UDP Client

Comunicazione unicast e multicast

Corso Informatica classe 4ª

Ambiente: .NET 2.0+

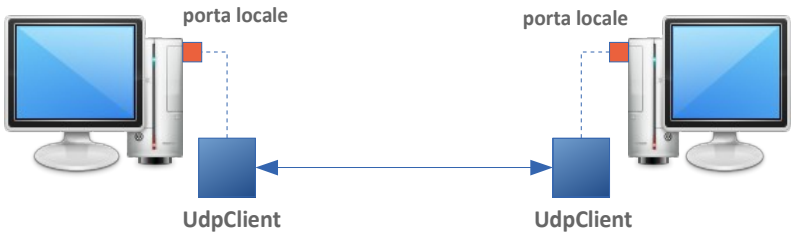
Anno 2017/2018

Indice generale

1 Comunicazione via UDP

Il protocollo UDP implementa un tipo di comunicazione "non affidabile"; il protocollo non garantisce che i pacchetti (***datagrammi***) arrivino a destinazione e/o vengano ricevuti nello stesso ordine con il quale sono stati inviati. Non esiste il concetto di "connessione", né di flusso dei dati (*stream*). La comunicazione avviene mediante singoli datagrammi, i quali sono memorizzati in vettori di byte.

L'invio e la recezione di datagrammi sono realizzati mediante un oggetto di tipo `UdpClient`. Questo è in grado di ricevere datagrammi "mettendosi in ascolto" su una determinata porta (***porta locale***). Lo stesso oggetto può inviare datagrammi a un host remoto specificando il suo *endpoint* (indirizzo/nome host remoto + porta remota).



1.1 Comunicazione "client-server"

Data la natura del protocollo, non siamo di fronte al classico concetto di "client-server", poiché un oggetto `UdpClient` può fungere sia da server che da client. Vale la regola: un oggetto che funziona da server (attende richieste e fornisce risposte), deve connettersi a una porta locale ben definita e dunque conosciuta dai client che vogliono comunicare con esso. Un oggetto che funziona da client può connettersi a una porta locale qualsiasi, con il vincolo che la stessa porta non può essere utilizzata da più oggetti contemporaneamente.

1.2 Comunicazione "multicast"

Il protocollo prevede due forme di comunicazione: ***unicast*** e ***multicast***. Nella prima vi sono soltanto due soggetti coinvolti; nella seconda, i datagrammi possono essere spediti da un mittente e ricevuti contemporaneamente da più riceventi.

2 Introduzione a UdpClient

2.1 Crezione di un oggetto UdpClient

Perché un oggetto `UdpClient` possa inviare e/o ricevere messaggi è necessario che si colleghi a una porta locale. Se non ne viene fornita una, l'oggetto viene connesso a una porta locale scelta a caso dal SO e tenta di comunicare utilizzando la scheda di rete predefinita.

2.1.1 Specificare la porta locale

È possibile specificare la porta locale direttamente nel costruttore:

```
var cli = new UdpClient(10000); // -> porta locale: 10000
```

Nel caso in cui l'host abbia più schede di rete, l'oggetto comunicherà attraverso la scheda di rete predefinita.

> **Host con più schede di rete**
>
> Se un host ha più schede di rete e si desidera ricevere i datagrammi da una qualsiasi di esse è necessario utilizzare un approccio diverso, specificando l'*end point* locale (indirizzo+porta) al quale collegare l'oggetto:
>
> ```
> var epLocal = new IPEndPoint(IPAddress.Any, 10000);
> var cli = new UdpClient(epLocal); -> qualunque scheda, porta: 10000
> ```
>
> `IPAddress.Any` sta per "qualsiasi indirizzo" e fa sì che l'oggetto possa ricevere i datagrammi attraverso una scheda di rete qualsiasi del computer locale.

2.2 Invio di datagrammi

Un datagramma è rappresentato da un vettore di byte e può essere spedito mediante il metodo `Send()`, il quale richiede l'*endpoint* del destinatario (*end point* remoto). Il seguente codice invia la stringa `"Hello!"` al computer di indirizzo `192.168.50.1`, porta `10000`:

```
var epRemoto = new IPEndPoint(IPAddress.Parse("192.168.50.1"), 10000);

byte[] data = Encoding.UTF8.GetBytes("Hello!");
cli.Send(data, data.Length, epRemoto);
```

Nota bene: la stringa deve essere convertita in un vettore di byte; a questo scopo viene usato il metodo `GetBytes()` e la codifica UTF8.

Alternativamente, è possibile specificare l'*endpoint* remoto indicando il nome dell'host e la porta:

```
byte[] data = Encoding.UTF8.GetBytes("Hello!");
cli.Send(data, data.Length, "WKS-2D2-01", 10000);
```

Oppure il suo indirizzo e la porta:

```
cli.Send(data, data.Length, "192.168.50.1", 10000);
```

2.3 Ricezione di datagrammi

Il metodo `Receive()` si mette in attesa di un datagramma e lo restituisce in un vettore di byte. Il metodo blocca l'esecuzione fino a quando non riceve il datagramma. (Oppure fino a quando non scade il timeout di lettura, se questo è stato impostato.) Il metodo richiede come argomento un oggetto `IEndPoint`, al quale assegna l'*endpoint* del mittente.

```
IPEndPoint ipe = null;       // è necessario inizializzarlo (qualunque valore va bene)
byte[] data = cli.Receive(ref ipe);
```

Nota bene: anche se non si intende utilizzare l'*endpoint*, è necessario passarlo al metodo.

2.4 Ricezione in un thread separato

L'esecuzione di `Receive()` sospende l'esecuzione del thread fino all'arrivo di un datagramma; ciò è un problema se l'oggetto `UpdClient` funziona da server e dunque si suppone che debba restare in attesa dei messaggi inviati dai client. In questo caso è necessario eseguire `Receive()` in un thread secondario, in modo da non bloccare il thread principale.

Il codice che segue implementa un server che produce un "eco" dei datagrammi ricevuti.

```
static UdpClient server = new UdpClient(10000);  // ascolta sulla porta 10000

static void Main(string[] args)
{
    Task.Run(() => ServerEcho());            // esegue in un thread secondario
   ...
    Console.ReadKey();
}

private void ServerEcho()
{
    while (true)
    {
        IPEndPoint ipe = null;
        byte[] data = server.Receive(ref ipe);  // ipe contiene l'endpoint del mittente
        server.Send(data, data.Length, ipe);    // rispedisce il pacchetto al mittente.
    }
}
```

Nota bene: il metodo non termina mai; resta costantemente in attesa di datagrammi in arrivo, che rispedisce al mittente. L'*end point* del mittente è memorizzato nella variabile `ipe`, valorizzata dal metodo `Receive()`.

Quello mostrato è un esempio di comunicazione *unicast*. Di seguito mostro come inviare e ricevere datagrammi attraverso un indirizzo *multicast*.

3 Comunicazione *multicast*

Il protocollo IP prevede un range di indirizzi dedicato alla comunicazione *multicast*, nella quale un datagramma può essere ricevuto contemporaneamente da più processi. In questo range esiste un sotto insieme di indirizzi locali (visibili soltanto all'interno della rete locale):

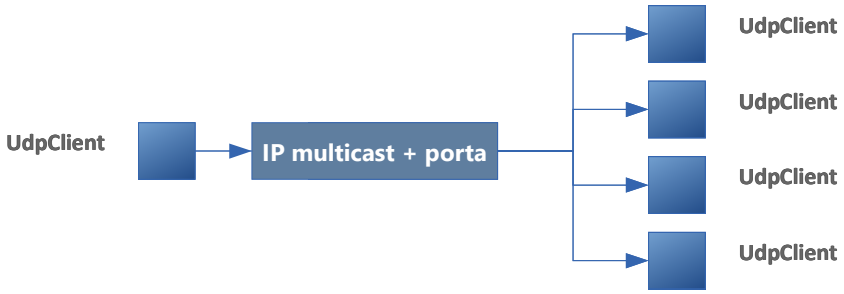
224.0.0.0 – 224.0.0.255

Esiste inoltre un sotto insieme di indirizzi globali (pubblici):

224.0.1.0 – 238.255.255.255

3.1 Registrazione su un indirizzo *multicast*

Nella comunicazione *multicast* i datagrammi non vengono inviati a un host ma a un indirizzo *multicast*: tutti processi "registrati" su quell'indirizzo riceveranno i datagrammi.



La registrazione avviene mediante il metodo `JoinMulticastGroup()`.

Nel codice seguente, un oggetto si registra su un indirizzo *multicast* locale e quindi resta in attesa di datagrammi, che si limita a visualizzare.

```
static UdpClient receiver = new UdpClient(10000);  // porta ben definita

static void Main(string[] args)
{
    var ipMulticast = IPAddress.Parse("224.0.0.1");

    receiver.JoinMulticastGroup(ipMulticast);

    Task.Run(() => ReceiverLoop());
    ...
    Console.ReadKey();
}

private void ReceiverLoop()
{
    while (true)
    {
        IPEndPoint ipe = null;
        byte[] data = receiver.Receive(ref ipe);
        string msg = Encoding.UTF8.GetString(data);
        Console.WriteLine(msg);
    }
}
```

Nota bene: la registrazione (metodo `JoinMulticasGroup()`) all'indirizzo *multicast* non è sufficiente; è necessario che l'oggetto `UpdClient` sia collegato a una porta ben definita, che sarà la stessa utilizzata dagli altri processi per l'invio dei datagrammi.

3.2 Invio di datagrammi a un indirizzo *multicast*

L'invio di datagrammi funziona con lo stesso meccanismo visto finora. Il mittente deve semplicemente specificare l'*end point* remoto, che in questo caso è rappresentato da "indirizzo multicast + porta".

3.3 Invio e recezione di datagrammi in *multicast*

Se l'applicazione deve sia inviare che ricevere datagrammi, è necessario utilizzare due oggetti `UdpClient` distinti, uno per l'invio, l'altro per la ricezione. I due oggetti devono essere collegati a porte locali distinte, e soltanto quello usato per la ricezione dovrà registrarsi all'indirizzo *multicast* mediante il metodo `JoinMulticasGroup()`.

4 Scenari applicativi

Segue la presentazione di tre semplici scenari che vedono l'uso di oggetti `UdpClient`.

4.1 Trasmissione di dati

In questo scenario l'applicazione deve soltanto inviare dati, senza alcuna necessità di leggerli:

1. erogazione di streaming video: l'applicazione invia i pacchetti video a un indirizzo multicast.
2. sincronizzazione ora: un servizio invia un pacchetto contenente l'ora esatta.
3. trasferimento file: un'applicazione invia il contenuto di un file a un'altra[1].

4.2 Ricezione dati

L'applicazione deve soltanto ricevere dati. È sufficiente un solo oggetto `UdpClient`:

1. fruizione di streaming video: l'applicazione si registra per uno streaming video e riceve i pacchetti che compongono il flusso video.
2. sincronizzazione ora: un client si registra su un indirizzo *multicast* che fornisce l'ora esatta e la usa per impostare l'ora del computer.
3. trasferimento file: un'applicazione riceve il contenuto di un file e lo salva su disco.

4.3 Trasmissione/ricezione dati

L'applicazione trasmette e riceve dati. Di norma sono necessari almeno due oggetti `UdpClient`:

1. chat: ricezione e invio dei messaggi di chat.
2. trasferimento affidabile di file: mittente e ricevente devono implementare un protocollo che implichi la ritrasmissione di eventuali pacchetti persi. Il mittente deve ricevere conferme dal ricevente.
3. sincronizzazione ora a richiesta: un client "pinga" un servizio di sincronizzazione e riceve come risposta l'ora esatta.

In linea generale (a parte l'ultimo esempio), è importante che i due oggetti lavorino in thread separati.

1 È soltanto uno scenario di esempio; nella realtà il protocollo TCP si presenterebbe meglio ad un simile compito.