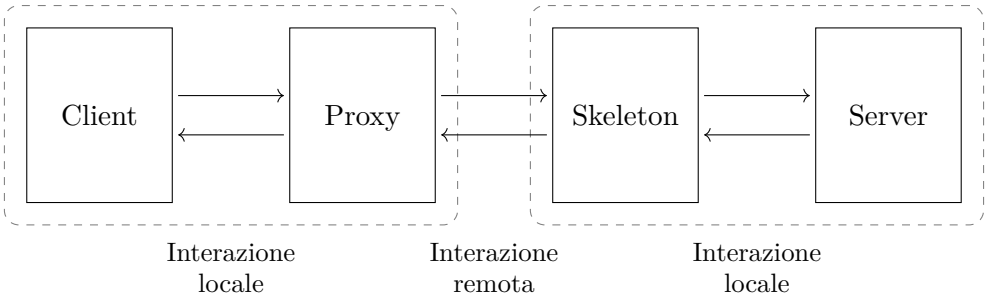
7 Pattern proxy lato server: skeleton

Il proxy lato client solleva il client dalle problematiche di comunicazione. Il server, tuttavia, ha ancora l'onere di implementare i necessari meccanismi di comunicazione assieme alla logica applicativa.

Si aggiunge allora un proxy anche a lato server, chiamato **skeleton**, che si faccia carico della comunicazione con il proxy lato client. Lo skeleton avrà la responsabilità di:

- ricevere le richieste di servizio;
- gestire il protocollo applicativo (strutturando i dati forniti in ingresso, e "confezionando" per i dati in uscita per permetterne l'invio sul socket);
- fare la chiamata al server reale, per eseguire concretamente i servizi richiesti;
- ricevere dal server eventuali risultati, e rispedirli al proxy lato client.

In questo modo, così come il client interagisce con un "server locale" (il proxy), senza doversi occupare della comunicazione, anche il server riceve le richieste da un "client locale" (lo skeleton).



7.1 Implementazione

In generale, lo skeleton può essere implementato in due modi:

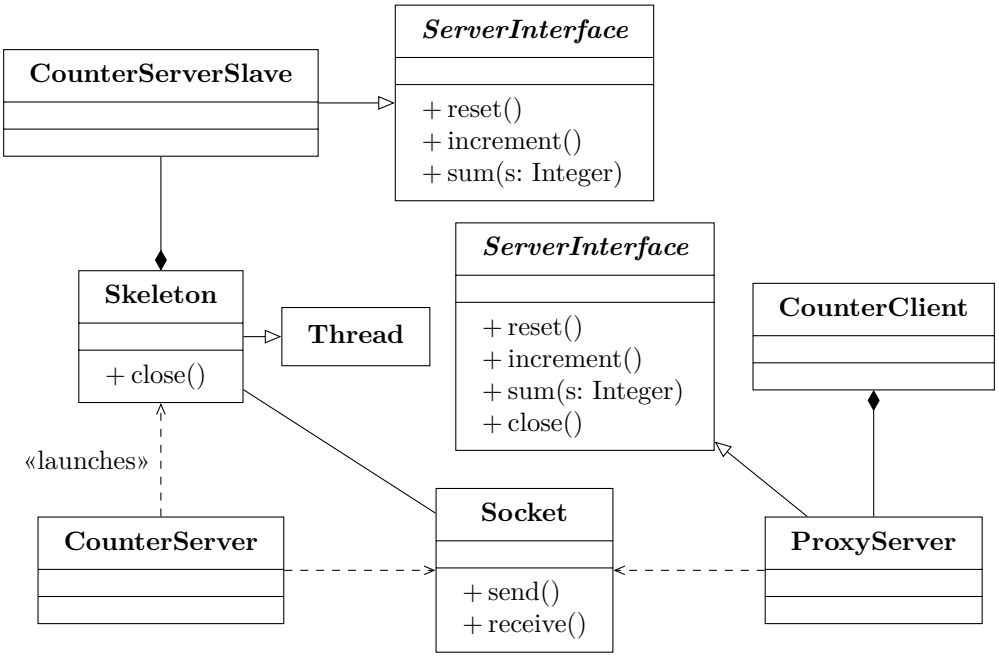
- *Per delega*: la classe `Skeleton` presenta al suo interno un riferimento alla classe server.
- *Per ereditarietà*: la classe `Skeleton` eredita dalla classe server, aggiungendo gli opportuni schemi di comunicazione.

8 Esempio: contatore remoto con proxy e skeleton

Riprendendo l'esempio del contatore remoto, si sceglie di implementare lo skeleton mediante la delega. Il funzionamento sarà il seguente:

1. Il `ProxyServer` crea un socket e si connette al `CounterServer`.
2. Subito dopo aver stabilito la connessione, il `CounterServer` lancia un thread `Skeleton`, passandogli il socket.
3. Quando arriva una richiesta di servizio dal `ProxyServer`, lo `Skeleton` la delega al `CounterServerSlave`, poi confeziona la risposta e la rimanda al `ProxyServer` tramite il socket.

Il design diagram diventa allora:



Osservazione: `CounterServerSlave` implementa una versione di `ServerInterface` nella quale non è presente il metodo `close()`, che viene invece implementato dallo `Skeleton`. Infatti, `CounterServerSlave` vede solo le richieste di servizio inoltrate localmente dallo `Skeleton`, e non ha la minima idea del fatto che ci sia un client remoto, quindi un'operazione come `close()` non avrebbe alcun senso.

8.1 Implementazione

La classe `CounterServer` implementa il thread master, che si occupa solo di accettare le connessioni e lanciare uno skeleton per ciascuna di esse:

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class CounterServer {
    public static void main(String[] args) throws IOException {
        try (
            ServerSocket serverSocekt =
                new ServerSocket(ServerInterface.PORT)
        ) {
            System.out.println("Started: " + serverSocekt);
            while (true) {
                System.out.println(
```

```
                    "Server: waiting for a connection..."
                );
                Socket socket = serverSocekt.accept();
                System.out.println("Server: new client connected");
                new Thread(new Skeleton(socket)).start();
            }
        }
    }
}
```

Come già anticipato, nella versione di `ServerInterface` usata a lato server non è più presente il metodo `close()` (poiché sarà lo `Skeleton` a occuparsi di tutto ciò che riguarda la comunicazione, compresa la chiusura del socket):

```
import java.io.IOException;

public interface ServerInterface {
    static final int PORT = 8888;

    int reset() throws IOException;
    int increment() throws IOException;
    int sum(int s) throws IOException;
}
```

La classe `CounterServerSlave` implementa solo la logica applicativa del contatore:

```
public class CounterServerSlave implements ServerInterface {
    private int counter = 0;

    public int reset() {
        counter = 0;
        return counter;
    }

    public int increment() {
        counter++;
        return counter;
    }

    public int sum(int s) {
        counter += s;
        return counter;
    }
}
```

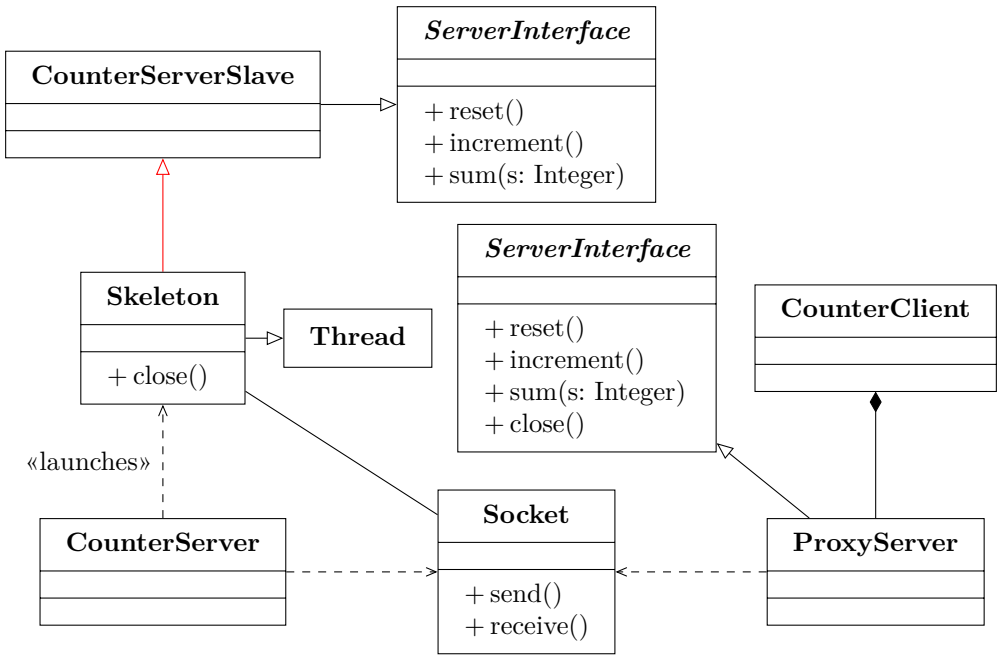
Lo `Skeleton` crea una propria istanza di `CounterServerSlave` (l'implementazione del server reale), e delega a essa le richieste di servizio, gestendo invece tutti gli aspetti della comunicazione:

```
import java.io.*;
import java.net.Socket;

public class Skeleton implements Runnable {
    private final Socket socket;
    private final CounterServerSlave server = new CounterServerSlave();

    public Skeleton(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        try {
            try (
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(socket.getInputStream())
                );
                PrintWriter out = new PrintWriter(
                    new BufferedWriter(
                        new OutputStreamWriter(socket.getOutputStream())
                    ),
                    true
                )
            ) {
                serveClient(in, out);
            } finally {
                socket.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void serveClient(BufferedReader in, PrintWriter out)
            throws IOException {
        String operation;
        while ((operation = in.readLine()) != null) {
            int result = 0;
            if (operation.equals("<reset>")) {
                result = server.reset(); // Chiamata al server
```

```
            } else if (operation.equals("<incr>")) {
                result = server.increment(); // Chiamata al server
            } else if (operation.startsWith("<sum>")) {
                int s = Integer.parseInt(operation.split(" ")[1]);
                result = server.sum(s); // Chiamata al server
            } else {
                System.err.println(
                    "Operation not recognized: " + operation
                );
            }
            out.println(result);
        }
    }
}
```

8.2 Alternativa: skeleton derivato

In alternativa alla delega, si potrebbe utilizzare l'ereditarietà, definendo `Skeleton` come sottoclasse di `CounterServerSlave`. A livello di design diagram, l'unica differenza rispetto alla versione con la delega è la relazione tra le suddette due classi (che qui è evidenziata in rosso):



9 Accesso concorrente a oggetti remoti

Negli esempi precedenti, il server creava un contatore separato ("privato") per ogni client. In questo caso, non ci possono essere problemi di concorrenza.

Spesso, invece, diversi client richiedono al server dei servizi che utilizzano le medesime risorse. Allora, si hanno le solite problematiche di concorrenza: race condition (da evitare attraverso mutua esclusione), ecc.

10 Esempio: contatore remoto condiviso

Per illustrare come il server deve gestire una risorsa alla quale accedono concorrentemente più client, si considera ancora l'esempio del contatore remoto, ma questa volta con un singolo contatore condiviso tra tutti i client.

Come prima, i servizi offerti dal server (e messi quindi a disposizione dei client, tramite i proxy) sono specificati dall'interfaccia `ServerInterface`. Per alleggerire il codice, al fine di concentrarsi sulla gestione della concorrenza, è stato rimosso il metodo `sum`.

```
import java.io.IOException;

public interface ServerInterface {
    static final int PORT = 9999;

    int reset() throws IOException;
    int increment() throws IOException;
}
```

L'implementazione del proxy lato client è analoga a quella vista in precedenza:

```
import java.io.*;
import java.net.InetAddress;
import java.net.Socket;

public class Proxy implements AutoCloseable, ServerInterface {
    private Socket socket = null;
    private BufferedReader in = null;
    private PrintWriter out = null;

    public Proxy() throws IOException {
        InetAddress addr = InetAddress.getByName(null);
        System.out.println("addr = " + addr);
        socket = new Socket(addr, ServerInterface.PORT);
        System.out.println("socket = " + socket);
```

```
        try {
            in = new BufferedReader(
                new InputStreamReader(socket.getInputStream())
            );
            out = new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream())
                ),
                true
            );
        } catch(IOException e) {
            if (out != null) {
                out.close();
            }
            if (in != null) {
                in.close();
            }
            socket.close();
            throw e;
        }
    }

    public int reset() throws IOException {
        out.println("<reset>");
        return result();
    }

    public int increment() throws IOException {
        out.println("<incr>");
        return result();
    }

    private int result() throws IOException {
        return Integer.parseInt(in.readLine());
    }

    public void close() throws IOException {
        System.out.println("Closing...");
        out.close();
        in.close();
        socket.close();
    }
}
```

Ciascun thread client crea una propria istanza di `Proxy`, e la usa per richiedere al server un'operazione di reset e una serie di incrementi. Tra il reset e gli incrementi è inserito uno sleep, per fare in modo che ciascun client inizi a richiedere incrementi solo dopo l'esecuzione di tutti i reset.

```
import java.io.IOException;

public class CounterClient extends Thread {
    public void run() {
        try (Proxy server = new Proxy()) {
            int init = server.reset();
            Thread.sleep(500);
            for (int i = 0; i < 1000; i++) {
                server.increment();
            }
        } catch (InterruptedException | IOException e) {}
    }
}
```

La classe `MultiClient` si occupa di avviare più client concorrenti:

```
import java.io.IOException;

public class MultiClient {
    private static final int NUM_CLIENTS = 4;

    public static void main(String[] args)
            throws InterruptedException, IOException {
        CounterClient[] clients = new CounterClient[NUM_CLIENTS];
        for (int i = 0; i < NUM_CLIENTS; i++) {
            clients[i] = new CounterClient();
            clients[i].start();
        }
        for (CounterClient client : clients) {
            client.join();
        }
    }
}
```

Il main del server è quello "standard" per la gestione di client multipli, ma, in più, crea anche un oggetto `Counter` (chiamato `server`, perché corrisponde all'implementazione del "server" invocata dallo skeleton) da condividere:

```
import java.io.IOException;
import java.net.ServerSocket;
```

```
import java.net.Socket;

public class CounterServer {
    public static void main(String[] args) throws IOException {
        Counter server = new Counter();
        try (
            ServerSocket serverSocket =
                new ServerSocket(ServerInterface.PORT)
        ) {
            System.out.println("Started: " + serverSocket);
            while (true) {
                System.out.println(
                    "Server: waiting for a connection..."
                );
                Socket socket = serverSocket.accept();
                System.out.println("Server: new client connected");
                new Thread(new Skeleton(socket, server)).start();
            }
        }
    }
}
```

Lo skeleton riceve un riferimento al `Counter` condiviso, al quale inoltra le richieste di operazioni ricevute dai client tramite il socket:

```
import java.io.*;
import java.net.Socket;

public class Skeleton implements Runnable {
    private final Socket socket;
    private final Counter server;

    public Skeleton(Socket socket, Counter server) {
        this.socket = socket;
        this.server = server;
    }

    public void run() {
        try {
            try (
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(socket.getInputStream())
                );
                PrintWriter out = new PrintWriter(
```

```
                    new BufferedWriter(
                        new OutputStreamWriter(socket.getOutputStream())
                    ),
                    true
                )
            ) {
                serveClient(in, out);
            } finally {
                socket.close();
            }
        } catch (IOException e) {}
    }

    private void serveClient(BufferedReader in, PrintWriter out)
            throws IOException {
        String operation;
        while ((operation = in.readLine()) != null) {
            int result = 0;
            if (operation.equals("<reset>")) {
                result = server.reset();
            } else if (operation.equals("<incr>")) {
                result = server.increment();
            } else {
                System.err.println(
                    "Operation not recognized: " + operation
                );
            }
            out.println(result);
        }
    }
}
```

La classe `Counter` è l'elemento fondamentale per la gestione degli accessi condivisi: essa implementa un contatore thread-safe, poiché i suoi metodi sono `synchronized` (quindi si ha la mutua esclusione: un solo client, o meglio, un solo skeleton alla volta può accedere al contatore condiviso).

```
public class Counter implements ServerInterface {
    private int counter = 0;

    public synchronized int reset() {
        counter = 0;
        return counter;
    }
```

```
    public synchronized int increment() {
        counter++;
        return counter;
    }
}
```