Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate

Programmazione Concorrente e Distribuita
Proxy

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
luigi.lavazza@uninsubria.it

Applicazione client/server con socket

- In un'applicazione Client/Server, tipicamente client e server devono implementare meccanismi per:
  - la connessione
  - creazione dei dati da trasmettere
  - l'invio/ricezione delle richieste
  - la ricostruzione dei valori dei dati ricevuti
- Problemi:
  - molti dettagli di implementazione della comunicazione C/S distraggono dalla realizzazione delle funzionalità dell'applicazione;
  - difficoltà di manutenzione e porting: client e server mischiano il codice applicativo a quello per la comunicazione.

Separare logica applicativa e comunicazione

- Client e server devono fare due cose
  - Gestire la comunicazione con la controparte
    - Inclusa le preparazione dei dati da trasmettere e l'interpretazione dei dati ricevuti
  - Implementare la logica applicativa

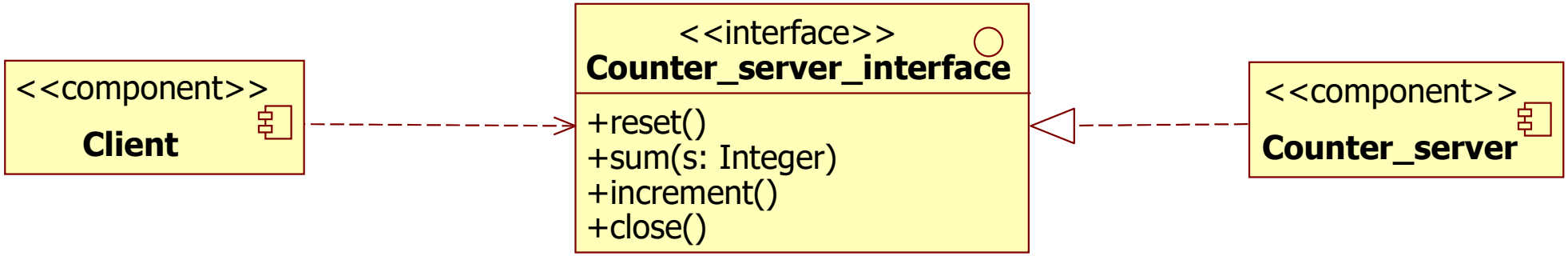
- L'organizzazione dei programmi è più semplice se
  - La parte che implementa la logica applicativa non conosce i dettagli dell'interazione
  - La parte che gestisce l'interazione non deve preoccuparsi di quando e perché inviare o ricevere dati

- **Separiamo** la logica applicativa dai dettagli dei meccanismi di interazione con la controparte.

Esempio: Contatore Remoto

- Il Server gestisce un oggetto Contatore che
  - permette l'inizializzazione o reset di una variabile
  - permette l'incremento della stessa variabile di conteggio
- Il Client
  - Richiede un certo numero di operazioni
  - Calcola il tempo medio impiegato.

Contatore Remoto: vista logica



Contatore Remoto: server

```
import java.io.*;
import java.net.*;
import java.util.StringTokenizer;
public class CounterServer {
  public static final int PORT = 8888;
  ServerSocket serverSocket=null;
  int theCounter;
  CounterServer(){
    try {
      serverSocket = new ServerSocket(PORT);
    } catch (IOException e) {
      System.err.println("Server: ServerSocket failure");
      System.exit(0);
    }
    theCounter=0;
    System.out.println("Started: " + serverSocket);
  }
  public static void main(String[] args) {
    new CounterServer().exec();
  }
```

Contatore Remoto: server

```
private void interpret(String oper) {
   if (oper.equals("<incr>"))
     theCounter++;
   else if (oper.equals("<reset>"))
     theCounter=0;
   else if (oper.startsWith("<sum>")){
     StringTokenizer st = new StringTokenizer(oper);
     String op = st.nextToken();
     String add = st.nextToken();
     if (op.equals("<sum>")){
       theCounter += Integer.parseInt(add);
     }
   } else{
     System.out.println("operation not recognized: " + oper);
   }
 }
```

Contatore Remoto: server

```
private void exec() {
  while (true) {
    System.out.println("Waiting a connection...");
    Socket socket = null;
    try {
      socket = serverSocket.accept();
      BufferedReader istream = new BufferedReader(new
          InputStreamReader(socket.getInputStream()));
      PrintWriter ostream = new PrintWriter(new
          BufferedWriter(new OutputStreamWriter(
              socket.getOutputStream())), true);
      String myOper;
      while ((myOper = istream.readLine())!= null) {
        interpret(myOper);
        ostream.println(theCounter);
      }
    } catch (IOException e) {
      System.err.println("Server: I/O failure"); System.exit(0);
    } finally { try { socket.close();} catch (IOException e) {}}
} } }
```

Contatore Remoto: client

```
import java.net.*;
import java.io.*;
public class CounterClient {
  InetAddress addr;
  CounterClient() throws UnknownHostException {
    addr = InetAddress.getByName(null);
    System.out.println("addr = " + addr);
  }
  public static void main(String[] args) {
    CounterClient cc = null;
    try {
      cc = new CounterClient();
    } catch (UnknownHostException e) {
      System.err.println("Client: no server's IP address");
      System.exit(0);
    }
    try {
      cc.exec();
    } catch (IOException e) {
      System.err.println("Client: I/O error"); System.exit(0);
  }}
```

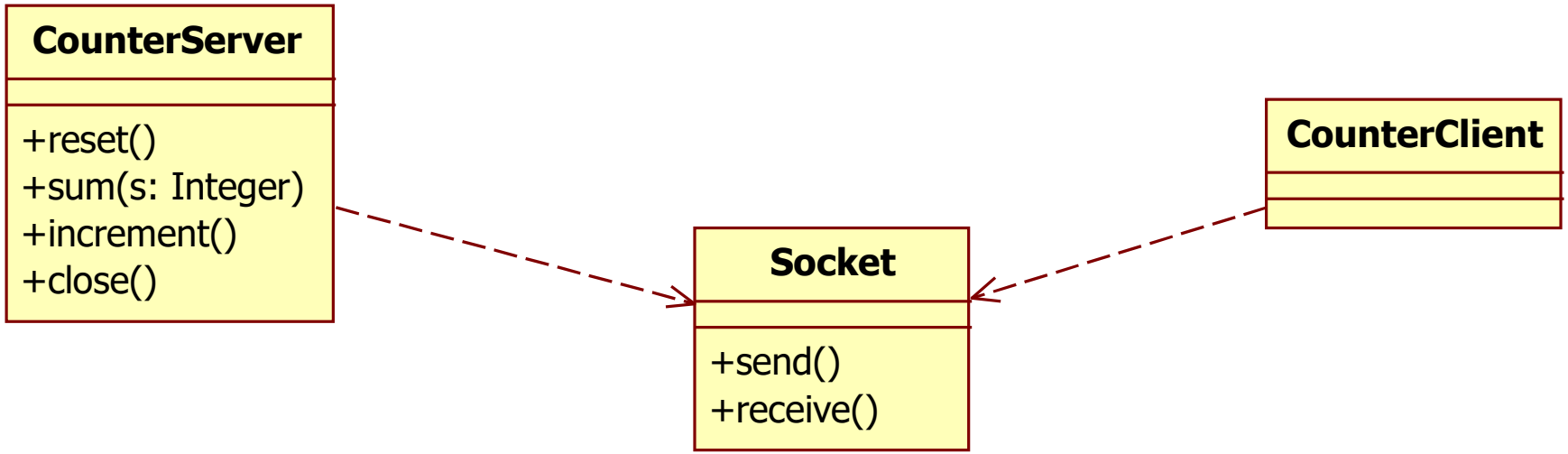
Contatore Remoto: client

```
private void exec() throws IOException {
  Socket socket = new Socket(addr, CounterServer.PORT);
  try {
    System.out.println("socket = " + socket);
    BufferedReader in = new BufferedReader(new
        InputStreamReader(socket.getInputStream()));
    PrintWriter out = new PrintWriter(new BufferedWriter(
        new OutputStreamWriter(socket.getOutputStream())),true);
    out.println("<reset>"); // azzera il contatore
    String strResult = in.readLine();
    System.out.println("reset: "+strResult);
    long startTime = System.currentTimeMillis();
    // Effettuo 1000 richieste di incremento:
    for(int i = 0; i < 1000; i++){
      out.println("<incr>");
      strResult = in.readLine();
      System.out.println("increment: "+strResult);
    }
```

Contatore Remoto: client

```
      //Registro l'istante di conclusione:
      long endTime = System.currentTimeMillis();
      System.out.println("Elapsed time: "+
                         (endTime-startTime)+"ms");
    } finally {
      System.out.println("closing...");
      socket.close();
    }
  }
}
```

Contatore remoto: vista di design

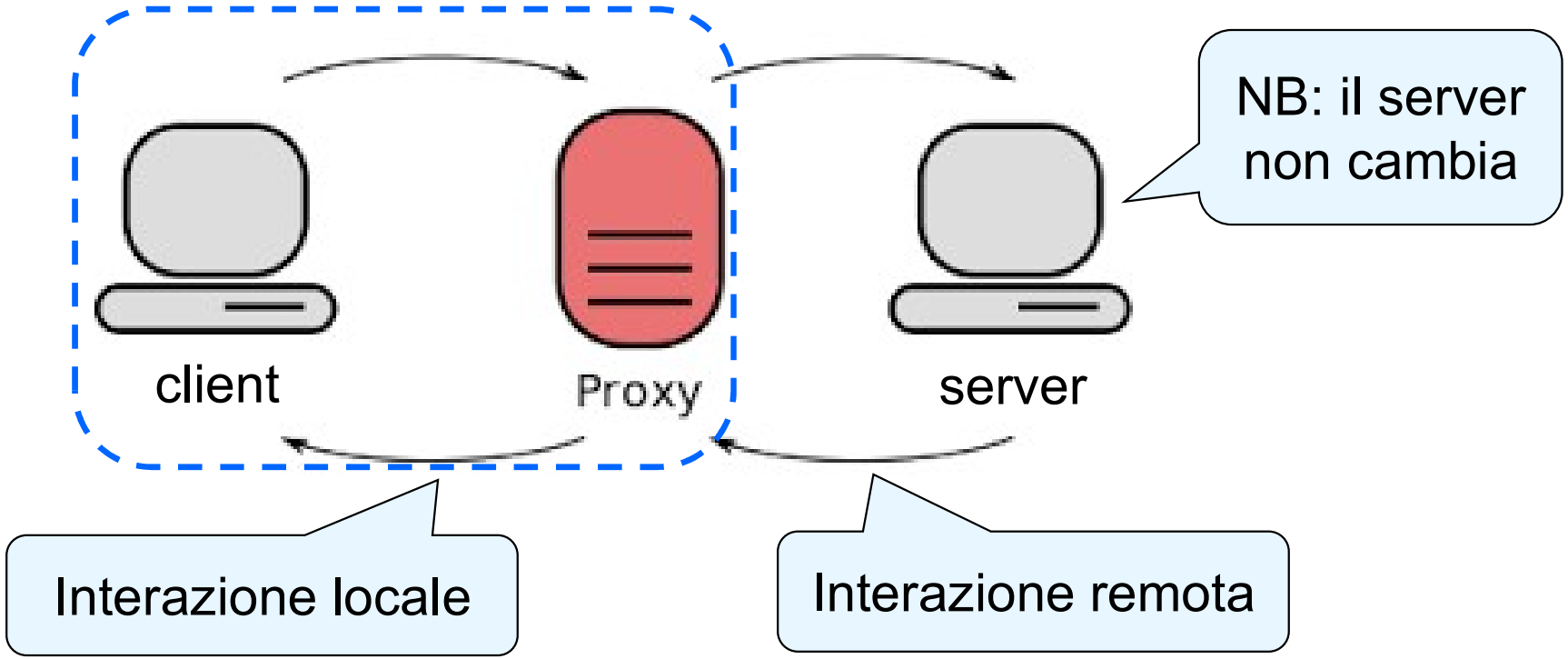


Pattern Proxy lato client

- Il pattern Proxy introduce un ulteriore componente nel modello di interazione client-server.

Pattern Proxy Remoto

- Il Proxy si presenta come una implementazione del servizio remoto, locale al client
  - interfaccia del proxy = interfaccia server reale
- I meccanismi di basso livello necessari per l'instaurazione della comunicazione, e lo scambio dei dati, sono implementati ed incapsulati all'interno del Proxy.



Esempio: Contatore Remoto con ProxyServer

```
public interface ServerInterface {
   public static final int PORT = 8888;

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

Queste sono le funzionalità offerte dal server ...
E quindi anche dal proxy server.

Esempio: Contatore Remoto con ProxyServer

```
public class ProxyServer implements ServerInterface {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  // connessione:
  public ProxyServer() throws Exception {
    InetAddress addr = InetAddress.getByName(null);
    System.out.println("addr = " + addr);
    socket = new Socket(addr, InterfacciaServer.PORT);
    // crea gli stream di input/output
    System.out.println("socket = " + socket);
    in = new BufferedReader(new InputStreamReader(
            socket.getInputStream()));
    out = new PrintWriter(new BufferedWriter(
        new OutputStreamWriter(socket.getOutputStream())), true);
  }
  // metodi dell'interfaccia ServerInterface
  // ...
}
```

Il proxy server stabilisce la connessione col server ...

Esempio: Contatore Remoto con ProxyServer

```
// metodi dell'interfaccia ServerInterface
public int sum(int s) throws IOException {
  out.println("<sum> "+Integer.toString(s));
  String strResult = in.readLine();
  return Integer.parseInt(strResult);
}
public int reset() throws IOException {
  out.println("<reset>");
  String strResult = in.readLine();
  return Integer.parseInt(strResult);
}
public int increment() throws IOException {
  out.println("<incr>");
  String strResult = in.readLine();
  return Integer.parseInt(strResult);
}
public void close() throws IOException {
  System.out.println("closing...");
  out.println("<end>");
  socket.close();
}
```

...
il proxy server inoltra al server tutte le richieste arrivate dal client

Esempio: Contatore Remoto con ProxyServer

```
import java.io.IOException;
```

NB: non c'è nemmeno bisogno delle librerie di rete

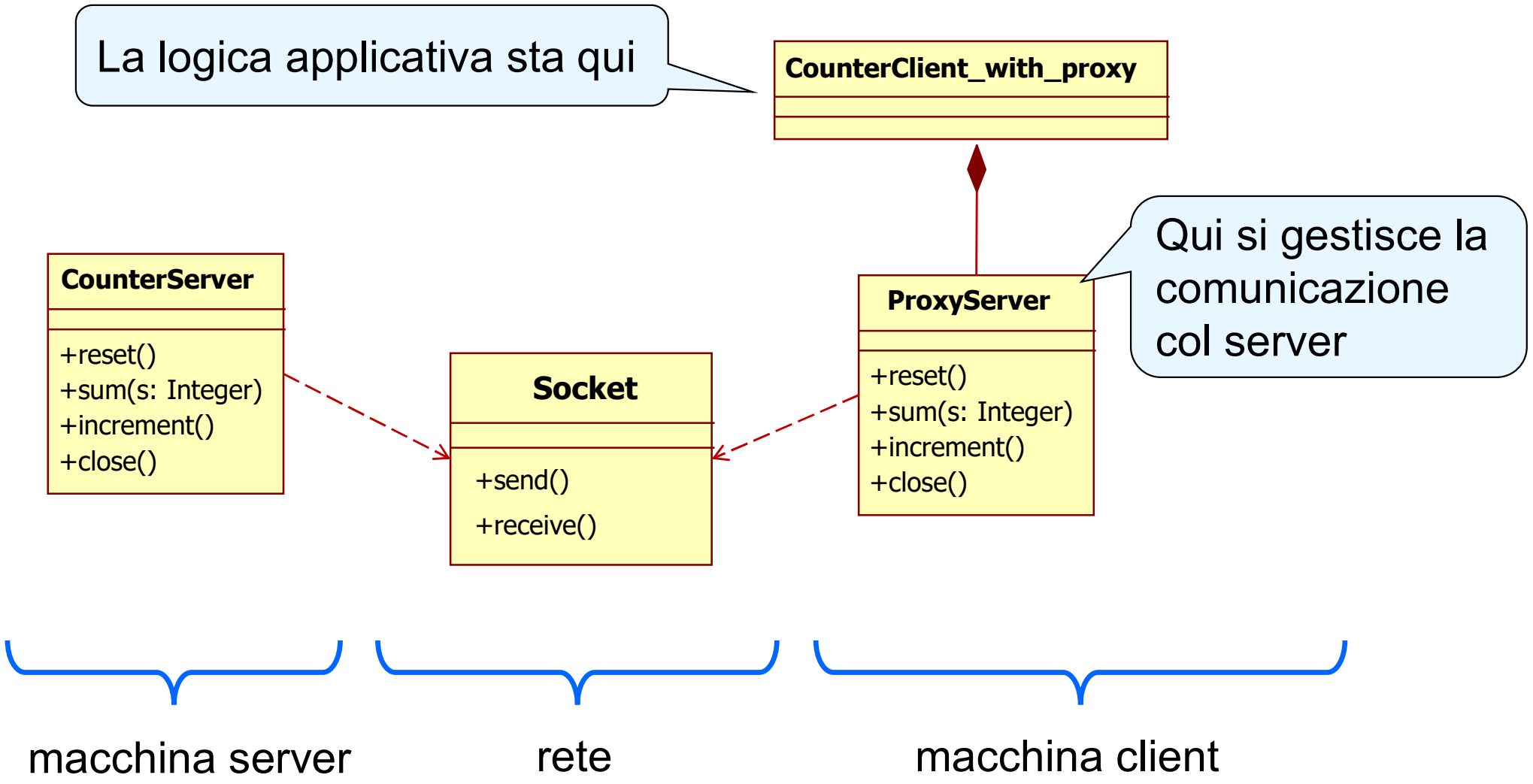
```
public class CounterClient {
  public static void main(String[] args) {
    new CounterClient().exec();
  }
```

Esempio: Contatore Remoto con ProxyServer

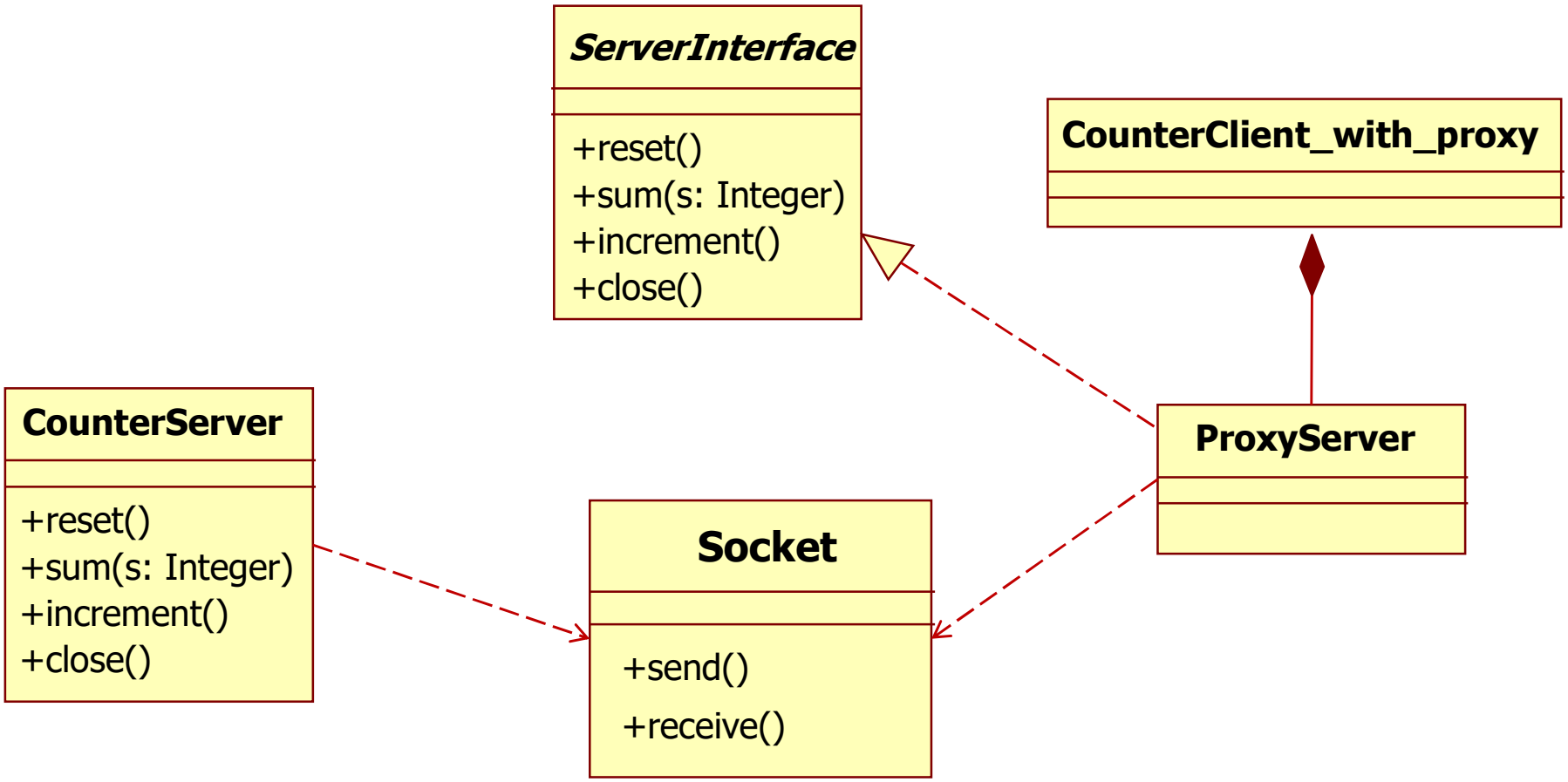
```
  private void exec() {
    ServerInterface server = null;
    try {
      server = new ProxyServer();
      int init = server.reset();
      System.out.println("reset: "+init);
      long startTime = System.currentTimeMillis();
      for(int i = 0; i < 1000; i++){
        int r = server.increment();
        System.out.println("increment: "+r);
      }
      long endTime = System.currentTimeMillis();
      System.out.println("Elapsed: "+(endTime-startTime)+"ms");
    } catch (Exception e) {
      System.err.println("Client: no proxy");
    } finally {  try {
      server.close();
    } catch (IOException e) {}
    }
} }
```

Il client non deve più preoccuparsi della comunicazione: si rivolge a un proxy server locale.

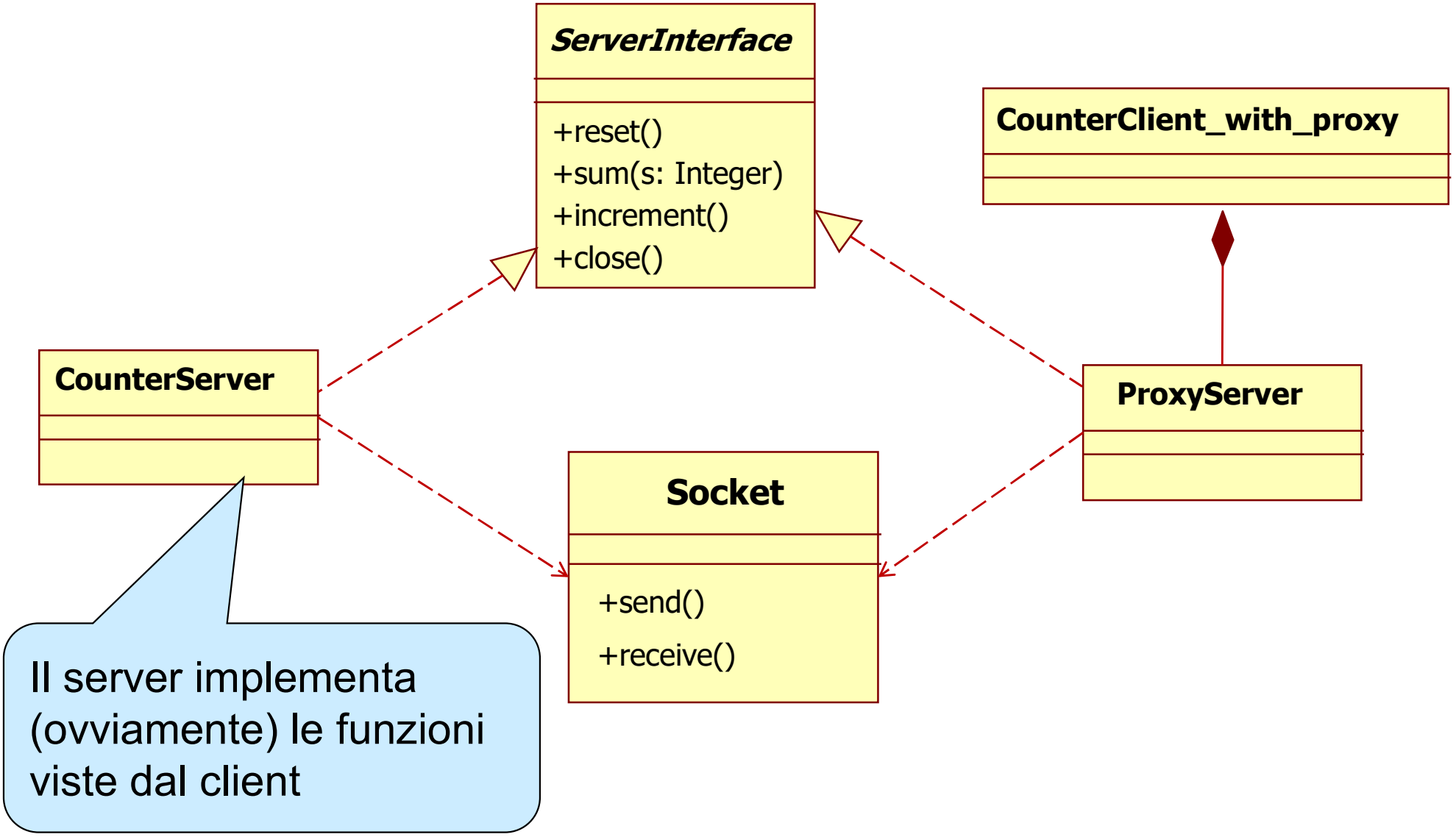
Design diagram



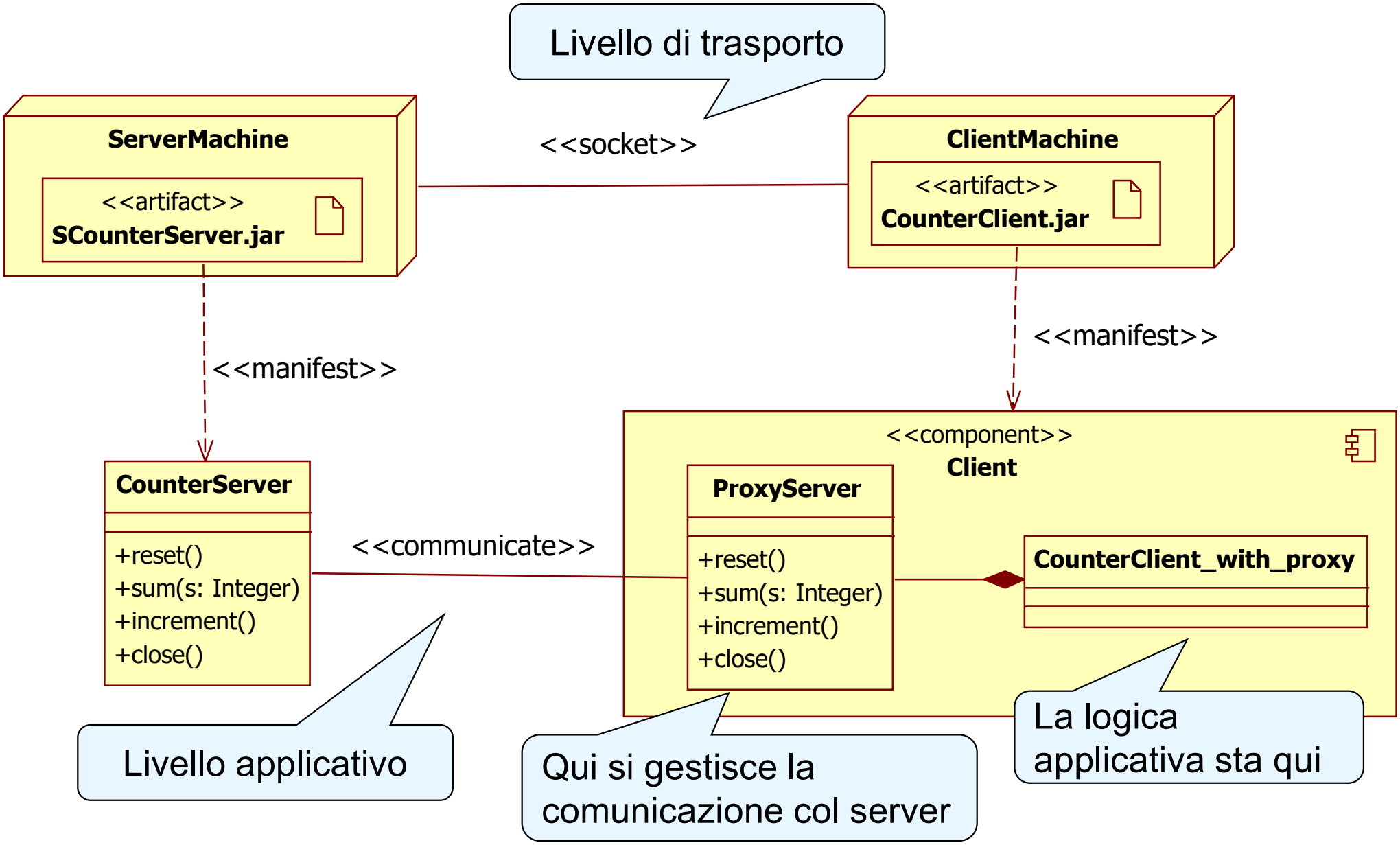
Design diagram (con interface)



Design diagram (con interface)



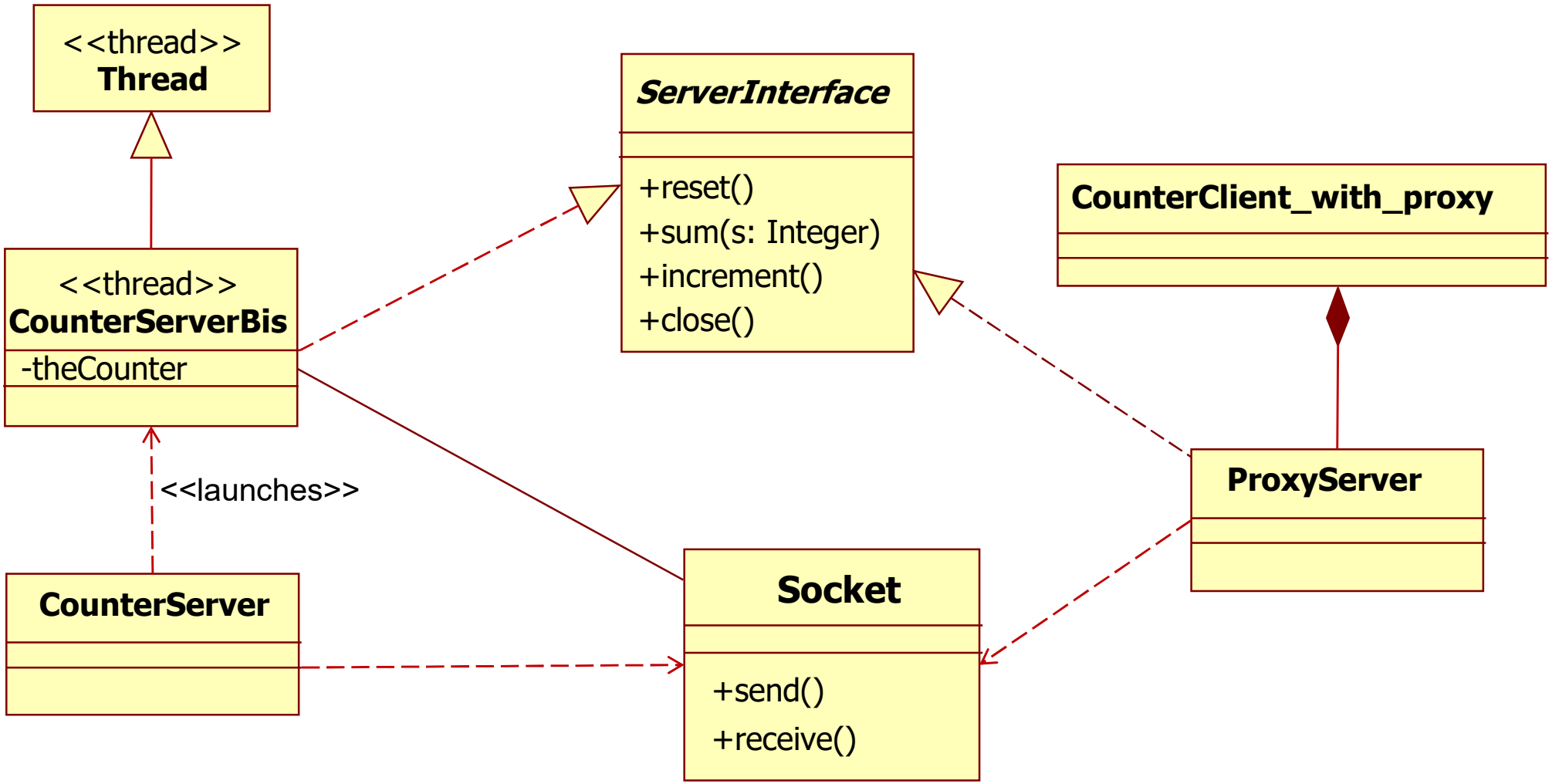
Deployment diagram



Contatore Remoto: server multithread

- Reimplementiamo il server, facendo in modo che
  - Implementi la stessa interfaccia vista dal client
  - Lanci un thread per ogni connessione ricevuta

Design diagram



Esempio

- Realizziamo un contatore remoto con le seguenti caratteristiche:
  - Il server genera un thread dedicato per ciascun client
  - Il client è dotato di proxy server.
  - Per testare il sistema, il main client genera tanti thread client paralleli (ciascuno col suo proxy).

Interface

```
import java.io.IOException;

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

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

Server

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

public class CounterServer {
  ServerSocket serverSocket=null;
  CounterServer() throws IOException{
    serverSocket = new ServerSocket(ServerInterface.PORT);
    System.out.println("Started: " + serverSocket);
  }
  public static void main(String[] args) {
    try {
      new CounterServer().exec();
    } catch (IOException e) {
      System.err.println("Server: server socket  not started");
      System.exit(0);
    }
  }
```

Server

```
private void exec() {
    while (true) {
      System.out.println("Server: waiting a connection...");
      Socket socket;
      try {
        socket = serverSocket.accept();
        System.out.println("Server: new client connected...");
        new Thread(new CounterServerSlave(socket)).start();
      } catch (IOException e) {
        System.err.println("Server: accept failed");
        return;
      }
    }
  }
}
```

Un nuovo thread per ogni client

Server slave

```
import java.net.*;
import java.util.StringTokenizer;
import java.io.*;

public class CounterServerSlave implements ServerInterface,
                                           Runnable {
  private Socket theSocket;
  private int theCounter = 0;
  private BufferedReader istream;
  private PrintWriter ostream;
  public CounterServerSlave(Socket socket) {
    theSocket = socket;
    try {
      istream = new BufferedReader(new InputStreamReader(
          socket.getInputStream()));
      ostream = new PrintWriter(new BufferedWriter(new
          OutputStreamWriter(socket.getOutputStream())), true);
    } catch (IOException e) {  }
  }
```

NB: un contatore per ogni client (no condivisione)

Server slave

```
// metodi dell'interfaccia ServerInterface
public int sum(int s) throws IOException {
  theCounter += s; return theCounter;
}
public int reset() throws IOException {
  theCounter = 0;  return theCounter;
}
public int increment() throws IOException {
  theCounter++;    return theCounter;
}
public void close() {
  System.out.println("closing...");
  try {
    theSocket.close();
  } catch (IOException e) {}
}
```

Server slave

```
public void run() {
  try {
    while (!theSocket.isClosed()) {
      int result = 0;
      String myOper = istream.readLine();
      if (myOper.equals("<incr>"))
        result = increment();
      else if (myOper.equals("<reset>"))
        result = reset();
      else if (myOper.startsWith("<sum>")) {
        StringTokenizer st = new StringTokenizer(myOper);
        String op = st.nextToken();
        String add = st.nextToken();
        if (op.equals("<sum>"))
          result = sum(Integer.parseInt(add));
      } else if (myOper.startsWith("<end>")) { close();
      } else { System.out.println(myOper+" not recognized"); }
      ostream.println(result);
    }
  } catch (Exception e) {}
}
```

Esempio: Contatore Remoto con ProxyServer

```
public class CounterClient {
  public static void main(String[] args) throws Exception {
    int numClients=4;
    for(int i=numClients; i>0; i--) {
      new CounterClientThread(i).start();
      System.out.println("Master client: thread "+i+" created");
      Thread.sleep(2);
    }
  }
}
```

Client thread

```
import java.io.*;
public class CounterClientThread extends Thread {
  private int id;
  public CounterClientThread(int nc){ id=nc; }
  public void run(){
    ServerInterface localServer=null;
    try {
      localServer = new ProxyServer();
      int init = localServer.reset();
      System.out.println("Client "+id+" reset: "+init);
      long startTime = System.currentTimeMillis();
      for(int i = 0; i < 100; i++){
        int r = localServer.sum(1);
        System.out.println("Client "+id+" increment: "+r);
      }
      System.out.println("Client "+id+" Elapsed time: "+
          (System.currentTimeMillis()-startTime)+ "ms");
    } catch (Exception e) { e.printStackTrace();
    } finally {
      System.out.println("Client "+id+" closing...");
      try { localServer.close();} catch (IOException e) { }
} } }
```

Esempio: Contatore Remoto con ProxyServer

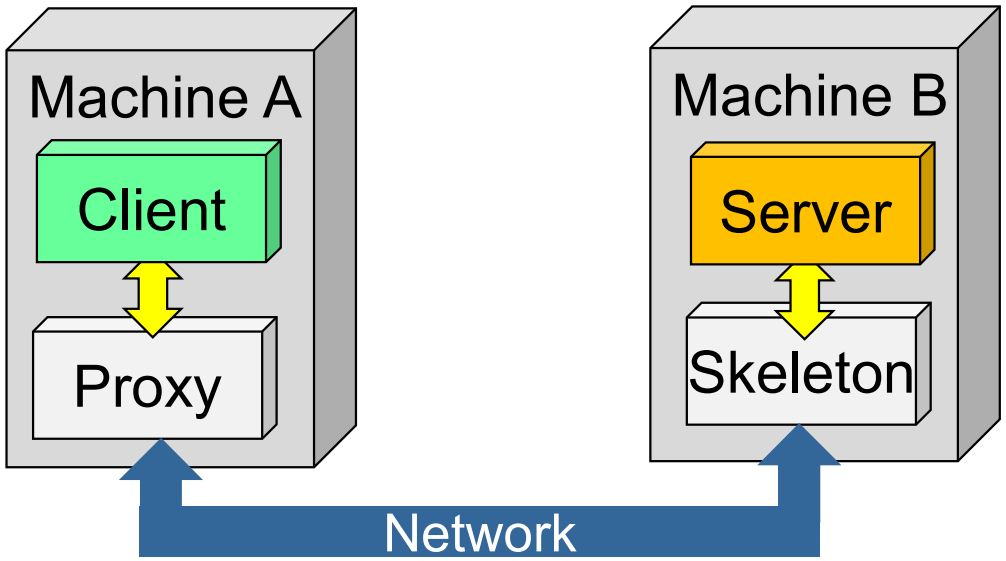
```
public class ProxyServer {
  // come prima
}
```

Pattern Proxy lato server: Skeleton

- Il Proxy lato client “solleva” il client reale dalle problematiche di comunicazione.
- Il server tuttavia ha ancora l’onere di implementare i necessari meccanismi di comunicazione assieme alla logica

Pattern Proxy lato server: Skeleton

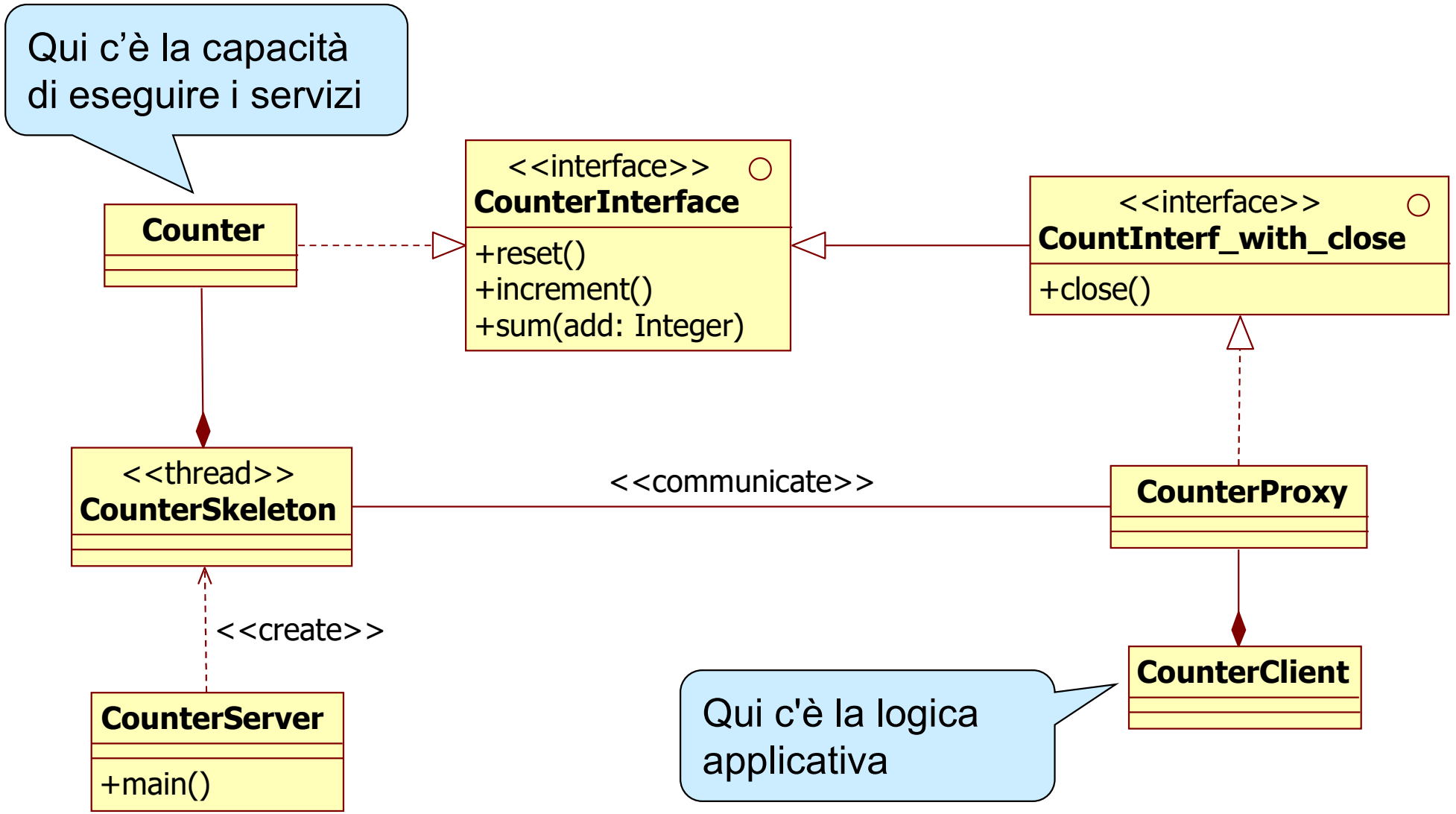
- Aggiungiamo un Proxy lato server, detto Skeleton, che si faccia carico della comunicazione con il Proxy lato client.
- Lo skeleton avrà la responsabilità di
  - ricevere le richieste di servizio
  - strutturare l'informazione fornita in ingresso
  - fare la chiamata al server reale
  - ricevere da questi eventuali risultati e rispedirli al proxy lato client



Implementare lo Skeleton

- Lo skeleton lato server può essere implementato in due modi:
  - Per delega: la classe **`Skeleton`** presenta al suo interno un riferimento al **`CounterServer`**
  - Per ereditarietà : la classe **`Skeleton`** implementa solo gli opportuni schemi di comunicazione, mentre il **`CounterServer`** fornisce l'implementazione ai metodi astratti.

Design diagram (skeleton usa delega)



Esempio: Contatore Remoto con ProxyServer e Skeleton

```
import java.io.*;

public interface CounterInterface {
       public static final int PORT = 8888;
         public int sum(int s) throws IOException;
         public int reset() throws IOException;
         public int increment() throws IOException;
}
```

È ragionevole che il numero di port faccia parte della descrizione del servizio

Non c'è il metodo **`close`**
Lo skeleton ci pensa da solo a chiudere i socket.

Esempio: Contatore Remoto con ProxyServer e Skeleton

```
public class Counter implements CounterInterface {
 private int theCounter;
   public Counter() {
     theCounter = 0;
   }
   public int sum(int s) {
     theCounter += s;
     return theCounter;
   }
   public int reset() {
     theCounter = 0;
     return theCounter;
   }
   public int increment() {
     theCounter++;
     return theCounter;
   }
}
```

La classe **`Counter`** implementa i servizi, senza sapere di far parte di una applicazione distribuita.

Esempio: Contatore Remoto con ProxyServer e Skeleton

```
public class CounterServer {
  void exec() {
    ServerSocket serverSocket;
    Socket clientSocket;
    try {
     serverSocket = new ServerSocket(CounterInterface.PORT);
     System.out.println("Started: " + serverSocket);
     while (true) {
       clientSocket = serverSocket.accept();
       new CounterSkeleton(clientSocket).start();
     }
    } catch (IOException e) {
      System.err.println();
    }
 }
 public static void main(String[] args) {
   new CounterServer().exec();
 }
}
```

Esempio: Contatore Remoto con ProxyServer e Skeleton

```
public class CounterSkeleton extends Thread {
        private Socket theSocket;
        private BufferedReader istream;
        private PrintWriter ostream;
        private Counter server=null;
        public CounterSkeleton(Socket socket) {
                theSocket = socket;
                server = new Counter();
        }
```

Lo skeleton riceve un socket connesso e lo usa per le comunicazioni

Oggetto **`Counter`** locale che eseguirà le richieste di servizi

Esempio: Contatore Remoto con ProxyServer e Skeleton

```
public void run() {
 try {
  istream = new BufferedReader(new InputStreamReader
                                (theSocket.getInputStream()));
  ostream = new PrintWriter(new BufferedWriter(new
    OutputStreamWriter(theSocket.getOutputStream())), true);
  while (!theSocket.isClosed()) {
   int result = 0;
   String myOper = istream.readLine();
   if (myOper.equals("<incr>"))
    result = server.increment();
   else if (myOper.equals("<reset>"))
    result = server.reset();
   else if (myOper.startsWith("<sum>")) {
    StringTokenizer st = new StringTokenizer(myOper);
    String op = st.nextToken(); String add = st.nextToken();
    result = server.sum(Integer.parseInt(add));
   } else if (myOper.startsWith("<end>")) {
    theSocket.close();
   } else { System.out.println("operation not recognized: "+myOper); }
   ostream.println(result);
  }
 } catch (Exception e) { }
}
```

Lo skeleton legge un comando dal socket e chiama il metodo corrispondente del contatore

Esempio: Contatore Remoto con ProxyServer e Skeleton

```
import java.io.IOException;

public interface CounterInterface_with_close extends
                                                CounterInterface{
      public void close() throws IOException;
}
```

Esempio: Contatore Remoto con ProxyServer e Skeleton

```
public class CounterClient {
  void exec() {
    int numClients=4;
    for(int i=numClients; i>0; i--) {
      new CounterClientThread(i).start();
      System.out.println("Master client: thread "+i+" created");
      try { Thread.sleep(2);
      } catch (InterruptedException e) { }
    }
  }
  public static void main(String[] args) throws Exception {
    new CounterClient().exec();
  }
}
```

Esempio: Contatore Remoto con ProxyServer e Skeleton

```
public class CounterClientThread extends Thread {
 private int id;
 CounterInterface_with_close localServer=null;
 CounterClientThread(int i){ id=i; }
 public void run() {
  try {
   localServer = new CounterProxy();
   int init = localServer.reset();
   System.out.println("Client "+id+" reset: "+init);
   long startTime = System.currentTimeMillis();
   for(int i = 0; i < 100; i++){
    int r = localServer.sum(1);
    System.out.println("Client "+id+" increment: "+r);
   }
   System.out.println("Client "+id+" Elapsed time: "+
     (System.currentTimeMillis()-startTime)+ "ms");
  } catch (Exception e) { e.printStackTrace();
  } finally {
    try { localServer.close();} catch (IOException e) { }
} } }
```

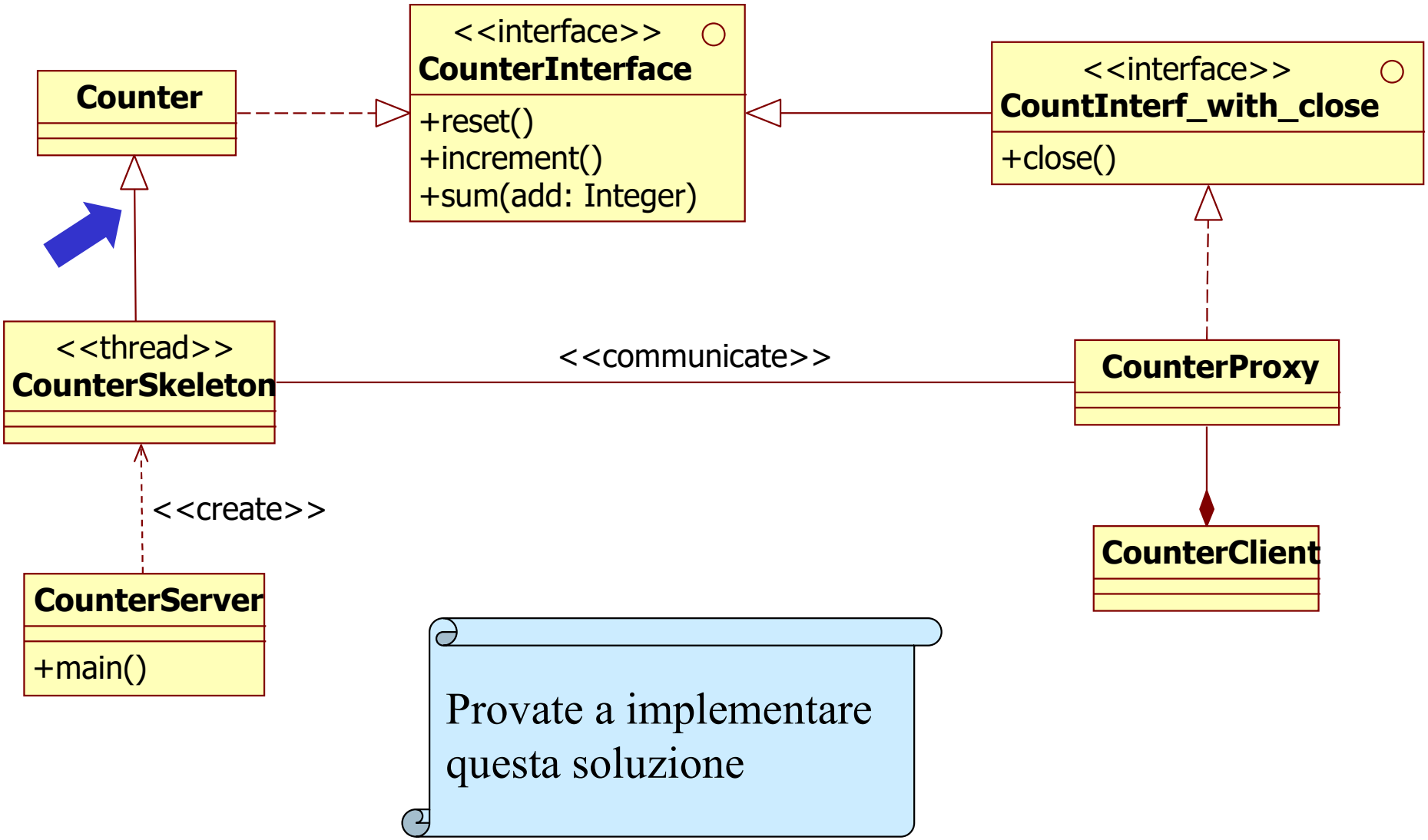
Esempio: Contatore Remoto con ProxyServer e Skeleton

```
public class CounterProxy implements
CounterInterface_with_close{
 private Socket socket;
 private BufferedReader in;
 private PrintWriter out;
 public CounterProxy() throws Exception {
  InetAddress addr = InetAddress.getByName(null);
  System.out.println("addr = " + addr);
  socket = new Socket(addr, CounterInterface.PORT);
  System.out.println("socket = " + socket);
  in = new BufferedReader(new
           InputStreamReader(socket.getInputStream()));
  out = new PrintWriter(new BufferedWriter(new
           OutputStreamWriter(socket.getOutputStream())), true);
 }
```

Esempio: Contatore Remoto con ProxyServer e Skeleton

```
// metodi dell'interfaccia ServerInterface
public int sum(int s) throws IOException {
 out.println("<sum> "+Integer.toString(s));
 String strResult = in.readLine();
 return Integer.parseInt(strResult);
}
public int reset() throws IOException {
 out.println("<reset>");
 String strResult = in.readLine();
 return Integer.parseInt(strResult);
}
public int increment() throws IOException {
 out.println("<incr>");
 String strResult = in.readLine();
 return Integer.parseInt(strResult);
}
public void close() throws IOException {
 System.out.println("closing...");
 out.println("<end>");
 socket.close();
}}
```

Design diagram (skeleton usa delega)



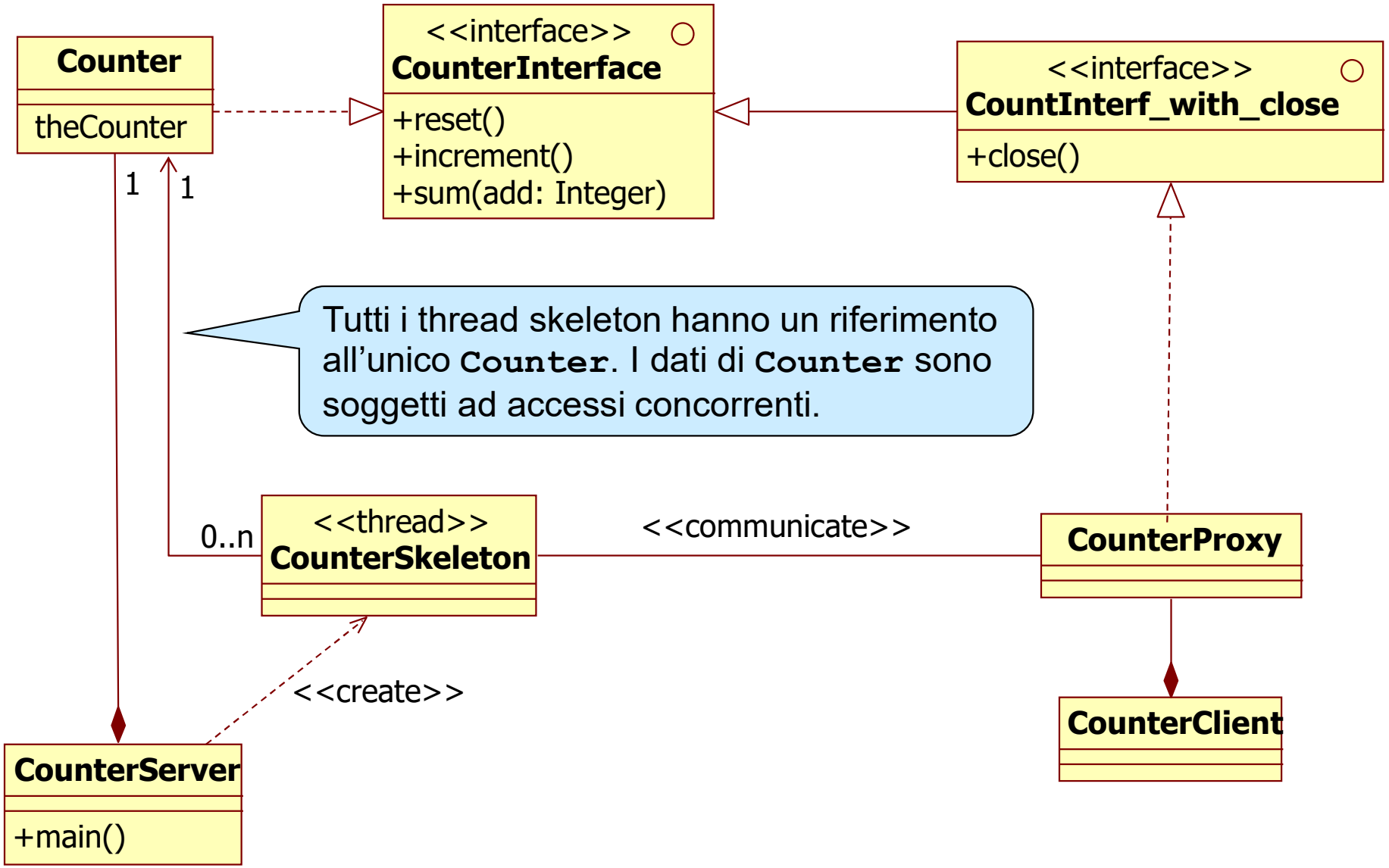
Accesso concorrente a oggetti remoti

- Nell'esempio del contatore remoto avevamo che ogni client chiedeva al server la creazione di un proprio contatore privato.
- In questo caso evidentemente non ci sono problemi di concorrenza.
- Vediamo adesso come si gestisce un server che permette l'accesso concorrente ad un *oggetto condiviso da diversi client*.
- Ci sono le solite problematiche:
  - Corse critiche, da evitare attraverso mutua esclusione
  - ...

Esempio: Contatore Remoto condiviso

- Vediamo l'esempio del contatore remoto, ma questa volta il contatore è condiviso tra tutti i client.

Design diagram (skeleton usa delega)



Contatore thread-safe

```
public class Counter implements CounterInterface {
  private int theCounter;
    public Counter() {
      theCounter = 0;
    }
    public synchronized int reset() {
      theCounter = 0;
      return theCounter;
    }
    public synchronized int increment() {
      theCounter++;
      return theCounter;
    }
    public synchronized int sum(int s) {
      theCounter += s;
      return theCounter;
    }
}
```

I metodi sono synchronized: un solo client alla volta accede al contatore condiviso.

Server

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

public class CounterServer {
  Counter theCounter;
  CounterServer(){
    theCounter=new Counter();
  }
  public static void main(String[] args) {
    new CounterServer().exec();
  }
}
```

L'unica istanza di **Counter** è creata dal server.

Server

```
void exec() {
  ServerSocket serverSocket;
  Socket clientSocket;
  try {
    serverSocket = new ServerSocket(CounterInterface.PORT);
    System.out.println("Started: " + serverSocket);
    while (true) {
      System.out.println("Master waiting a connection...");
      clientSocket = serverSocket.accept();
      System.out.println("Master connected with "+clientSocket);
      new CounterSkeleton(clientSocket, theCounter).start();
    }
  } catch (IOException e) {
    System.err.println();
  }
}
```

Per ogni client connesso si crea uno skeleton, passandogli il contatore condiviso e il socket connesso.

Skeleton

```
public class CounterSkeleton extends Thread {
  private Socket theSocket;
  private BufferedReader istream;
  private PrintWriter ostream;
  private Counter server;
  public CounterSkeleton(Socket socket, Counter cnt) {
    theSocket = socket;
    server=cnt;
  }
  public void run() {
  // come prima
}
```

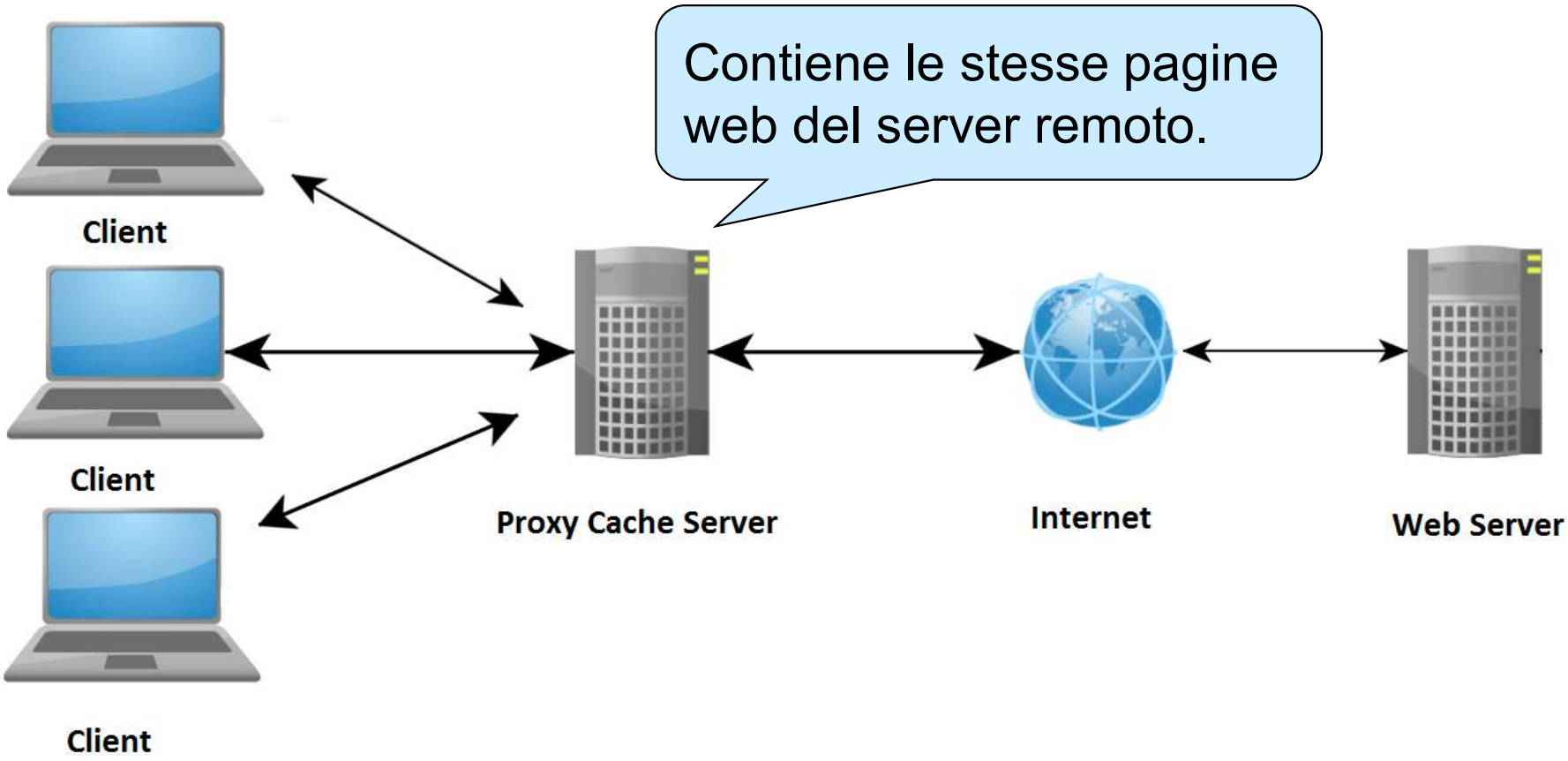
Riceve il contatore condiviso.

Cosa abbiamo visto

- Il pattern proxy: un modo per implementare client o server, separando logica applicative e gestione delle comunicazioni.

- Spesso, col termine «proxy» si intende
  - una macchina diversa dal client e dal server
  - Che implementa gli stessi servizi del server (o un sottoinsieme)

Una macchina proxy



- Il proxy fornisce lo stesso servizio del server, con dei vantaggi
  - Ad es., la velocità di un accesso su rete locale invece che su rete geografica