Relazione Progetto
Programmazione di Reti

Federico Campanozzi

May 23, 2021

Contents

1 Descrizione Generale **2**

2 Sviluppo **3**
2.1 Classi 3
2.2 Multithreading 8

3 Risorse esterne **9**

4 Test **9**

1 Descrizione Generale

L'applicazione, sviluppata in python, deve simulare un ambiate IOT dove **n** dispositivi inviano, a loro discrezione, dei dati in un formato standard ad una interfaccia di rete tramite il protocollo UDP. Quest'ultima poi deve re-indirizzare i dati ricevuti ad un server remoto per essere memorizzati e poi eventualmente processati.
Ecco uno schema sommario del funzionamento dell'applicazione:

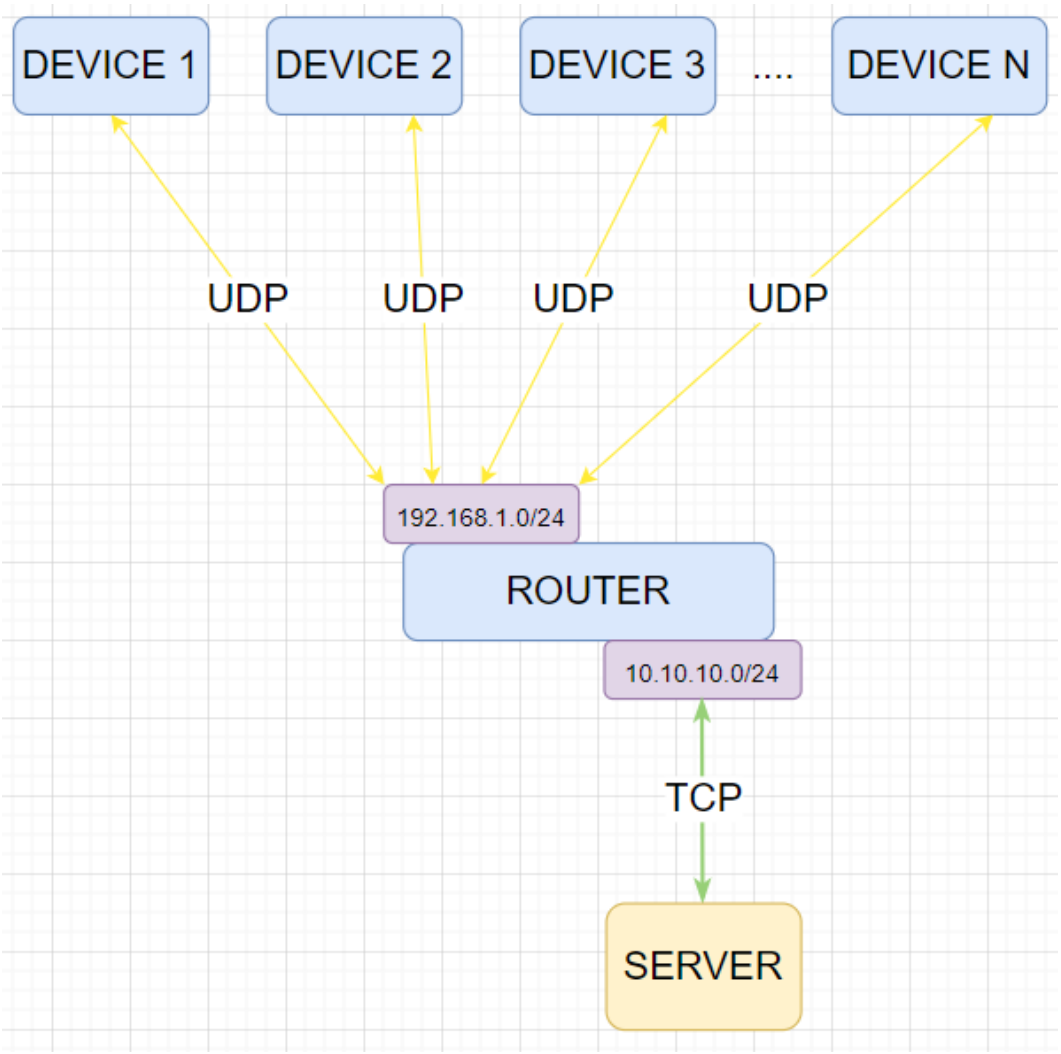

Figure 1: Infrastruttura di rete generale

2 Sviluppo

2.1 Classi

In questa sezione trattero' di come ho implementato le singole classi e cerchero' di riassumere le loro caratteristiche principali.
Partiamo dalla suddivisione dei moduli. Allora abbiamo:

- **core**

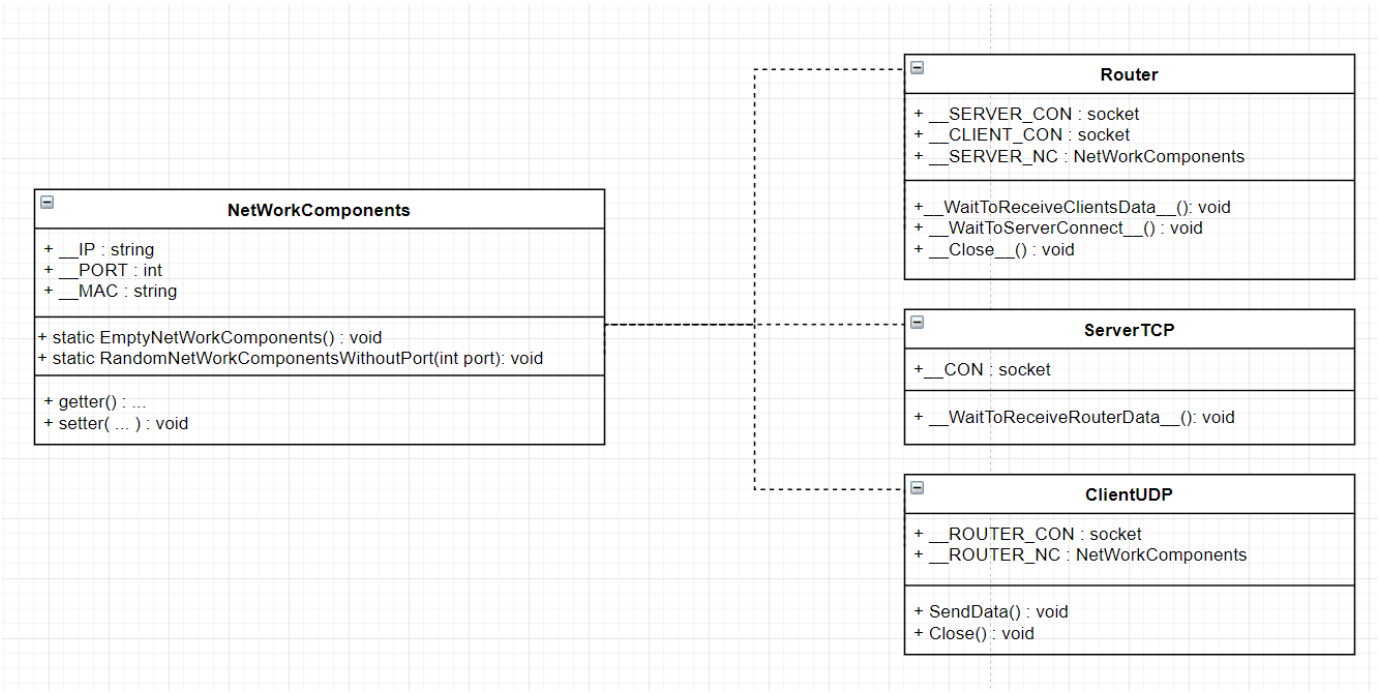

Figure 2: Core module

In questo modulo sono implementate le classi principali al funzionamento di una qualsisi applicazione di rete. Ho quindi creato una classe *NetWorkComponents* che racchiude il concetto di componente di rete.
Come si puo' notare dalla fig. 2 le classi *Router*, *ServerTCP* e *ClientUDP* estendono il concetto di *NetWorkComponents*.
Nello specifico abbiamo:

1. NetWorkComponents
   Come gia' detto in questa classe e' racchiuso il concetto di componente di rete. Quindi sono memorizzate i valori di IP, MAC e Porta, caratteriste fodamentali di qualsiasi connessione TCP e UDP.
   I metodi statici *NetWorkComponents::EmptyNetWorkComponents* e

NetWorkComponents::RandomNetWorkComponentsWithoutPort seguono il **simple factory template** e forniscono rispettivamente un oggetto vuoto e un oggetto con ip e mac generati casualmente mentre la porta viene passata come argomento al metodo.

2. Router Questa classe rappresenta un router, nel mio progetto il router deve poter comunicare sia con i client che con il server in due interfaccie differenti quindi ho creato due connessione rappresentate rispettivamente dagli oggetti __SERVER_CON e __CLIENT_CON.
   L'oggetto __SERVER_NC mi serve solo per avere sempre disponibili le informazioni del server e riempire l'oggetto della classe **Package** quando ricevo i dati da un client e devo re-indirizzarlo verso il server.
3. ClientUDP Questa classe rappresenta un qualsiasi host (pc, cellurare, tablet ecc...) in grado di connettersi ad una rete e inviare dati.
   In questa simulazione il client invia i dati, generati nel metodo *ClientUDP::SendData()*, al router grazie all'oggetto socket __ROUTER_CON.
   Anche in questo caso, come nel *Router*, la variabile __ROUTER_NC mi serve per riempire l'oggetto della classe **Package** prima di inviare il messaggio.
   Il metodo *ClientUDP::Close()* serve per chiudera la connesione e liberare la porta. Anche se UDP e' un protocollo non orietato alla connessione (*connnectionless*) non e' richiesta la chiusura ma e' buona norma specificarlo.
4. ServerTCP Il server, una volta istanziato, si collega al router e resta in attesa di ricevere i dati dal *Router*

- **test**

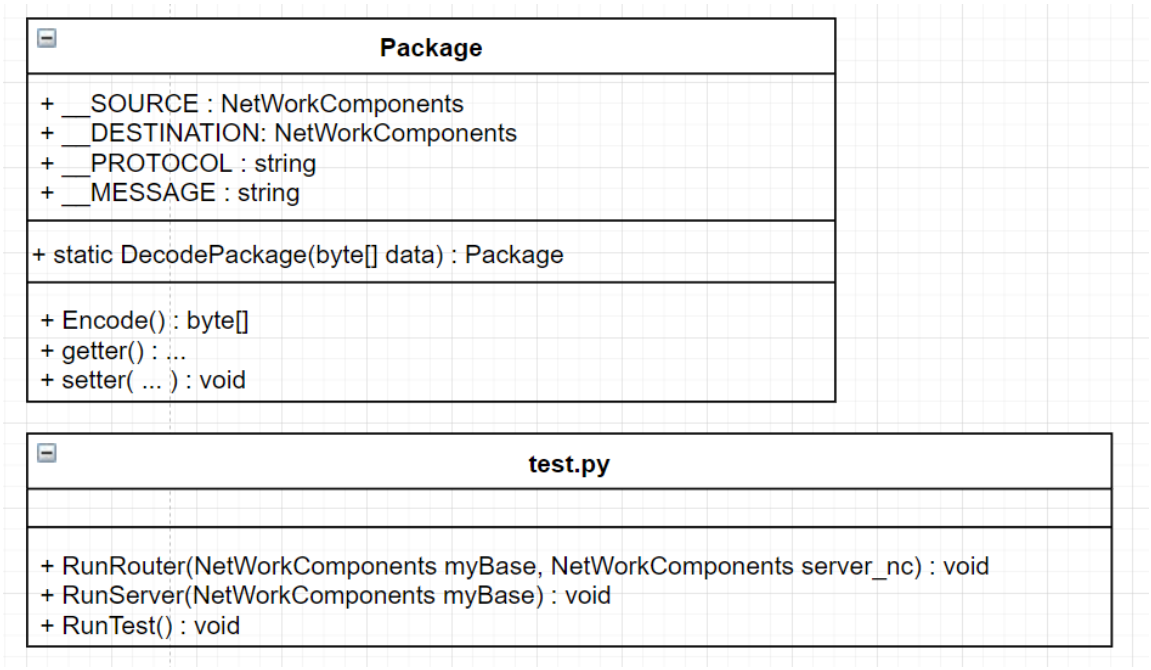

Figure 3: Test Module

In questo modulo sono contenute quelle classi che servono alla messa a punto del programma. Nello spefico abbiamo:

1. Package Questa classe rappresenta il formato "standard" (all'interno dell'applicazione) che mi sono inventato per passare i dati in maniera intelligente ed efficace e per sfruttare al massimo il concetto di riusabilita' del codice.
   Il metodo *Package::Enconde()* traduce le informazione da testo, quindi string o int, in byte per essere poi pronte ad essere instradate all'interno del canale di comunicazione, ad esemppio doppino,cavo coassiale, fibra ottina, segnale radioe ecc... .
   Il metodo *Package::DecodePackage()* fa l'operazione inversa di decofica. Quindi prende i dati che gli arrivano come array di byte e li converte nel formato corrispondente. Anche qui ho struttato una **simple factory**.
2. test.py
   Questa script si occupa della gestione della simulazione, cioe' gestisce tutte le componenti, quindi server, router e molteplici client. Inoltre si occupa delle fasi di instaurazione della connessione, dell'invio dei dati e della chiusaura delle connessione; nonche' del multi-trheaduing.

Questo aspetto sara' approfondito nel prossimo paragrafo.
Ecco un piccolo schema riassuntivo dello script:

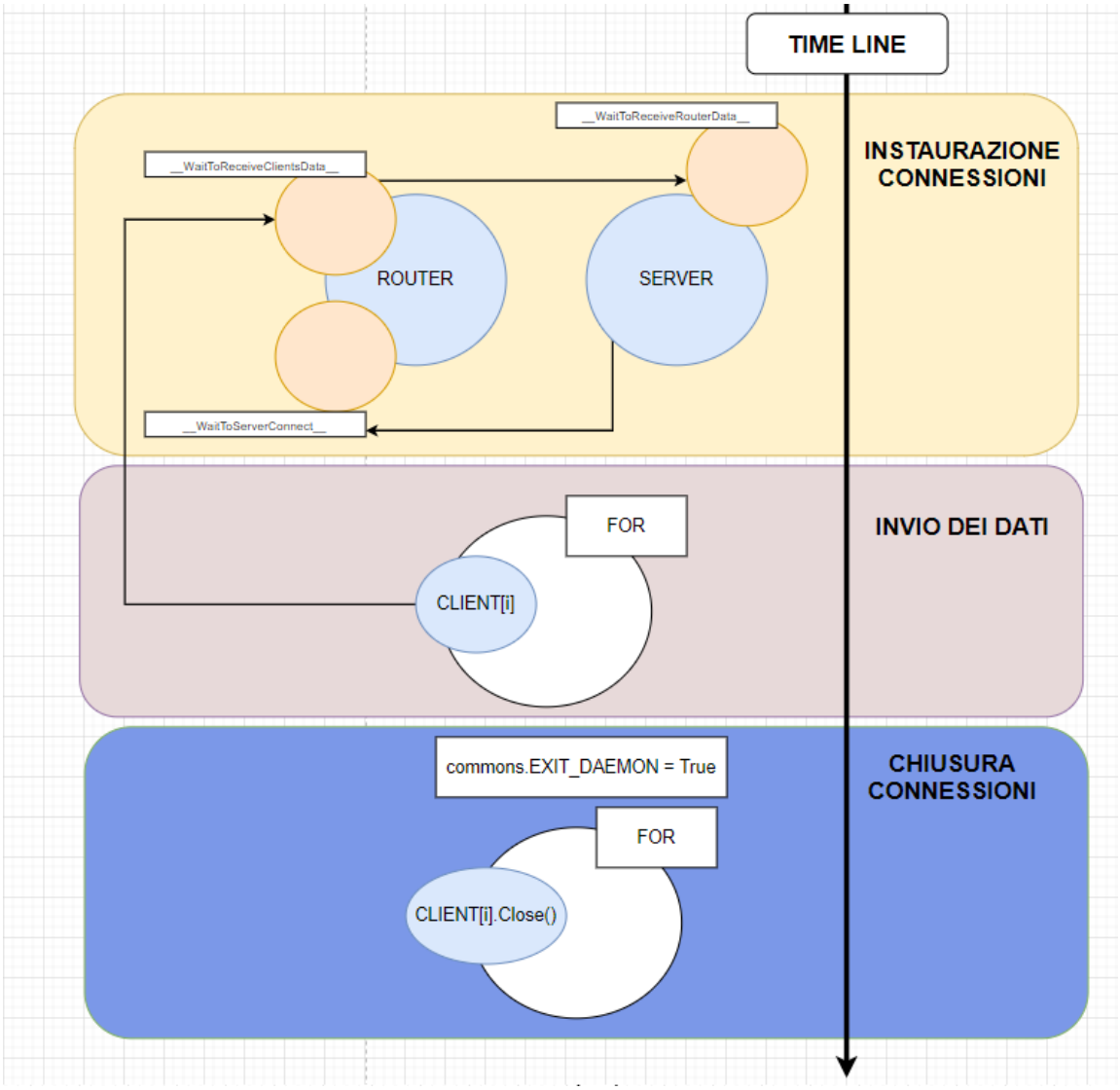

Figure 4: Schema degli eventi dello script test.py

- **utilities**

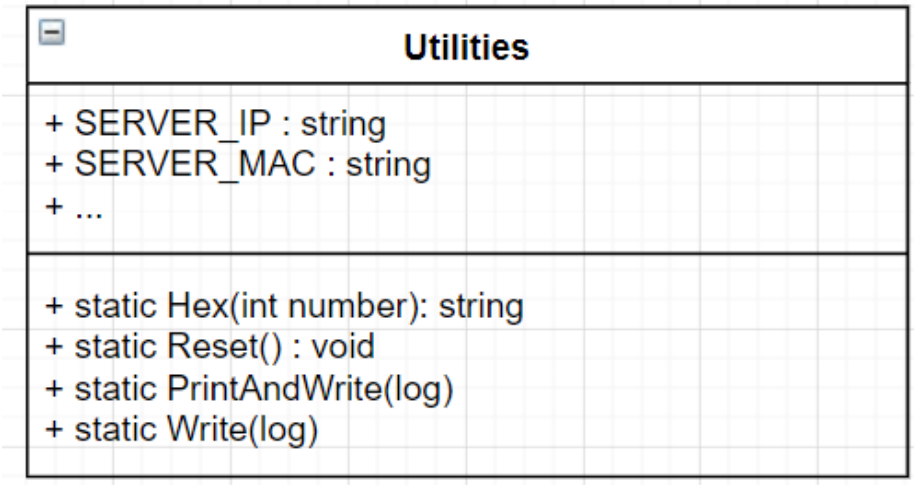

Figure 5: Utilities class

Come si puo' notare dalla figura (fig. 5) questa classe non fanno altro che fornire metodi e costanti utili e comode a tutte le altri classi dell'applicazione. Infatti, sia metodi che variabili, sono dichiarate come **static** e **public** proprio per essere utilizzate con estrema facilita'.

2.2 Multithreading

E' stato necessario sviluppare l'applicazione con piu' thread per gestire al meglio la sincronizzazione.
Nell'immagine seguente e' mostrato il flusso completo dei thread:

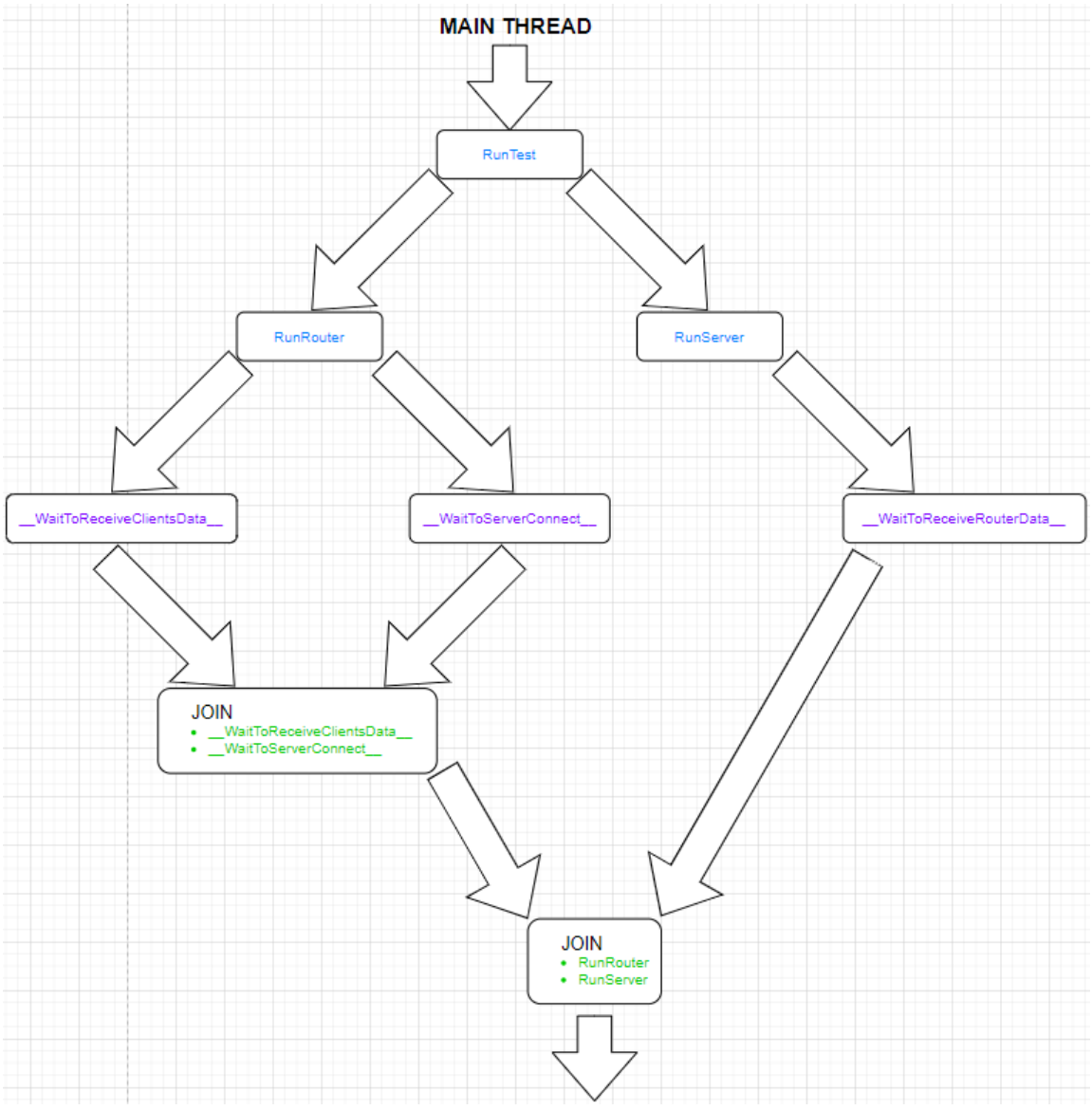

Figure 6: Schema concettuale dei thread

Nello script test.py abbiamo il metodo *RunTest*. Quest'ultimo si occupa

della gestione, su due thread separati, del metodo *RunRouter*, che istanzia un oggetto della classe *Router*, e il del metodo *RunServer*, che istanzia un oggetto della classe *Server*.
Il Router a sua volta gestiste altri due thread uno che aspetta la connessione da parte del server e uno che aspetta l'invio dei dati da parte dei client sull'interfaccia 192.168.1.1/24. Essi si ricongiungono alla fine del metodo __init__().
Anche il Server necessita di un secondo thread il quale rimane in attesa di una ricezione dati proveniente dal router sull'interfaccia 10.10.10.1/24.

3 Risorse esterne

Ho utilizzato le seguenti librerie native in *Python3*.

- socket
- threading
- os
- random
- time
- datetime

Per la gestione del versioning del progetto ho utilizzato il DVCS GitHub. E' possibile visionare il sorgente cliccando **QUI**.

4 Test

Per testare il programma basta aver installato *Python3* ed eseguire lo script *test.py*.
Se tutto va a buon fine, ovvero se le porte sono disponibile e l'ambiente di sviluppo e' correttamete settato, si vedranno delle stampe sulla console. Per un maggior dettaglio si puo' vedere il file `Log.txt` sotto la cartella `Log` situata all'interno della cartella `src`.