A.A. 2020/2021
Relazione di progetto
Programmazione di reti

Leon Baiocchi 0000938801, Federico Brunelli 0000934810

24 giugno 2021

Sommario

Questo documento è una relazione del progetto d'esame di Programmazione di Reti, in particolare della **TRACCIA 1 - Progetto IoT**. Lo scopo di questo documento è quello di permettere la comprensione dei dettagli di progettazione e del funzionamento del seguente progetto. Inoltre, dovrebbe permettere un'ampia visione del lavoro svolto dal team, che ricordiamo essere composto da Leon Baiocchi e Federico Brunelli, in termini di organizzazione, oltre che di progettazione. Per ciascuna delle sezioni del documento sarà fornita una descrizione di ciò che riguarda ognuna di esse, a partire da quella inerente alla fase di *Analisi* fino ai *Commenti finali* per poi arrivare ad una piccola appendice munita di *Guida utente*, utile se si volesse mettere in azione l'applicativo.

Riferirsi all'*Indice* presente nella pagina sottostante per maggiori informazioni. Buona lettura!

Indice

Capitolo 1

Analisi

Il Fondo Ambiente Italiano ci ha commissionato un progetto per la salvaguardia dell'ambiente. Lo scopo di questo progetto consiste nella raccolta di misurazioni relative a temperatura e umidità del terreno in una zona a rischio. La fondazione ha messo a disposizione quattro dispositivi per il rilevamento dati, denominati Smart Meter IoT, e un Gateway per instradare le misurazioni raccolte dai vari device verso un server cloud centrale di proprietà del FAI. La rete che permette ai device di scambiare dati con il gateway ha un'indirizzamento di *Classe C* del tipo **192.168.1.0/24**.

Essa non è l'unica in quanto vi è un'altra rete, che permette al gateway di inoltrare messaggi al *Cloud Server* del FAI tramite un'indirizzamento di classe **10.10.10.0/24**.

I quattro dispositivi si occupano di inviare, una volta al giorno, i dati raccolti attraverso una connessione UDP verso il gateway, il quale, dopo aver ricevuto tutti i pacchetti, inoltrerà tali dati al cloud tramite una connessione TCP.

Essendo questa una relazione di progetto del corso di Programmazione di reti, ci concentreremo solo sugli aspetti di rete, escludendo quelli di IoT.

1.1 Requisiti

Il software, commissionato dal FAI, consiste nel realizzare un *network* che permetta ai quattro **dispositivi** di interfacciarsi con il **gateway**. Quest'ultimo possiede un'ulteriore interfaccia di rete, quindi il cloud è in un'altro *network*, che permette la comunicazione con il **server cloud**.

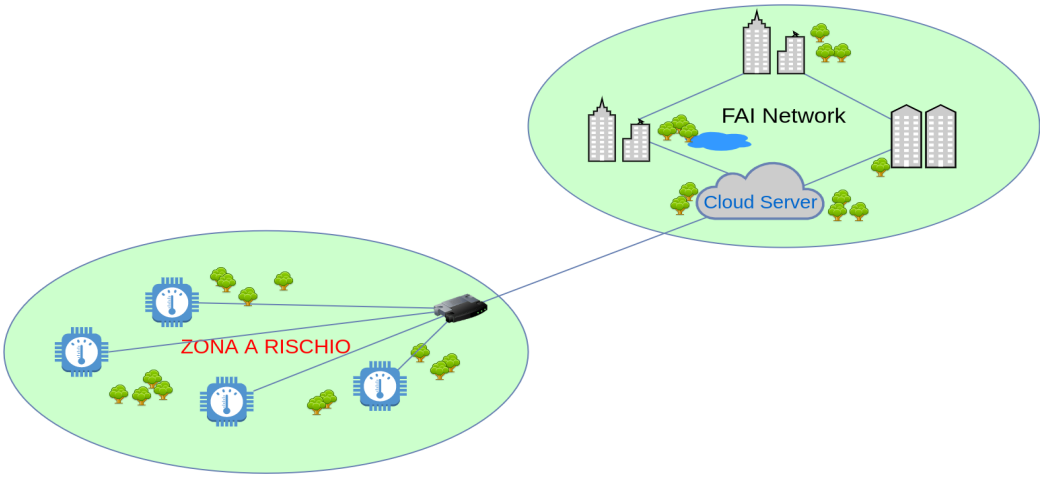

Figura 1.1: Scenario

Requisiti funzionali

- **Addressing**: i *dispositivi* situati nella zona a rischio e l'*interfaccia di rete del gateway* adibita alla ricezione delle misurazioni devono avere indirizzi IP appartenenti alla classe **192.168.1.0/24** mentre l'interfaccia del gateway che si occupa di instradare le misurazioni dei device verso il *server cloud* e quest'ultimo devono avere indirizzi IP del tipo **10.10.10.0/24**.

- I quattro *dispositivi* devono salvare su un **file** le misurazioni ottenute indicando *ora della misurazione, valore della temperatura* e *valore dell'umidità*. I quattro dispositivi si occupano di inviare, *una volta al giorno*, i dati raccolti attraverso una connessione UDP verso il gateway.

- Il *gateway* deve ricevere i dati delle misurazioni dai *dispositivi* e instradarli verso il *cloud*.

- E' inoltre richiesto che il *server cloud* visualizzi, su una console, le misurazioni ottenute nel seguente formato:
IpAddressDevice1–OraMisura–ValoreTemperatura–ValoreUmidità

Requisiti non funzionali

- E' importante che i *dati* vengano scambiati in maniera efficace e consistente riducendo al minimo il sovraccarico dei vari *canali di trasmissione*.

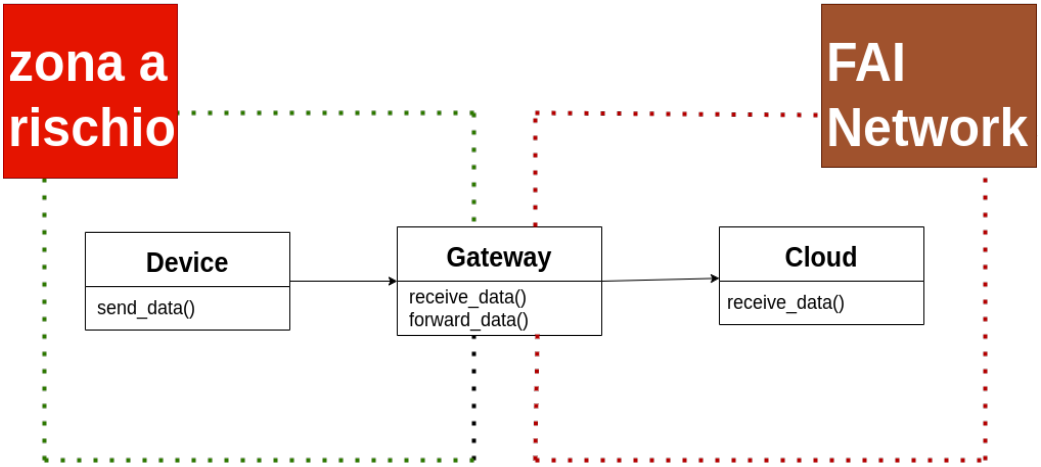

Figura 1.2: Schema UML delle comunicazioni tra gli host

- E' previsto anche che siano prese delle misurazioni riguardanti i *tempi di trasmissione* di ciascun **canale**

1.2 Analisi e modello del dominio

La scenario è il seguente: abbiamo a disposizione **4 dispositivi IoT** in grado di ottenere *dati* dalle misurazioni di temperatura e umidità. Essi si trovano all'interno di una **zona a rischio** e si interfacciano con un **gateway** al quale inviano **periodicamente** i dati raccolti attraverso il protocollo UDP.

Dall'altra parte invece, il **cloud server** centrale situato all'interno della *rete* del FAI aspetta che il gateway si connetta tramite connessione TCP ed inizia a ricevere i dati delle misurazioni, che possono arrivare già formattati correttamente oppure ancora da formattare.

Essenziale il compito del *gateway* che deve ricevere i dati in arrivo dai **dispositivi** per poi riuscire ad instradarli in maniera efficente, eventualmente formattandoli correttamente. La *dimensione dei canali trasmissivi* potrebbe dipendere dal momento in cui si formattano le misurazioni.

Da notare, inoltre, come gli host siano distribuiti su due reti diverse, una situata nella *zona a rischio* mentre l'altra è la *rete del FAI*. Per questo motivo gli host delle due reti hanno **indirizzamento** diverso.

Una visione in UML di come comunicano tra di loro gli host è osservabile nella Figura 1.2.

Capitolo 2

Design

Come già detto nel capitolo precedente, ci occuperemo principalmente degli aspetti di rete, *emulando* le funzionalità dei quattro dispositivi IoT. Inoltre, data la natura didattica del progetto, il sistema deve utilizzare l'interfaccia di loopback per simulare le varie connessioni tra i differenti **hosts** e perciò l'**IP address** e il **MAC address** sono impostati con un valore di *default*.

Per modellare al meglio le varie entità abbiamo preferito sfruttare la *programmazione ad oggetti*. Questo per garantire una maggior leggibilità del codice oltre che una maggiore facilità di manutenzione dello stesso.

2.1 Architettura

Il sistema presenta due **network** differenti, quello della zona a rischio e quello del FAI.

Il committente ha richiesto che le classi di indirizzamento delle due reti siano di tipo C e figurino nel seguente modo: **192.168.1.0/24** e **10.10.10.0/24**. Entrambe hanno quindi stessa *maschera di sottorete*: *255.255.255.0*, che permette di indirizzare 255 possibili host, se, oltre ai già 254 indirizzi disponibili, viene solo considerato l'indirizzo del default gateway e non quello di broadcast.

Perchè lo consideriamo come indirizzo valido?

Perchè avendo un dispositivo che deve avere funzione di *router* come il nostro **Gateway** l'indirizzo del default gateway fa proprio al caso nostro dato che l'host da contattare per mandare messaggi da una rete all'altra è proprio quello. Motivo per il quale questo particolare dispositivo ha due interfaccie di rete.

Gli indirizzi assegnati ai vari hosts delle due reti sono:

- **Zona a rischio**
  - ***Device_1***:
    - IP: 192.168.1.10
    - MAC: 36:DF:28:FC:D1:67
  - ***Device_2***:
    - IP: 192.168.1.15
    - MAC: 04:EA:56:E2:2D:63
  - ***Device_3***:
    - IP: 192.168.1.20
    - MAC: 6A:6C:39:F0:66:7A
  - ***Device_4***:
    - IP: 192.168.1.25
    - MAC: 96:34:75:51:CC:73
  - ***Gateway***:
    - IP: 192.168.1.1
    - MAC: 7A:D8:DD:50:8B:42
- **FAI network**
  - ***Gateway***:
    - IP: 10.10.10.1
    - MAC: 7A:D8:DD:50:8B:42
  - ***Cloud server***:
    - IP: 10.10.10.2
    - MAC: FE:D7:0B:E6:43:C5

Gli **indirizzi MAC** impostati per ogni interfaccia di rete sono tutti *locally administered*(U/L bit = 1) e *unicast*(I/G bit = 0). Per una visione ampia e abbastanza dettagliata dell'architettura di rete e delle funzioni che i vari host svolgono all'interno di essa è necessario prestare attenzione alla Figura 2.1

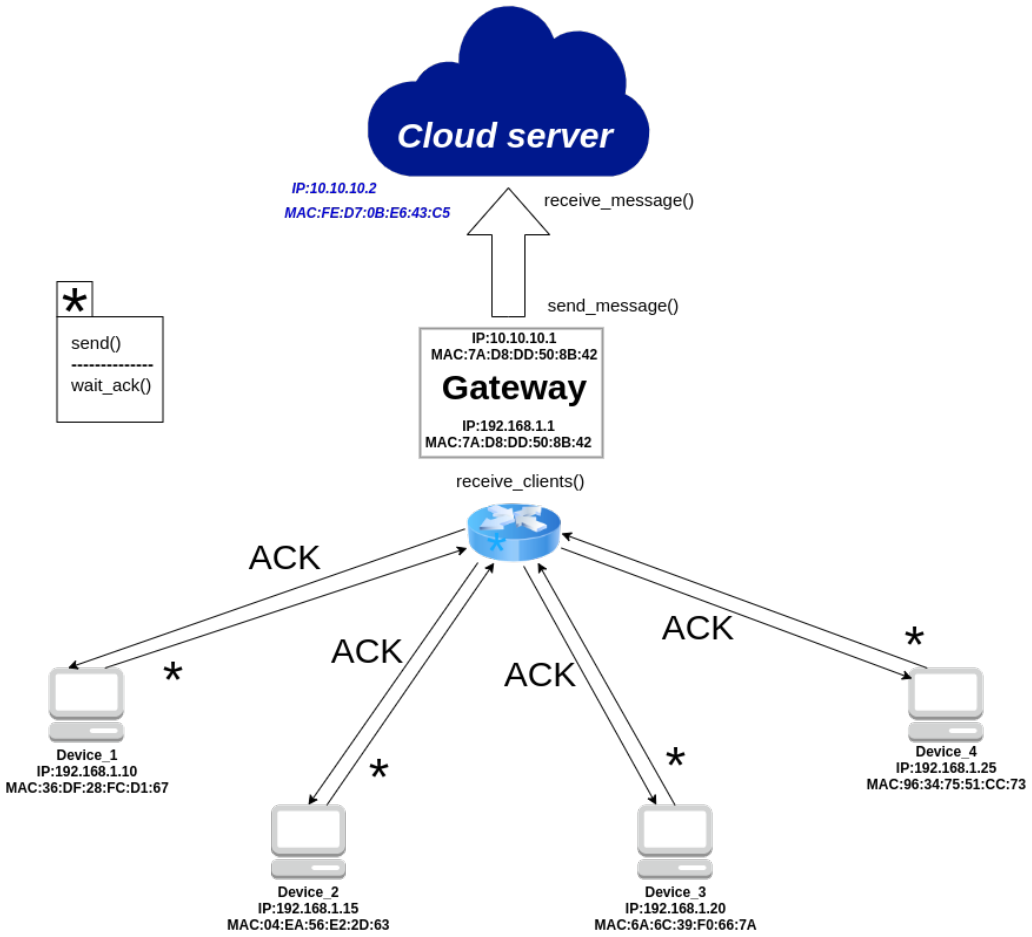

Figura 2.1: Schema rappresentativo dell'architettura di rete e delle operazioni di base svolte dai relativi dispositivi

2.2 Design dettagliato

In questa sezione approfondiremo alcuni elementi del design con maggior dettaglio. Analizziamo quindi la fig. 2.1: partendo dal modo in cui i **device** comunicano con il **gateway**(parte bassa dello schema), durante la fase di progettazione siamo giunti alla conclusione che, essendo UDP un protocollo di trasporto non garante di *data reliability*, avremmo dovuto implementare da soli un meccanismo di avvenuta ricezione del pacchetto.

Questo meccanismo, già noto, sfrutta i messaggi **ACK**nowledged per garantire una corretta fruizione dei *pacchetti di rete* ed assicurarne la ricezione. E' un sistema molto simile, e soprattutto ispirato, al meccanismo utilizza-

to dal protocollo *TCP*. In questo modo i dispositivi possono sapere se il pacchetto inviato è stato consegnato a destinazione.

Number	Time	Source	Destination	Protocol	Lenght	Source-Destination Port
369	115.450801825	127.0.0.1	127.0.0.1	UDP	271	57690 → 10002 Len=229
370	115.451388778	127.0.0.1	127.0.0.1	UDP	45	10002 → 57690 Len=3

Figura 2.2: Cattura Wireshark: comunicazione tra *device* e *gateway*

Una volta che il *gateway* ha inoltrato l'ACK, salva il pacchetto arrivato in una **arp table modificata**, nel senso che oltre a tenere memoria di ogni coppia indirizzo IP, indirizzo MAC; memorizza per ognuna di esse l'ultimo pacchetto ricevuto dal device con IP corrispondente. Ad ogni ricezione viene poi verificato se tutti e 4 i device hanno inviato i dati e se ciò è vero tutti i payload vengono prima formattati e inseriti all'interno di un unico grande pacchetto che successivamente, inseriti gli **headers**, viene inoltrato al *cloud* il quale riceve e stampa a video tutte le misurazioni nel formato richiesto. Proseguiamo ora con una descrizione ancora più nel dettaglio di tutti i vari componenti.

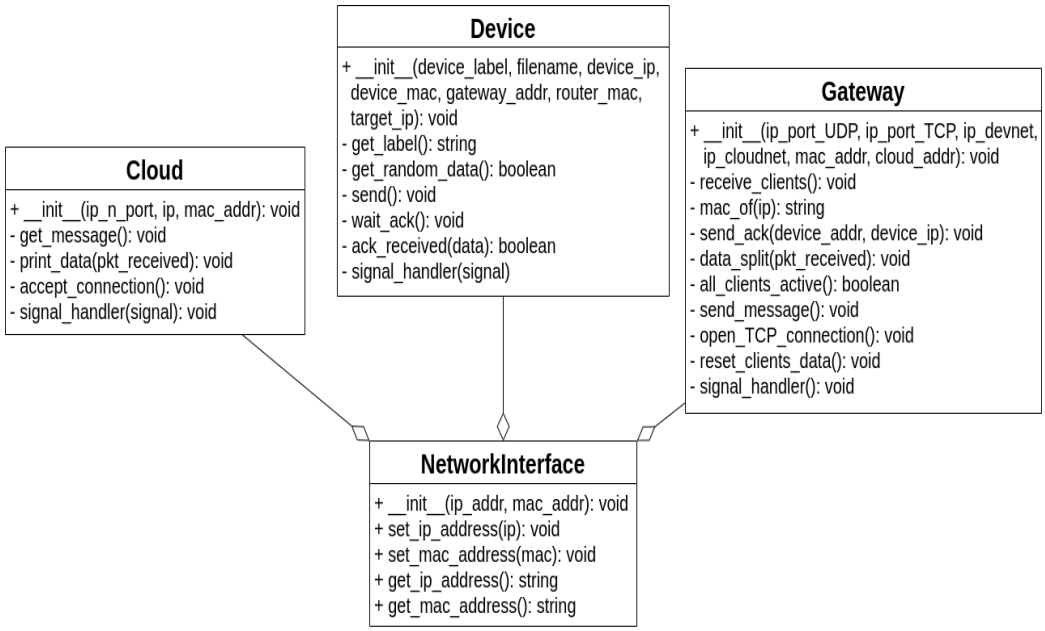

Figura 2.3: UML figurativo delle classi relative agli host

Per rendere più chiaro il network, abbiamo preferito distinguere le interfacce dei vari host, così da permettere una maggiore leggibilità. Va sottolineato il fatto che il **Gateway** possiede due interfacce di rete:

- *device_interface*
- *cloud_interface*

La prima permette la comunicazione con i device, mentre la seconda si interfaccia con il server centrale. Come si può osservare nella Figura 2.3, la classe **NetworkInterface** tiene traccia dell'indirizzo mac dell'host di riferimento e l'indirizzo IP legato alla rete.

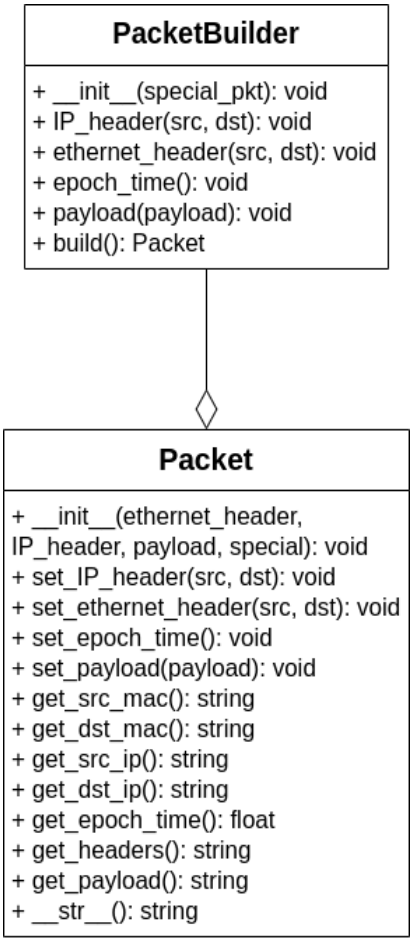

Figura 2.4: UML figurativo del pacchetto trasmesso

Questo schema rappresenta la classe **Packet**: astrazione del pacchetto di rete che viene trasmesso tra i vari host. Esso contiene l'intestazione riguardante gli *indirizzi* **MAC** e **IP** del **mittente** e del **destinatario** e l'**epoch**

time relativo all'inoltro del messaggio, utile se si vuole calcolare per esempio il *tempo impiegato* per la *trasmissione* di tale pacchetto, proprio come nel nostro caso. Inoltre, nel pacchetto vi 'e presente il messaggio vero e proprio(**payload**).

Un oggetto *Packet* è istanziabile attraverso il costruttore che permette anche di specificare se si tratta di un **"pacchetto speciale"** come potrebbe essere nel caso di un *ACK*. Ma non solo, difatti abbiamo pensato che potesse tornare utile la creazione a cascata di un oggetto simile in modo da permettere di creare una molteplicità di pacchetti differenti facilmente. E' per questo che abbiamo sfruttato il pattern **builder** dando vita così alla classe PacketBuilder la quale fornisce metodi a cascata per la creazione degli *headers* ed il *payload*, infine vi è il metodo build() che mette insieme i componenti generando un'oggetto Packet del tipo definito.

Tornando alla classe Packet è da delineare il fatto che i *parametri* del costruttore siano **opzionali**, questo poichè abbiamo voluto dare spazio a varie interpretazioni della classe, difatti con tali argomenti è possibile per esempio creare dapprima un pacchetto vuoto per poi aggiungere "strada facendo" i vari elementi(headers e payload). Sono poi presenti tutti i rispettivi getters di headers, sub-headers(quali sorgente e destinazione di IP header e l'ethernet header) e payload.

Viene fornito anche un metodo per *convertire* il contenuto del pacchetto in stringa.

2.2.1 Classi host: Device

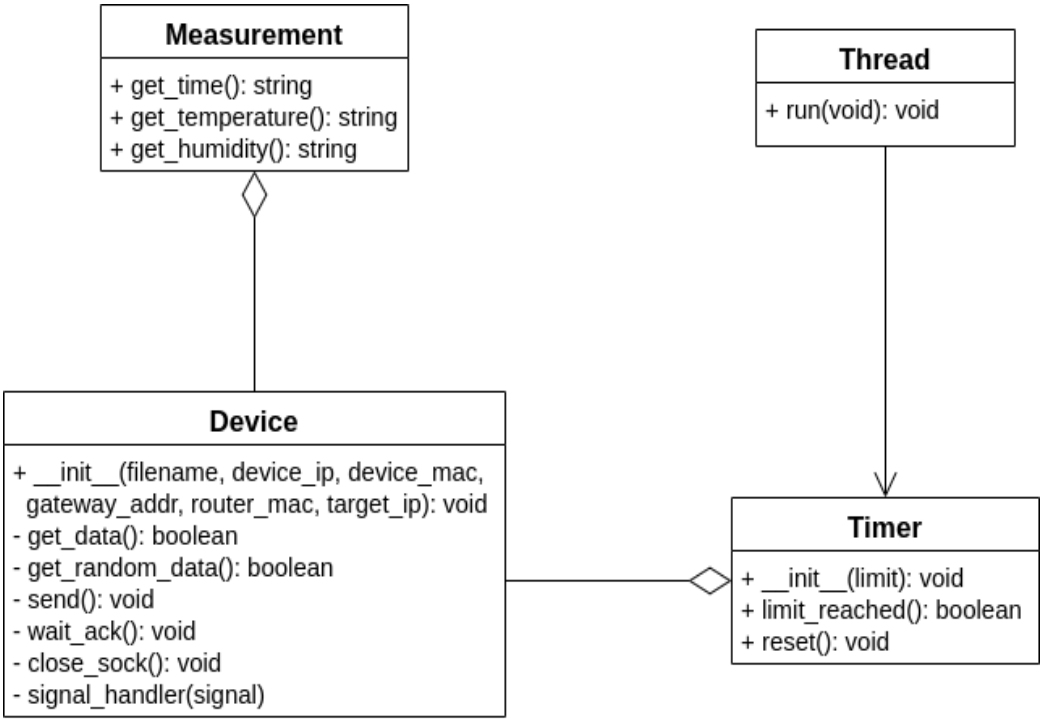

Figura 2.5: UML rappresentativo della classe Device

In figura viene mostrata la struttura della classe **Device** che rappresenta un dispositivo IoT, essa si compone di oggetti di altre classi: **Measurement**, che consente di ottenere misurazioni randomiche; **Timer**, classe figlia di *Thread* che si occupa di scandire il tempo tra l'invio di un pacchetto e un altro.

Device presenta un costruttore, dove avviene tutto, con *parametri obbligatori* ed una serie di metodi tutti privati. Abbiamo fatto questa scelta per tutte le classi che rappresentano dispositivi di rete perchè volevamo rappresentare degli *host specifici* per questo progetto e non universali, per cui ci siamo orientati verso un maggior incapsulamento delle informazioni.

Prima di partire a descrivere il funzionamento della classe *Device* è opportuno introdurre prima le altre due classi:

- *Measurement* è costituita da 3 metodi pubblici che ritornano rispettivamente un'**orario**, un valore di **temperatura** scelto tra un range di valori prestabiliti, un valore indicante l'**umidità** anch'esso scelto allo stesso modo. Il tipo di questi 3 valori di ritorno è sempre lo stesso: una *stringa*.

- *Timer* è una sotto-classe di *Thread* per cui ne eredita le caratteristiche, aggiungendone altre: questa classe difatti è essenziale per inviare *periodicamente* messaggi al *gateway*. Un'oggetto di questo tipo ha infatti un timer interno ed un limite, da impostare via costruttore. Quando il timer giunge al limite il metodo *limit_reached()* ritorna *True*, dopodichè va fatta la *reset()* del timer se si vuole continuare ad utilizzarlo.

Ora possiamo continuare con la classe **Device**: innanzitutto viene istanziato un'oggetto *Timer* e si avvia, successivamente parte un loop che ogni N secondi controlla se il timer è scaduto(limit_reached()=True) e viene generata e salvata su file una misurazione randomica grazie al metodo privato get_random_data() che sfrutta l'utilizzo di *Measurement* per dare origine a dati randomici. Se il timer è scaduto allora viene chiamato il metodo *send()* che si occupa di aggiungere *epoch_time* e *payload*, prelevandolo dal file del dispositivo, da notare che il pacchetto viene inizializzato nel costruttore, prima che parta il timer, con l'*intestazione IP ed ethernet* già costruite; poi il pacchetto viene serializzato, tramite il modulo **pickle**, ed inviato. A questo punto il device si mette ad aspettare l'ACK del getaway prima di fare qualsiasi altra cosa.

Ricevuto l'ACK il dispositivo può passare al reset del file di salvataggio misurazioni e del timer.

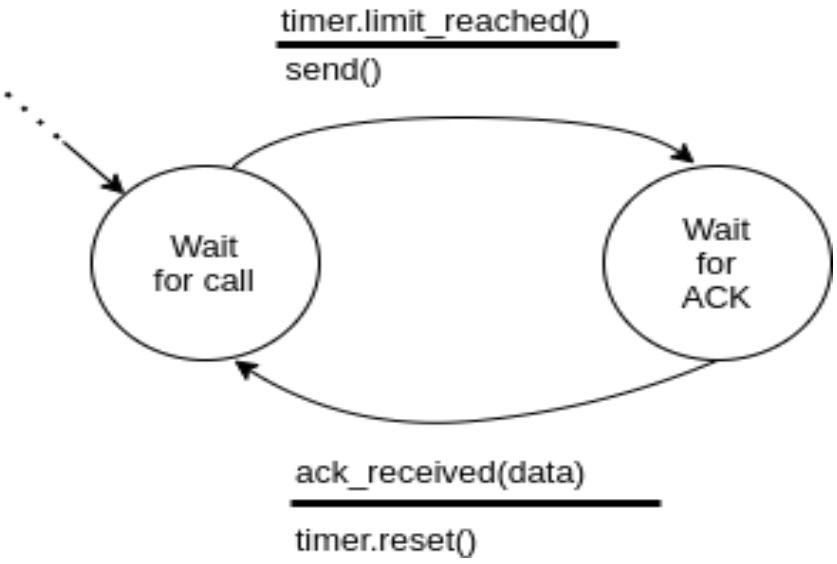

Figura 2.6: Automa a stati finiti che rappresenta stati e comportamenti di Device

2.2.2 Classi host: Gateway

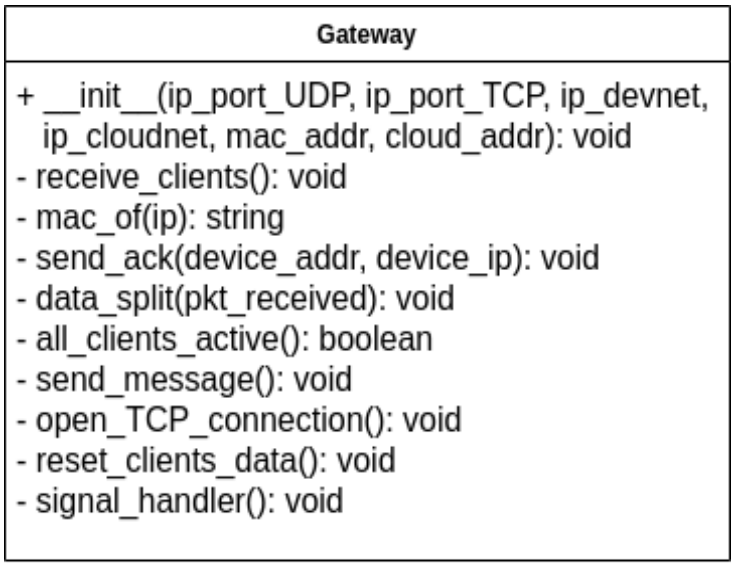

Figura 2.7: Schema UML del Gateway

Il **gateway** viene rappresentato dalla classe Gateway: anch'essa a livello di organizzazione dei metodi ha le stesse caratteristiche della classe **Device**, come appunto detto nell'apposita sotto-sezione. In questa classe il metodo *receive_clients()* svolge il ruolo di *main loop*: all'interno di questo ciclo avviene la ricezione del messaggio inviato da uno dei *device*, il messaggio viene poi de-serializzato tramite modulo **pickle**. Se il messaggio de-serializzato non è un messaggio vuoto *presumiamo* sia quello giusto creando quindi un pacchetto speciale, che è il nostro ACK, tramite *PacketBuilder*. Dopodichè il *pacchetto* viene serializzato ed inviato al *device* da cui il *gateway* ha ricevuto l'ultimo pacchetto.

All'interno della classe **Gateway** c'è una **struttura dati** molto importante, essenziale per instradare i pacchetti dei client nel modo giusto. Si tratta di un *dizionario* dove ogni chiave corrisponde ad una **coppia indirizzo IP, indirizzo MAC** corrispondente ognuna ad un device diverso; ogni valore è un riferimento al pacchetto arrivato, può anche non essere arrivato, in questo caso il valore viene impostato a *None*, valore di default.

Dopo questa breve spiegazione della struttura dati utilizzata per la *gestione dei client* passiamo alla fase successiva. Quando un *pacchetto* passa per la data_split viene verificato se l'indirizzo IP sorgente corrisponde ad uno degli indirizzi IP presenti nella **arp table modificata**, se la verifica da esito negativo viene sollevata un'**eccezione**, le socket vengono chiuse e il

programma termina perchè vuol dire che ha provato a connettersi un *device non identificato*. Contemporaneamente viene fatto un'altro controllo che analizza il valore di ogni chiave dell'*arp table* e verifica se è nullo. Se queste due condizioni danno esito *positivo*, il *pacchetto* viene inserito nell'*arp table* ed il **contatore** di clients che hanno inviato il proprio messaggio viene *incrementato*. Nel caso in cui il **contatore** raggiunge il numero totale di *device* del sistema, quindi 4: viene chiamato il metodo *send_message()*.

Esso verifica se è già stata aperta una connessione TCP verso il cloud e se non è stato fatto ne apre una:

Number	Time	Source	Destination	Protocol	Length	Sorce-Destination Port	Info
403	128.254451844	127.0.0.1	127.0.0.1	TCP	74	33850 → 45002	[SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM…
404	128.254484258	127.0.0.1	127.0.0.1	TCP	74	45002 → 33850	[SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=6549…
405	128.254520161	127.0.0.1	127.0.0.1	TCP	66	33850 → 45002	[ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3733810…

Figura 2.8: Cattura Wireshark: *three-way handshake* **Gateway - Cloud**

Successivamente prende tutti i **payload** dei pacchetti contenuti nell'*arp table modificata* e li *unisce* in un unico nuovo pacchetto creato tramite *PacketBuilder*, il quale viene poi serializzato ed inviato al *cloud*. Infine tutti i *valori* del dizionario vengono resettati al *valore di default* ed il *contatore* di clients che hanno inviato il proprio messaggio viene *azzerato*.

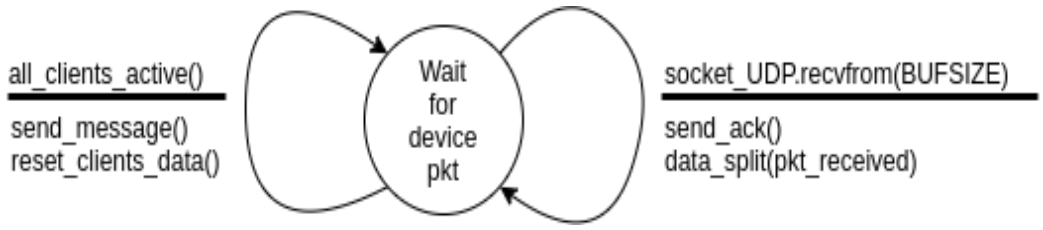

Figura 2.9: Automa a stati finiti rappresentante il gateway

2.2.3 Classi host: Cloud

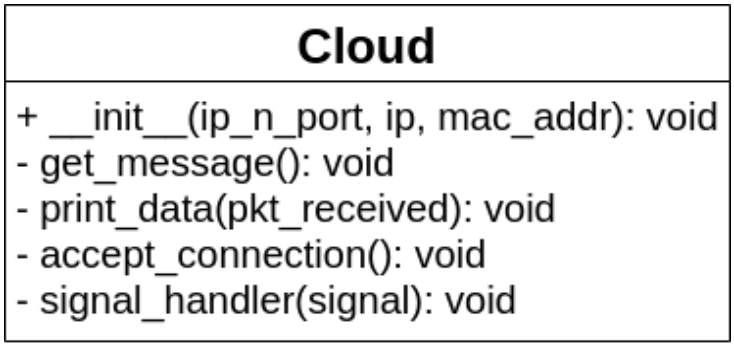

Figura 2.10: Schema UML del Cloud Server

Il *Cloud* rappresenta il nostro server cloud TCP, il suo compito è quello di stampare a video le misurazioni ottenute dai dispositivi IoT.
Attraverso il metodo *accept_connection()* si mette in collegamento con il *Gateway*, di conseguenza aspetta di ricevere il messaggio.

Number	Time	Source	Destination	Protocol	Length	Source-Destination Port	Info
406	128.255114276	127.0.0.1	127.0.0.1	TCP	689	33850 → 45002	[PSH, ACK] Seq=1 Ack=1 Win=65536 Len=623 TSval=…
407	128.255144038	127.0.0.1	127.0.0.1	TCP	66	45002 → 33850	[ACK] Seq=1 Ack=624 Win=64896 Len=0 TSval=37338…

Figura 2.11: Cattura Wireshark: avvenuta consegna pacchetto ridefinito

Quando ottiene il pacchetto elaborato, stampa su una console il payload e il tempo di trasmissione.

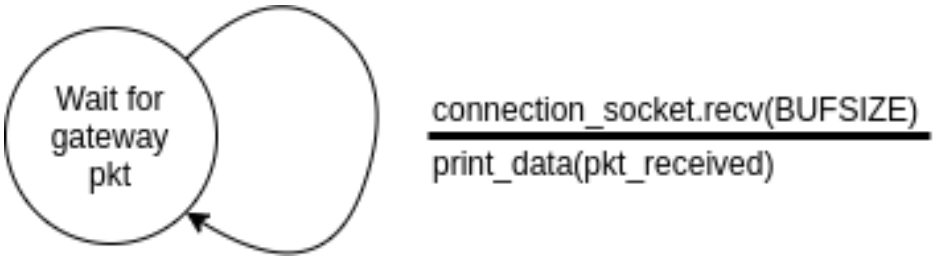

Figura 2.12: Automa a stati finiti raffigurante il comportamento del Cloud

2.2.4 Modulo factories

Nel modulo **factories** sono presenti 3 classi: **DeviceFactory, Gateway-Factory, CloudFactory**, insieme alle costanti che tengono memoria delle

porte UDP e TCP da utilizzare. Tutte e tre condividono lo stesso *pattern* che è il *FactoryMethod*, utile perchè consente di fabbricare oggetti predefiniti che abbiamo poi utilizzato per eseguire la *simulazione* e nella fase di *testing*. Di seguito le varie classi vengono descritte più nel dettaglio.

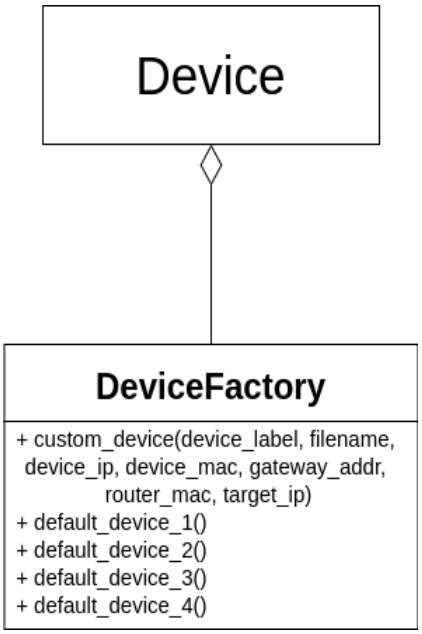

Figura 2.13: Diagramma UML della classe DeviceFactory

La classe *DeviceFactory* presenta 5 metodi pubblici: uno che permette la fabbricazione di un device *"custom"*, mentre i restanti 4 fabbricano i *device di default* utili alla *simulazione*.

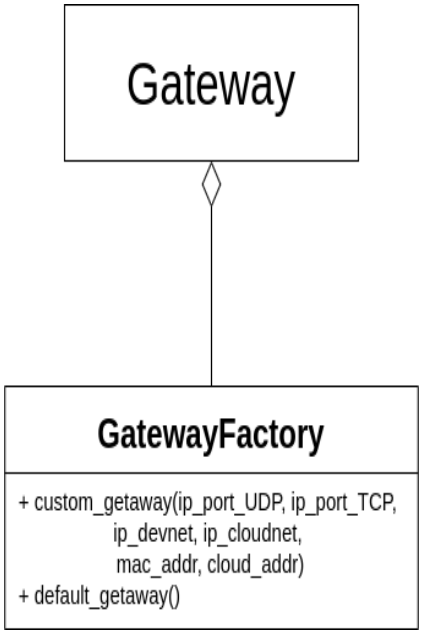

Figura 2.14: Diagramma UML della classe GatewayFactory

La classe GatewayFactory fornisce due metodi pubblici: uno per produrre gateway a piacimento e l'altro per generare il *gateway di default* utile alla *simulazione*.

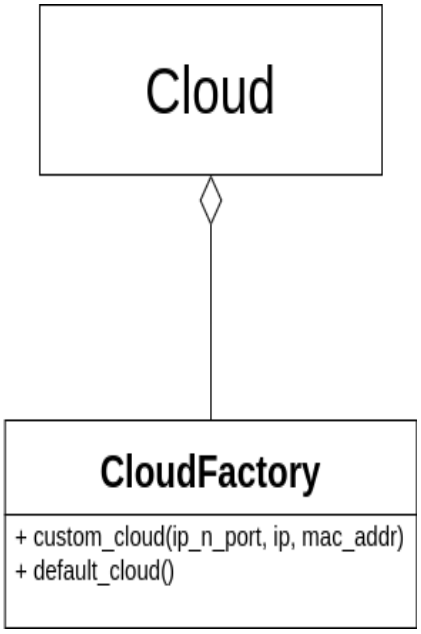

Figura 2.15: Diagramma UML della classe GatewayFactory

La classe CloudFactory contiene due metodi pubblici anch'essa: uno sempre per la fabbricazione di oggetti *custom* e l'altro per la creazione del *server cloud* utile alla *simulazione*.

2.2.5 Note sui buffer utilizzati nei vari canali

I *canali* utilizzati per la comunicazione fra i vari host sono 2 per *device*: uno per l'inoltro delle misurazioni al *gateway* il quale poi apre il canale con lo stesso dispositivo per inviare l'*ACK*. Vi è poi un altro collegamento via *TCP* tra gateway e cloud.

Per quanto riguarda il **buffer** di ogni device, utilizzato per ricevere gli ACK che sono messaggi di dimensione inferiore ai 400 byte, ci è sembrato opportuno impostarli a 1024 byte in modo da non creare un *overhead*.

Vale lo stesso per il **buffer** del *gateway*, dedicato alla ricezione dei pacchetti dei client, in quanto questi ultimi non superano di norma i 500 byte.

Infine, il **buffer** del *cloud* è quello con dimensione maggiore(4096 byte) perchè il *gateway* prima di inviare il pacchetto lo riempie con il payload dei 4 pacchetti arrivati dai device, per cui risulterà più pesante.

Siccome vengono sommati soltanto i *payload*, il Packet risultante non sarà il quadruplo. Nonostante ciò, come osservato nelle **catture di Wireshark** eseguite durante il testing dell'applicativo, il pacchetto si mostra comunque più grande di 1024 Byte per questo è necessario un *buffer più grosso*.

Nota Bene: Le seguenti considerazioni sono valide soltanto per invii periodici dove il periodo è uguale a 60 secondi, come nella nostra simulazione.

Capitolo 3

Sviluppo

3.1 Testing

Dopo aver descritto la struttura del software, andiamo ad analizzare una componente fondamentale del progetto, ovvero il **testing** del programma. Purtroppo, non avendo a disposizione un ambiente per fare testing automatizzato, abbiamo optato per vie più manuali.
Attraverso la creazione di alcuni **script di test** tutti contenuti nella cartella omonima:

- *test_dev1.py*: mette in esecuzione il *device IoT* con IP: 192.168.1.10
- *test_dev2.py*: mette in esecuzione il *device IoT* con IP: 192.168.1.15
- *test_dev3.py*: mette in esecuzione il *device IoT* con IP: 192.168.1.20
- *test_dev4.py*: mette in esecuzione il *device IoT* con IP: 192.168.1.25
- *test_devices.py*: mette in esecuzione tutti e 4 i device, sfruttando il modulo **subprocess**
- *test_gateway.py*: mette in esecuzione il *gateway*
- *test_cloud.py*: mette in esecuzione il *cloud*

I soprastanti script vengono utilizzati anche per la simulazione, devono essere eseguiti nel seguente ordine per funzionare correttamente:

- test_cloud.py >test_gateway.py >test_devices.py
- test_cloud.py >test_gateway.py >test_dev1.py >test_dev2.py >test_dev3.py >test_dev4.py

3.2 Metodologia di lavoro

Durante tutte le fasi del progetto abbiamo lavorato in coppia da **remoto** o in **presenza** coordinandoci nelle diverse operazioni da svolgere. Per la maggior parte del tempo abbiamo lavorato in coppia mentre per la relazione ci siamo divisi i vari paragrafi per poi integrarli alla fine. Inoltre per la fase di programmazione vera e propria ed anche per la stesura di questa relazione ci siamo concentrati insieme su ogni singolo problema.

3.3 Note di sviluppo

Il *linguaggio di programmazione* utilizzato per lo sviluppo del progetto è **Python**.
Sono stati utilizzati dei moduli non studiati in laboratorio:

- il modulo **pickle** che grazie ai metodi *dumps(data)* e *loads(data)* permette di serializzare/deserializzare oggetti;
- il modulo **subprocess** che permette di eseguire più sotto-programmi all'interno di un singolo programma.

Appendice A

Guida utente

Per l'avvio del programma seguire le seguenti istruzioni:

1. Lanciare il cloud in una console specializzata con il comando:
   - python test_cloud.py
2. Lanciare il gateway in una console specializzata con il comando:
   - python test_gateway.py
3. Lanciare i 4 dispositivi in una console specializzata con il comando:
   - python test_devices.py
4. O in alternativa lanciare i 4 device su console differenti:
   - python test_dev1.py
   - python test_dev2.py
   - python test_dev3.py
   - python test_dev4.py

N.B. Prima di eseguire le istruzioni, controllare di essere posizionati sulla cartella 'test'.

Premere **Ctrl+C** su ogni console avviata per interrompere il programma.