Tcp tutorial

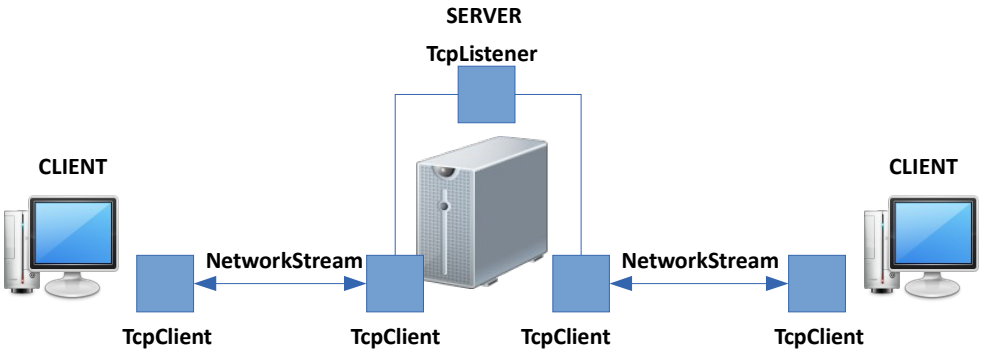
Comunicazione client-server via protocollo TCP

Anno 2017/2018

Indice generale

1 Comunicazione client-server

Una comunicazione via TCP avviene tra un (processo) *server* e uno o più (processi) *client*. Il client è quello che *avvia una richiesta di connessione*. Il server attende l'arrivo di una richiesta da parte di un client (un singolo server può servire le richieste di molti client).



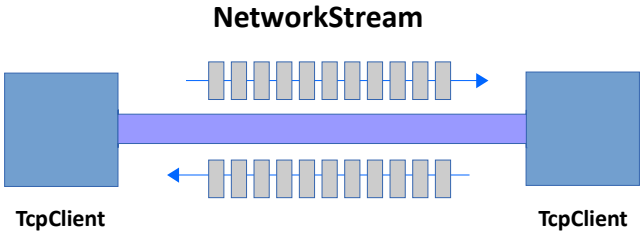
Il server si mette in ascolto a una porta, in attesa di ricevere una richiesta di connessione da parte di un client. Quando la riceve, stabilisce una connessione mediante un oggetto `TcpClient`. Da questo momento i due oggetti `TcpClient` sono connessi e possono comunicare tra loro utilizzando un `NetworkStream`, nel quale scrivere/leggere i dati.

Al netto di errori o problemi hardware, la connessione viene chiusa e la comunicazione interrotta quando uno dei due oggetti esegue il metodo `Close()`.

1.1 Comunicazione via NetworkStream

La comunicazione TCP è orientata alla connessione e al flusso di dati, diversamente da quella UDP, nella quale due oggetti si inviano dei pacchetti.

La connessione tra due oggetti `TcpClient` può essere vista come un canale di comunicazione, nel quale è possibile far viaggiare un flusso di byte mediante un `NetworkStream`.



Trattandosi di un flusso (*stream*, appunto) non esiste un "confine" tra un messaggio e il successivo (come avviene in UDP); è responsabilità di client e server interpretare i dati in accordo a un protocollo condiviso.

2 Client: uso della classe TcpClient

La classe `TcpClient` consente di gestire una comunicazione TCP tra due processi, in esecuzione su computer distinti o sullo stesso computer. Dal lato client la comunicazione si svolge normalmente nel seguente modo:

connessione al server

comunicazione: scambio dati con il server

chiusura della connessione.

2.1 Connessione al server

Per connettersi si usa il metodo `Connect()`, specificando l'*endpoint* (indirizzo IP+porta) del server:

```
TcpClient cli = new TcpClient();
cli.Connect("127.0.0.1", 8080);
```

`Connect()` esiste in più versioni, alcune che accettano valori `IPAddress` o `IPEndPoint`.

2.2 Lettura/Scrittura dati

Le operazioni di lettura/scrittura possono essere eseguite utilizzando un `NetworkStream`, ottenuto dall'oggetto `TcpClient` mediante il metodo `GetStream()`.

```
TcpClient cli = new TcpClient();
cli.Connect("127.0.0.1", 8080);
NetworkStream ns = cli.GetStream();
```

Il `NetworkStream` può essere utilizzato sia per operazioni di lettura che di scrittura.

2.2.1 Scrittura dati: Write()

Il metodo `Write()` scrive un vettore di byte nello *stream*. Il seguente codice scrive la stringa `“hello!”`:

```
TcpClient cli = new TcpClient();
cli.Connect("127.0.0.1", 8080);

byte[] data = Encoding.UTF8.GetBytes("hello!");    // codifica la stringa in byte

NetworkStream ns = cli.GetStream();
ns.Write(data, 0, data.Length);
```

Il metodo è “bloccante”: l'esecuzione ha termine soltanto dopo che i dati sono stati scritti oppure si è verificato un errore.

2.2.2 Lettura dei dati: Read()

Il metodo `Read()` legge un flusso di byte, memorizzandoli su un vettore precedentemente creato. Richiede il numero di byte da leggere e ritorna il numero di byte effettivamente letti.

```
cli.Connect("127.0.0.1", 8080);
NetworkStream ns = cli.GetStream();

byte[] data = new byte[32];            // il vettore deve già esistere
int byteCount = ns.Read(data, 0, 32);  // prova a leggere 32 byte e li memorizza in data
```

Il metodo è bloccante: attende che vi sia almeno un byte da leggere. La variabile `byteCount` contiene il numero di byte effettivamente letti; se vale zero significa che il metodo è terminato senza leggere alcun byte.

2.2.3 Lettura di un singolo byte alla volta

Il metodo `ReadByte()` legge un solo byte. Può essere opportuno usarlo in combinazione con la proprietà `DataAvailable`, così da eseguirlo soltanto se ci sono dati da leggere:

```
static byte[] Receive(NetworkStream ns)
{
    List<byte> list = new List<byte>();
    while(ns.DataAvailable)
    {
        byte b = (byte)ns.ReadByte(); // serve un cast, poiché ReadByte() torna un int
        list.Add(b);
    }
    return list.ToArray();
}
```

Nota bene: il metodo `Receive()`, così implementato, non blocca l'esecuzione se non ci sono dati da leggere.

2.3 Utilizzo di Writer e Reader su NetworkStream

Non è comodo usare direttamente i metodi `Read()` e `Write()` della classe `NetworkStream`, poiché essi lavorano su dei vettori di byte; in generale, si vuole leggere/scrivere i dati direttamente in formato, testo, binario o XML. A questo scopo si usano le classi:

- `StreamWriter`, `StreamReader`: per scrivere e leggere testo;
- `BinaryWriter`, `BinaryReader`: per scrivere e leggere dati binari;
- `XmlTextWriter`, `XmlTextReader`: per scrivere e leggere XML;

Ad esempio, il seguente codice scrive un elenco di stringhe sullo *stream*:

```
string[] lineList = { "Rossi, Andrea", "Bianchi, Giacomo", "Verdi Francesca" };
TcpClient client = new TcpClient();
client.Connect("127.0.0.1", 8080);
```

```
var ns = client.GetStream();

StreamWriter sw = new StreamWriter(ns);
foreach (var line in lineList)
{
    sw.WriteLine(line);   //scrive: stringa + \r\n
}
sw.Close();               //chiude anche il NetworkStream!
```

Analogamente, potrebbe essere utilizzato un `BinaryWriter`:

```
string[] lineList = { "Rossi, Andrea", "Bianchi, Giacomo", "Verdi Francesca" };

TcpClient client = new TcpClient();
client.Connect("127.0.0.1", 8080);
var ns = client.GetStream();

BinaryWriter bw = new BinaryWriter(ns);
foreach (var line in lineList)
{
    bw.Write(line);  //scrive: lunghezza stringa + stringa
}
bw.Close();          //chiude anche il NetworkStream!
```

Il flusso di byte generato dai due tipi di *writer* è leggermente diverso, dunque è necessario che il server legga i dati utilizzando l'oggetto giusto, `StreamReader` o `BinaryReader`.

2.3.1 Chiusura del Writer/Reader

La chiusura di un *writer/reader* causa la chiusura dello *stream* sottostante; questo può essere un problema se si tratta di un `NetworkStream`, poiché significa interrompere la comunicazione.

La scelta di chiudere il *writer/reader* dopo il suo utilizzo dipende dal tipo di comunicazione che si svolge tra client e server. In generale si dovrebbe chiudere la comunicazione utilizzando il metodo `Close()` dell'oggetto `TcpClient`. (Vedi: **Chiusura della connessione**.)

3 Server: connessione di un client

Un server resta in attesa che un client tenti di connettersi. Il processo di attesa e l'accettazione della richiesta del client vengono gestiti da un oggetto di tipo `TcpListener`.

Attesa di un tentativo di connessione dal client

Creazione di un TcpClient per comunicare con il client

comunicazione: scambio dati con il client

chiusura della connessione.

3.1 Connessione con il client: TcpListener

Un `TcpListener` avvia un processo di ascolto che attende una connessione da parte di un client. Il codice seguente mostra il tipico uso di un `TcpListener`:

```
TcpListener listener = new TcpListener(IPAddress.Any, 8080); // ascolta sulla porta 8080

listener.Start();
while(true)
{
    TcpClient tcp = listener.AcceptTcpClient();   // blocca in attesa del client
    // ... avvia comunicazione con il client
}
listener.Stop();          // ferma listener (con questa implementazione è inutile)
```

Dopo essere stato creato, il *listener* viene avviato mediante il metodo `Start()`. Dopodiché attende la richiesta di connessione da parte di un client mediante il metodo `AcceptTcpClient()`. Quando arriva una richiesta, il metodo crea un oggetto `TcpClient` che possa gestirla. Dopodiché il *listener* si mette in attesa di una nuova richiesta.

Il codice precedente mostra che un server può gestire la comunicazione con un numero qualsiasi di client, poiché per ognuno di essi crea un `TcpClient`.

3.2 Comunicazione con il client: TcpClient ↔ TcpClient

Il codice di comunicazione è del tutto identico a quello utilizzato nel client, poiché usa lo stesso tipo di oggetto. Il seguente metodo mostra come leggere un elenco di stringhe spedito dal client:

```
static void ReadLinesFromClient(TcpClient tcp)
{
    var ns = tcp.GetStream();
    StreamReader sr = new StreamReader(ns);
    string s = sr.ReadLine();
    while (s != null)        //termina quando il client chiude lo stream
    {
        Console.WriteLine(s);
        s = sr.ReadLine();
```

```
    }
}
```

Nota bene: nell'esempio, la condizione di terminazione del ciclo implica che il client chiuda lo stream; in caso contrario il ciclo non terminerà mai. Client e server si devono accordare su un protocollo che permetta al server di capire che l'elenco è terminato.

4 Chiusura / interruzione della connessione

La comunicazione tra due oggetti `TcpClient` può procedere fin quando la connessione resta attiva e il `NetworkStream` aperto. Al netto di problemi di connettività, è responsabilità di client e server mantenere la connessione attiva fintantoché è necessario.

4.1 Chiusura della connessione

Dopo che la comunicazione tra client e server è terminata, è opportuno chiudere la connessione eseguendo il metodo `Close()` sugli oggetti `TcpClient`. Ciò libera risorse del sistema operativo, compresa la porta occupata dai `TcpClient`, e chiude il `NetworkStream`, impedendo ulteriori scambi di dati.

4.1.1 Chiusura di StreamReader e StreamWriter

Non è necessaria, poiché entrambi gli oggetti non occupano risorse del sistema operativo.

4.1.2 Lettura/scrittura su una connessione già chiusa

Esistono due scenari.

Scrittura/lettura dopo aver chiuso la connessione

Eseguire un metodo di lettura/scrittura dopo aver chiuso il `TcpClient` (o il `NetworkStream`) produce un'eccezione.

Scrittura/lettura su una connessione chiusa "dall'altro lato"

Di norma, il tentativo di utilizzare una connessione che è stata chiusa dall'altro lato non produce errori: i metodi di lettura/scrittura non eseguono alcuna operazione e ritornano immediatamente. Esiste però un'eccezione, descritta dal seguente scenario.

A spedisce dei dati a B, che però non li legge e chiude la connessione. Se A tenta di leggere/scrivere dei dati viene sollevato un errore.

Ciò accade perché A "vede" la chiusura della connessione di B come un'interruzione anomala, poiché gli ha inviato dei dati che B non ha letto. (Vedi paragrafo successivo.)

4.2 Interruzione della connessione

Una connessione può essere interrotta per problemi di hardware e connettività, o per la terminazione di server e/o client.

Il tentativo di scrivere/leggere su una connessione interrotta provoca l'errore `IOException`. Attraverso la proprietà `InnerException` si può accedere all'oggetto eccezione che ha prodotto l'errore (tipicamente `IOException`, ma anche `SocketException`).

4.2.1 Proteggere il codice da errori

Il codice di lettura/scrittura su una connessione TCP dovrebbe essere protetto da errori. Dovrebbe essere garantito:

- La chiusura dell'oggetto `TcpClient`: uso di `try...finally` (o `using`).
- La protezione del processo dal crash: uso di `try...catch`.

Si tratta di un aspetto particolarmente importante nel server, poiché questo gestisce più connessioni: gli errori possono accumularsi e impattare sulla sua capacità di fornire i propri servizi.

4.2.2 Garantire il rilascio dello oggetto TcpClient

Per garantire la chiusura del `TcpClient` basta utilizzare l'oggetto all'interno di un costrutto `using`:

```
using (TcpClient client = new TcpClient())
{
    client.Connect(server, porta);
    var ns = client.GetStream();
    //... scrivi/leggi sullo stream

}   // chiude automaticamente il TcpClient (e lo stream) anche in caso di errore.
```

`using` equivale al costrutto `try...finally`: garantisce la chiusura dell'oggetto indipendentemente dal fatto che si sia verificata un'eccezione oppure no.

4.2.3 Gestire l'eccezione

Il codice precedente non gestisce l'eccezione, dunque non proteggere il processo dal crash. Di seguito mostro come gestire gestisce eventuali errori e garantire la chiusura del `TcpClient`:

```
static void ReadLinesFromClient(TcpClient tcp)
{
    try
    {
        var ns = tcp.GetStream();
        StreamReader sr = new StreamReader(ns);
        string s = sr.ReadLine();
        while (s != null)
        {
            Console.WriteLine(s);
            s = sr.ReadLine();
        }
    }
    catch(IOException e)  //gestisce l'eccezione
    {
        Console.WriteLine("ERRORE: {0}", e.Message);
        Console.WriteLine("Comunicazione con : {0}", tcp.Client.RemoteEndPoint);
    }
    finally               // chiude la connessione, con o senza errori
    {
        tcp.Close();
    }
}
```

Nota bene: la proprietà `Client` di un `TcpClient` referenzia l'oggetto `Socket` che implementa la comunicazione TCP (implementa anche quella UDP). La proprietà `RemoteEndPoint` identifica l'*endpoint* del client.

5 Gestione multitask della comunicazione

Il codice mostrato finora utilizza metodi bloccanti[1]. Soprattutto lato server, l'esecuzione di metodi bloccanti è accettabile soltanto se applicata contemporaneamente al multitask.

Considera il seguente frammento di codice:

```
TcpListener listener = new TcpListener(IPAddress.Any, 8080); // ascolta sulla porta 8080
listener.Start();
while(true)
{
    TcpClient tcp = listener.AcceptTcpClient();   // blocca in attesa del client
    // ... avvia comunicazione con il client
}
```

La parte evidenziata suppone l'esecuzione di metodi di lettura/scrittura sul `NetworkStream`: l'esecuzione del ciclo è bloccata e dunque il *listener* non può accettare la richiesta di un nuovo client. In pratica, un server simile potrebbe gestire soltanto un client per volta.

Si tratta soltanto di un esempio. Lo stesso codice che esegue il *listener* è bloccante e dunque, in alcuni scenari, dovrebbe essere eseguito in multitask.

5.1 Server: gestione multitask della comunicazione con il client

Per ogni client che viene gestito dal server è opportuno create un task separato.

```
TcpListener listener = new TcpListener(IPAddress.Any, 8080);
listener.Start();
while(true)
{
    TcpClient tcp = listener.AcceptTcpClient();
    Task.Run(() => ReadFromClient(tcp));
}
```

In questo modo il server può gestire i dati inviati da un numero qualunque di client.

5.1.1 Gestione multitask del listener

In base al tipo di applicazione, anche il *listener* può essere gestito in un task separato:

```
static void Main(string[] args)
{
    Task.Run(() => Listen(8080));
    Console.ReadKey();
}

static void Listen(int port)
{
    TcpListener listener = new TcpListener(IPAddress.Any, port);
    listener.Start();
```

1 Di questi metodi esistono anche le versioni asincrone, non bloccanti.

```
    while (true)
    {
        TcpClient tcp = listener.AcceptTcpClient();
        Task.Run(() => ReadFromTclient(tcp));
    }
}
```

5.1.2 Implementare la chiusura del listener

Come è stato implementato finora, il ciclo di gestione del *listener* non può terminare. Una soluzione è quella di usare una variabile `bool` come condizione del ciclo ed evitare che il *listener* resti bloccato sul metodo `AcceptToClient()`:

```
static bool stopListen = false;
static void Main(string[] args)
{
    TcpListener listener = new TcpListener(IPAddress.Any, port);
    listener.Start();
    while (!stopListen)
    {
        if (listener.Pending())
        {
            TcpClient client = listener.AcceptTcpClient();
            Task.Run(() => ReadFromClient(client));
        }
        else
            Thread.Sleep(20);  // rilascia la CPU agli altri thread
    }
    listener.Stop();
}
```

Il metodo `Pending()` ritorna `true` soltanto se c'è una richiesta di connessione; ciò evita di bloccare l'esecuzione in attesa di una richiesta. Quando si desidera terminare il *listener* basta impostare a `true` la variabile `stopListen`.

6 Demo di una applicazione completa

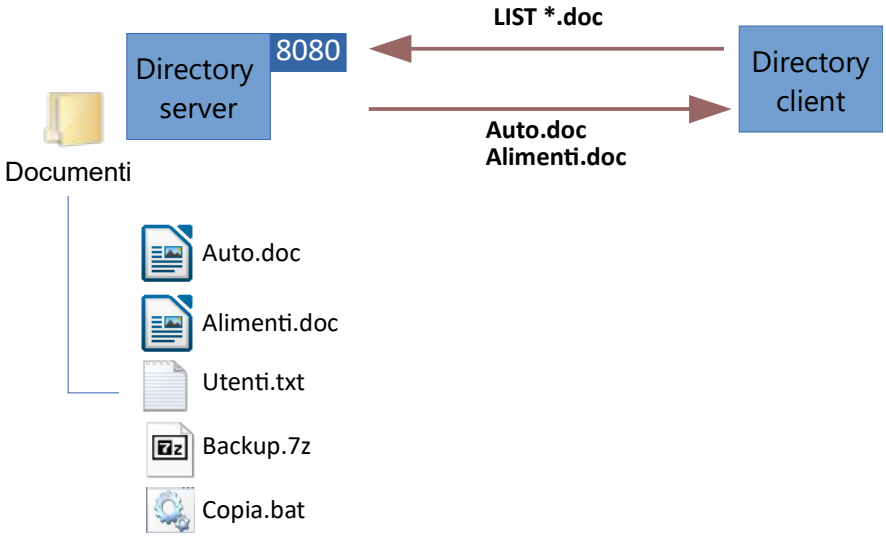
Si desidera realizzare un "*directory server*" che usi il protocollo TCP per rispondere alle richieste sui file contenuti in una cartella.

Il processo deve monitorare una cartella, accettando richieste sulla porta 8080. Ogni richiesta è caratterizzata dal comando:

LIST <pattern>$_{opz}$

dove *pattern* indica una stringa di ricerca contenente dei caratteri jolly. Se non viene specificato il pattern, si intende essere `*.*` (elenca tutti i file).

Il seguente schema mostra uno scenario ipotetico. La cartella gestita è Documenti; il comando inviato dal client è `LIST *.doc`.



6.1 Protocollo client-server

6.1.1 Client → server

Una richiesta del client è rappresentata da una linea di testo comprensiva di *newline* (`\r\n`). Nell'attuale versione, l'unico comando implementato è LIST. Ad esempio, il comando:

`LIST *.jpg`

Richiede l'elenco dei file immagine in formato JPEG.

6.1.2 Server → client

Il server risponde al comando LIST con un elenco di righe, ognuna delle quali contiene il nome di un file, seguito da *newline*. Ad esempio:

- client invia: **`LIST *.doc\r\n`**
- server risponde: **`Auto.doc\r\nAlimenti.doc\r\n\r\n`**

Se il client invia un comando sconosciuto, il server risponde con

ERROR\r\n

6.2 Implementazione del server

Il server utilizza un *listener* per gestire le connessioni dei client; ogni client viene gestito in un task separato. La comunicazione termina quando la connessione viene chiusa dal client.

```
static int port = 8080;
static string folder = Environment.GetFolderPath(Environment.SpecialFolder.Personal);

static void Main(string[] args)
{
    TcpListener listener = new TcpListener(IPAddress.Any, port);
    listener.Start();
    while (true)
    {
        TcpClient client = listener.AcceptTcpClient();
        Console.WriteLine("Connessione accettata: {0}", client.Client.RemoteEndPoint);
        Task.Run(() => ResponseToClient(client));
    }
}
```

6.2.1 Gestione del client

Una volta accettata la connessione, il server entra in un ciclo di lettura delle richieste inviate dal client; ciclo che termina quando il client chiude la connessione.

```
private static void ResponseToClient(TcpClient client)
{
    var ns = client.GetStream();
    while (true)        // è il client a decidere di terminare la comunicazione
    {
        StreamReader sr = new StreamReader(ns);

        // se client interrompe connessione, questo codice solleva un'eccezione!
        string cmd = sr.ReadLine();
        if (cmd == null)  // il client ha chiuso la connessione
            break;

        StreamWriter sw = new StreamWriter(ns);
        string[] fileList = ExecuteCommand(cmd);
        if (fileList != null)
        {
            foreach (var fileName in fileList)
            {
                sw.WriteLine(fileName);
            }
        else
        {
```

```
            sw.WriteLine("ERROR");
        }
        sw.WriteLine(); // termina elenco con linea vuota
        sw.Flush();
    }
}
```

Nota bene: il codice non è protetto da errori. Se il client viene interrotto, l'istruzione di lettura dallo *stream* solleverà un'eccezione.

Il metodo `ExecuteCommand()` ritorna l'elenco dei file che rispettano il pattern specificato. Il metodo analizza la validità del comando ricevuto e ritorna `null` se questo non rispetta la corretta sintassi.[2]

```
private static string[] ExecuteCommand(string cmd)
{
    string pattern = "*.*";
    string[] args = cmd.Split(' ');

    if (args[0].ToUpper() != "LIST")
        return null;

    if (args.Length > 1)
        pattern = args[1];

    return Directory.GetFiles(folder, pattern)
         .Select(n=>Path.GetFileName(n))        // seleziona il "nome corto"
         .ToArray();
}
```

6.3 Implementazione del client

Il client si connette al server ed entra in un ciclo di richiesta input all'utente. Il ciclo (e il programma) termina quando l'utente inserisce una stringa vuota.

```
static int port = 8080;

static void Main(string[] args)
{
    TcpClient cli = new TcpClient();
    cli.Connect("127.0.0.1", port);
    var ns = cli.GetStream();
    string cmd = "";
    do
    {
        Console.Write(">> ");
        cmd = Console.ReadLine();
        if (cmd != "")
        {
            string[] fileNames = SendCommand(cmd, ns);
```

2 Implementazione discutibile.

```
            foreach (var name in fileNames)
            {
                Console.WriteLine(name);
            }
        }

    }
    while (cmd != "");
    cli.Close();
}
```

Nota bene: prima di terminare, il client chiude la connessione. È fondamentale, altrimenti il server non sarebbe in grado di rilevare che la comunicazione è terminata.

6.3.1 Invio del comando al server

Il metodo `SendCommand()` invia il comando e legge la risposta del server:

```
static string[] SendCommand(string cmd, NetworkStream ns)
{
    List<string> fileNames = new List<string>();
    // NOTA: qui non uso uno StreamWriter ma direttamente il NetworkStream
    byte[] data = Encoding.UTF8.GetBytes(cmd+"\r\n");  // aggiunge fine riga
    ns.Write(data, 0, data.Length);

    StreamReader sr = new StreamReader(ns);
    string line = sr.ReadLine();
    while (line !=null && line != "")
    {
        fileNames.Add(line);
        line = sr.ReadLine();
    }
    return fileNames.ToArray();
}
```

L'invio del comando viene eseguito direttamente su `NetworkStream`; la risposta viene letta mediante uno `StreamReader`.

Nota bene: il ciclo di lettura termina quando sopraggiunge una linea vuota, oppure il server ha chiuso la connessione.

(`SendCommand()` non analizza il contenuto della risposta e dunque tratta il messaggio “ERROR” come se fosse un nome di file; questo rappresenta senz’altro un limite del protocollo.)

Appendice

Esiste una caratteristica delle classi `StreamReader` e `StreamWriter` che può provocare dei bug nell'implementazione delle operazioni di lettura/scrittura su `NetworkStream`: il *buffering*.

Il *buffering* implica la capacità gestire una *cache* dei dati letti o da scrivere su uno *stream*. Lo scopo è quello di aumentare le prestazioni, riducendo il numero di operazioni di lettura e scrittura. Ebbene, in alcuni scenari questa caratteristica è di ostacolo alla corretta implementazione di un protocollo di comunicazione basato su TCP.

6.4 Eliminare il *buffering* nella scrittura.

Considera il seguente codice: implementa un client che invia un messaggio a un server e resta in attesa di una risposta, che visualizza nello schermo (sorvolo sull'implementazione del server):

```
static void Main(string[] args)
{
    TcpClient cli = new TcpClient();
    cli.Connect("127.0.0.1", port);
    var ns = cli.GetStream();
    StreamReader sr = new StreamReader(ns);
    StreamWriter sw = new StreamWriter(ns);

    sw.WriteLine("Hello!");  // "Hello!" viene bufferizzato (non inviato)
                             // il server non riceverà alcun dato!
    string risposta = sr.ReadLine();  // la risposta non arriverà mai!

    Console.WriteLine(risposta);
}
```

L'esecuzione dimostra che il client resta bloccato sulla lettura della risposta, che non arriverà mai. Ma la colpa non è del server; infatti, il messaggio `"hello"` non arriva al server, ma resta memorizzato in un buffer interno dell'oggetto `sw`.

Una corretta implementazione del protocollo client-server richiede che ogni messaggio sia inviato immediatamente; a questo scopo si può impostare la proprietà `AutoFlush` dello `StreamWriter`:

```
static void Main(string[] args)
{
    ...
    StreamWriter sw = new StreamWriter(ns);
    sw.AutoFlush = true;
    ...
}
```

Dopodiché, ogni operazione di scrittura invierà immediatamente i dati sullo *stream*.

6.4.1 Implementazione di un metodo di scrittura

Benché sia facile superare i problemi di bufferizzazione dello `StreamWriter`, può essere comunque utile implementare il proprio metodo di scrittura:

```
public static void WriteLine(this Stream st, string text)
{
    var data = Encoding.UTF8.GetBytes(text);
    st.Write(data, 0, data.Length);
}
```

`WriteLine()` è implementato come metodo di estensione della classe `Stream` (notare `this` prima di `Stream`). Il metodo codifica i caratteri in UTF8; ovviamente, se la comunicazione dovesse avvenire mediante un diversa tipo di codifica, dovrebbe essere modificato.

Un simile metodo evita di dover creare uno `StreamWriter` e può essere usato direttamente sul `NetworkStream`:

```
static void Main(string[] args)
{
    TcpClient cli = new TcpClient();
    cli.Connect("127.0.0.1", port);
    var ns = cli.GetStream();
    ...
    ns.WriteLine("Hello!");  // Usa l'extension method
    ...
}
```

6.5 Eliminare il *buffering* nella lettura

In questo caso la questione è diversa e potenzialmente più complicata. Negli scenari come quello visto in precedenza, la funzionalità di *buffering* dello `StreamReader` non provoca alcun problema. Il problema sorge quando, per motivi di implementazione, è necessario che due o più `StreamReader` debbano leggere dallo stesso *stream*. In questo caso il *buffering* altera la sequenza di lettura dei messaggi.

Ad esempio, supponiamo che il client invii al server due messaggi (che chiamerò *head* e *body*) separati da *newline*. Per esigenze che qui non ci interessano, i messaggi sono letti da due metodi distinti, ognuno dei quali utilizzerà un proprio `StreamReader`.

```
void ReadHeadMessage(NetworkStream ns)
{
    StreamReader sr = new StreamReader(ns);
    string msg = sr.ReadLine();
    //... elabora intestazione del contenuto inviato dal client
}
```

```
void ReadBodyMessage(NetworkStream ns)
{
    StreamReader sr = new StreamReader(ns);
    string msg = sr.ReadLine();
    //... elabora corpo del contenuto inviato dal client
}
```

```
void ReadFromClient(TcpClient client)
{
    var ns = client.GetStream();
    ReadHeadMessage(ns);  // ReadLine() legge sia "testa" che "corpo"!
    ReadBodyMessage(ns);  // resta in attesa di un "corpo" che non arriverà mai!
    ...
}
```

Ebbene, questa implementazione non funziona. Il primo `ReadLine()` ad essere eseguito restituisce sì il messaggio *head*, ma contestualmente legge dallo *stream* tutti i dati disponibili (fino a un massimo stabilito dalla dimensione del buffer interno), e dunque anche il messaggio *body*, che dovrebbe essere letto dal secondo metodo.

Diversamente dallo `StreamWriter`, lo `StreamReader` non fornisce la possibilità di disabilitare la gestione del *buffering*; dunque: in scenari simili, non si deve usare la classe `StreamReader`.

In generale, è opportuno non usarla mai e realizzarsi da sé un metodo di lettura dei dati.

6.5.1 Implementazione di un metodo di lettura dei dati

Segue un metodo che legge una riga di testo da uno *stream:*

```
public static string ReadLine(this Stream st)
{
    List<byte> lineBuffer = new List<byte>();
    int b = -1;
    while (true)
    {
        b = st.ReadByte();
        if (b == 10 || b < 0 || b == 26) break;
        if (b != 13) lineBuffer.Add((byte)b);
    }
    if (b == -1 && lineBuffer.Count == 0)
        return null;
    return Encoding.UTF8.GetString(lineBuffer.ToArray());
}
```

Ecco come usare il metodo nell'esempio precedente:

```
void ReadHeadMessage(NetworkStream ns)
{
    string msg = ns.ReadLine();
    //... elabora intestazione del contenuto inviato dal client
}
```

```
void ReadBodyMessage(NetworkStream ns)
{
    string msg = ns.ReadLine();
    //... elabora corpo del contenuto inviato dal client
}
```

Non solo ora la comunicazione avviene correttamente, ma il codice è anche più semplice, poiché non è più necessario utilizzare gli `StreamReader` per leggere i dati.