Università degli Studi di Firenze
Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

Tesi di Laurea Triennale in Ingegneria Informatica

Analisi e progettazione di una rete multi-paradigma con applicazione all'Internet degli Oggetti Musicali

Candidato
Luca Bindini

Relatore
Prof. Francesco Chiti

Anno Accademico 2019/2020

Indice

Introduzione

"*La musica aiuta a non sentire*
dentro il silenzio che c'è fuori"

Johann Sebastian Bach

Negli ultimi anni grazie al miglioramento delle tecnologie trasmissive, soprattutto dal punto di vista della velocità di connessione, stiamo assistendo al fenomeno dilagante del cosiddetto *Internet of Things* (IoT) ossia l'Internet degli Oggetti.
Qualsiasi oggetto della nostra quotidianità, se opportunamente dotato di sensori e interconnesso con altri dispositivi, può diventare "smart", ovvero può essere in grado di raccogliere dati, elaborarli e comunicarli al resto della rete formata da altri dispositivi, o attuatori, che possono essere eventualmente eterogenei tra loro.
In questo elaborato l'obiettivo principale è quello di estendere la filosofia IoT al mondo della musica andando nel campo dell'*Internet of Musical Things* (IoMusT) ossia l'Internet degli Oggetti Musicali.
Gli strumenti musicali in questo caso non saranno più degli elementi passivi, bensì degli *Smart Musical Instruments* cioè degli oggetti in grado di comunicare con altri strumenti o con il musicista stesso, concetto di *Ambiente Intelligente*.

Sono stati analizzati diversi scenari d'utilizzo con diverse tipologie di musicista (principiante o professionista) in diverse modalità d'esecuzione, brano assegnato oppure Jam Session, facendo particolare attenzione alle possibilità offerte dalle attuali tecnologie e sopratutto ai limiti di quet'ultime.
Proprio in relazione alle varie criticità si andranno infine a vedere possibili prospettive future in grado di mitigarle.

Capitolo 1

Stato dell'Arte

In questo capitolo andremo ad analizzare più da vicino l'*Internet degli Oggetti Musicali* dandone una definizione e caratterizzando i possibili attori in gioco, ovvero i musicisti, e gli *Smart Musical Instruments* da essi utilizzati. Infine verranno proposti alcuni possibili ambiti d'interesse per le applicazioni IoMusT.

1.1 Internet of Musical Things

Come già accennato nell'introduzione, quando parliamo di *Internet degli Oggetti Musicali* (IoMusT) facciamo riferimento all'applicazione della filosofia IoT agli strumenti musicali, tuttavia possiamo provare a darne una definizione più rigorosa.
Secondo il Professor Luca Turchet dell'Università degli Studi di Trento quando si parla di IoMusT ci si riferisce a "l'insieme di interfacce, protocolli e rappresentazioni di informazioni relative alla musica che abilitano servizi e applicazioni che servono uno scopo musicale basato sulle interazioni tra esseri umani e oggetti musicali o tra gli oggetti musicali stessi, nei regni fisici e/o

digitali. Le informazioni relative alla musica si riferiscono ai dati rilevati ed elaborati da un oggetto musicale e/o scambiati con un essere umano o con un altro oggetto musicale".
Quando si parla di *Oggetto Musicale* si fa riferimento ad un dispositivo capace di acquisire, processare, attuare e scambiare dati atti ad uno scopo nell'ambito musicale.
Alcuni esempi di Oggetti Musicali sono gli *Smart Musical Instruments* (SMI) e i *Musical Haptic Wearables* (MHW).

- **Smart Musical Instruments** sono una famiglia di strumenti musicali caratterizzati da un'intelligenza computazionale incorporata, connettività wireless, un sistema di consegna del suono incorporato e un sistema di bordo per il feedback al musicista.
  Sono stati concepiti per offrire una comunicazione diretta punto a punto tra loro e altri dispositivi connessi a reti locali e ad Internet.

- **Musical Haptic Wearables** sono una classe di dispositivi indossabili per artisti e membri del pubblico, che comprendono la stimolazione tattile, il rilevamento dei gesti e le funzionalità di connettività wireless.

1.2 Smart Musical Instruments

Dopo averne dato una breve caratterizzazione nei punti precedenti andiamo a vedere più nel dettaglio cos'è uno *Smart Musical Instrument* (SMI) e il ruolo che esso svolge nella rete IoMusT.
Per farlo possiamo prendere come esempio la Sensus Smart Guitar, sviluppata dalla MIND Music Labs, che è una chitarra hollow-body potenziata con l'elaborazione a bordo, un sistema di più attuatori collegati alla cassa armonica, diversi sensori incorporati in varie parti dello strumento e comunicazione wireless interoperabile.

Figura 1.1: Sensus Smart Guitar della MIND Music Labs

Nella rete, che verrà descritta successivamente in questo elaborato, ogni SMI dovrà necessariamente essere dotato di:

- **Sensore rilevatore di BPM** che è in grado di rilevare il BPM, battiti per minuto, istantaneo dello SMI su cui esso è installato.
  Il valore delle misurazioni effettuate è inviato periodicamente ad un altro SMI o ad un apposito Server per SMI, il quale elaborerà i valori ricevuti e potrà notificare i musicisti (vedremo in seguito ulteriori dettagli su questa parte).

- **Modulo Wi-Fi 802.11** è un dispositivo che consente lo scambio di pacchetti mediante tecnologia wireless.
  L'IEEE 802.11 definisce un insieme di standard di trasmissione per reti WLAN con particolare riguardo al livello fisico e MAC del modello ISO/OSI, specificando sia l'interfaccia tra client e base station (o Access-Point) sia le specifiche tra client wireless.

Possono anche essere presenti altri sensori e/o attuatori a bordo dei nostri SMI, come ad esempio un display LCD dove il musicista può visualizzare eventuali messaggi e correzioni.
In alternativa possiamo avere un attuatore che esegue un feedback aptico come forma di notifica al musicista, avvicinandoci alla filosofia del cosiddetto *Internet tattile*.

1.3 Possibili Applicazioni nella Rete IoMusT

Diversi tipi di applicazioni e servizi possono essere costruiti nella rete IoMusT, che si rivolgono a utenti come musicisti (ad esempio compositori, esecutori e direttori), ingegneri del suono e membri del pubblico.
Tali applicazioni e servizi possono avere una natura interattiva o non interattiva.
Nelle applicazioni musicali interattive i calcoli in tempo reale hanno una particolare importanza (applicazioni bloccanti), per questo motivo le latenze devono necessariamente essere molto basse.
L'infrastruttura IoMusT abilita ecosistemi di dispositivi interoperabili che collegano i musicisti tra loro, così come con il pubblico, il che moltiplica le possibilità di interazione sia in contesti localizzati sia remoti.
Possiamo immaginare scenari su piccola e larga scala.
Un esempio di scenario su piccola scala potrebbe essere una performance musicale localizzata in una sala da concerto (rete LAN) dove vi è cooperazione tra i musicisti ed è questo lo scenario che verrà prevalentemente analizzato in questo elaborato.
Se invece pensassimo ad uno scenario su più larga scala, rete MAN o addirittura GAN, potremmo avere performance musicali online aperte che raccolgono migliaia di partecipanti in un ambiente virtuale.

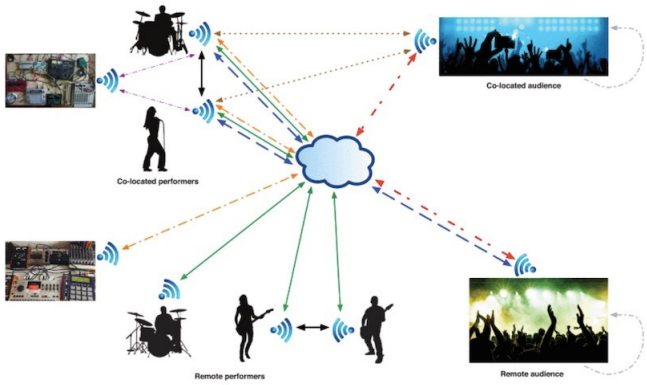

Figura 1.2: Rappresentazione di un possibile ecosistema nella rete IoMusT

Come si può vedere nel diagramma, la comunicazione tra i musicisti è mediata dagli SMI e si nota la presenza di membri del pubblico, ma può essere ulteriormente esteso per includere altri attori, come ingegneri del suono dal vivo, produttori musicali, compositori o direttori d'orchestra.
Le interazioni possono essere sia localizzate, quando i partecipanti si trovano nello stesso spazio fisico (ad esempio, sala da concerto, spazio pubblico), o remote, quando avvengono in diversi spazi fisici collegati da una rete.
Per quanto riguarda le interazioni localizzate, queste possono essere basate su comunicazioni punto-punto tra due SMI (mediate eventualmente da un Access-Point) oppure possono essere dirette ad un apposito server per Smart Musical Instruments, il quale sarà il responsabile di tutta l'elaborazione, della gestione dei dati e delle varie notifiche e correzioni ai musicisti.
Il fatto che l'elaborazione venga fatta da un server centrale o dagli SMI stessi determina se ci troviamo di fronte ad uno scenario di tipo *Fog Computing* oppure *Edge Computing*.
Nel prossimo capitolo, dove entreremo nel merito del progetto realizzato, andremo ad analizzare le differenze, vantaggi e svantaggi, tra questi due tipi di approccio differenti.

Capitolo 2

Progettazione di una Rete IoMusT

In questo capitolo andremo a vedere la progettazione di una rete IoMusT andando in primis a caratterizzare gli attori che ne prendono parte, cioè i musicisti stessi che sono stati modellati in due diverse categorie: principianti e professionisti.
Gli SMI avranno un comportamento diverso alla ricezione di notifiche e/o correzioni a seconda del fatto che essi siano utilizzati da una diversa categoria di musicisti.
Il parametro che verrà principalmente preso in considerazione e su cui verranno apportate eventuali correzioni sono i *battiti per minuto* (BPM) ovvero l'unità di misura di frequenza, utilizzata per l'indicazione metronomica in musica, ma in generale il report (beacon) che riceverà lo SMI periodicamente sarà multiparametrico (ritmo, tono, etc.).
Il valore rilevato di BPM, mediante un sensore a bordo degli SMI, viene inviato, ad altri SMI o ad un Server centralizzato, attraverso *pacchetti UDP* (vedremo in seguito le motivazioni che ci hanno portato a tale scelta) così

come i pacchetti di notifica/correzione che arrivano agli SMI stessi.
Verranno analizzati due principali scenari nella rete IoMusT:

- **Esecuzione di un brano prestabilito** in questo caso la rete è composta da un gruppo eterogeneo di musicisti, sia principianti sia professionisti.

  Tutti i musicisti suonano uno stesso brano con BPM definito a priori commettendo eventualmento un errore, che è proprio quello che la rete vuole andare a correggere.

  Per fare ciò si introduce un Server centralizzato che ha lo scopo di raccogliere le informazioni e notificare i vari SMI.

  Il paradigma di alto livello utilizzato in questo scenario è il *Client-Server*.

- **Jam Session** in questo caso la rete è composta da un gruppo di musicisti professionisti che suonano insieme senza alcun tipo di Server che funga da broker.

  I musicisti seguiranno il BPM di un musicista scelto come "riferimento" che ovviamente non è statico e deciso a priori, bensì tempo-variante.

  Il paradigma di alto livello utilizzato in questo scenario è il *Peer-to-Peer* (P2P).

2.1 Categorie di Musicisti

Nella modellazione della rete vengono distinti i musicisti in due categorie ben definite:

- Musicista Principiante
- Musicista Professionista

2.1.1 Musicista Principiante

Il musicista principiante è stato modellato assumendo che possa commettere un solo tipo di errore additivo, ossia se non notificato/corretto il musicista alle prime armi tenderà ad accelerare rispetto al BPM originale.
In particolare si assume che possa aumentare di 1 BPM ad ogni battuta con una probabilità del 50%, errore sistematico (correlato) ossia dipende dall'errore agli istanti precedenti.

2.1.2 Musicista Professionista

Il musicista professionista è stato modellato assumendo che possa commettere un errore avente una distribuzione di probabilità *discreta bilaterale uniforme* centrata nello zero e con una varianza molto bassa, data dal fatto che essendo professionista si suppone che non possa commettere errori grossolani.
In particolare il BPM potrà variare di ± 2 BPM rispetto a quello di riferimento, questo errore a differenza del caso precedente è incorrelato (casuale).

2.2 Formato del Messaggio

I messaggi scambiati dagli SMI, sia in entrata che in uscita, sfruttano tutti il protocollo di livello di Trasporto UDP descritto brevemente qui di seguito.

2.2.1 Protocollo UDP

Lo *User Datagram Protocol*, abbreviato UDP, è un protocollo del livello di Trasporto nella pila TCP/IP che consente l'invio senza connessione di datagrammi nelle reti basate su IP.
Per raggiungere i servizi desiderati sugli host di destinazione, il protocollo ricorre alle porte, elencate come uno dei componenti principali nell'intestazione UDP.
Con lo User Datagram Protocol un'applicazione può inviare delle informazioni molto velocemente, in quanto non occorre creare una connessione con il destinatario né attendere una risposta al contrario dell'altro protocollo principale a livello Trasporto ossia il TCP.
Tuttavia, essendo un protocollo non affidabile, non vi sono garanzie che i pacchetti arrivino interi e nella stessa sequenza in cui sono stati inviati.
I pacchetti difettosi sono riconoscibili da una checksum (somma di controllo) opzionale. Nonostante queste possibili problematiche, si è scelto di utilizzare questo protocollo per la velocità e semplicità d'utilizzo, la perdita di qualche sporadico campione non comprometterebbe in modo significativo il comportamento della rete.

Vediamo adesso brevemente la struttura del pacchetto UDP:

+	Bit 0-15	16-31
0	Source Port (optional)	Destination Port
32	Length	Checksum (optional)
64+	Data	

Figura 2.1: Struttura Pacchetto UDP

- **Header**
  - **Source port** [16 bit] identifica il numero di porta sull'host del mittente del datagramma
  - **Destination port** [16 bit] identifica il numero di porta sull'host del destinatario del datagramma
  - **Length** [16 bit] contiene la lunghezza totale in bytes del datagramma UDP (header+dati)
  - **Checksum** [16 bit] somma di controllo per il rilevamento degli errori;
- **Payload**
  - **Data** contiene i dati del messaggio

2.2.2 Tipologie di Messaggio

I messaggi inviati nella rete (esclusi gli ACK e i vari pacchetti Beacon dell'Access-Point) sono essenzialmente di due tipi:

- **pacchetti inviati dagli SMI** sono dei pacchetti UDP inviati dagli SMI al Server centralizzato o ad un altro SMI nel caso di una Jam Session.
  Questi pacchetti, inviati con cadenza periodica, oltre alle informazioni date dai campi dell'header del pacchetto UDP contengono le seguenti informazioni:
  - nome del SMI mittente
  - tempo di invio del pacchetto (utile ai fini delle statistiche sulla latenza)
  - BPM misurato dal sensore
- **notifiche/correzioni ricevute dagli SMI** sono dei pacchetti UDP inviati periodicamente dal Server o dal musicista di riferimento in una Jam Session che si basano sui dati ricevuti dagli SMI e possono avere una duplice azione:
  - **notifica correttiva** questo nel caso in cui il destinatario sia un musicista principiante.
    Il pacchetto ricevuto andrà a correggere il musicista in modo automatico (con un attuatore) o manuale (deve essere il musicista ad adattarsi al BPM ricevuto).
  - **notifica informativa** questo nel caso in cui il destinatario sia un musicista professionista.

Il pacchetto ricevuto andrà ad informare il musicista del suo andamento facendogli capire se sta andando troppo veloce o troppo piano, e possibilmente darà anche altri informazioni riguardanti ad esempio il tono.

Tuttavia non apporterà alcuna azione correttiva automatica in quanto trattandosi di un musicista professionista potrebbe essere una scelta prettamente interpretativa non un errore.

Queste notifiche vengono inviate all'inizio di ogni nuova battuta e si basano sull'andamento dello SMI nella battuta precedente, facendo un filtraggio tra i BPM dei pacchetti che ha ricevuto e confrontando con il BPM di riferimento.

2.3 Rete Standard per Smart Musical Instruments

La rete descritta qui di seguito realizza lo scenario d'esecuzione di un brano prestabilito, dove musicisti principianti e professionisti suonano una stessa canzone decisa a priori con un BPM ben preciso, utilizzando come paradigma di alto livello il *Client-Server*.

2.3.1 Topologia e Componenti della Rete

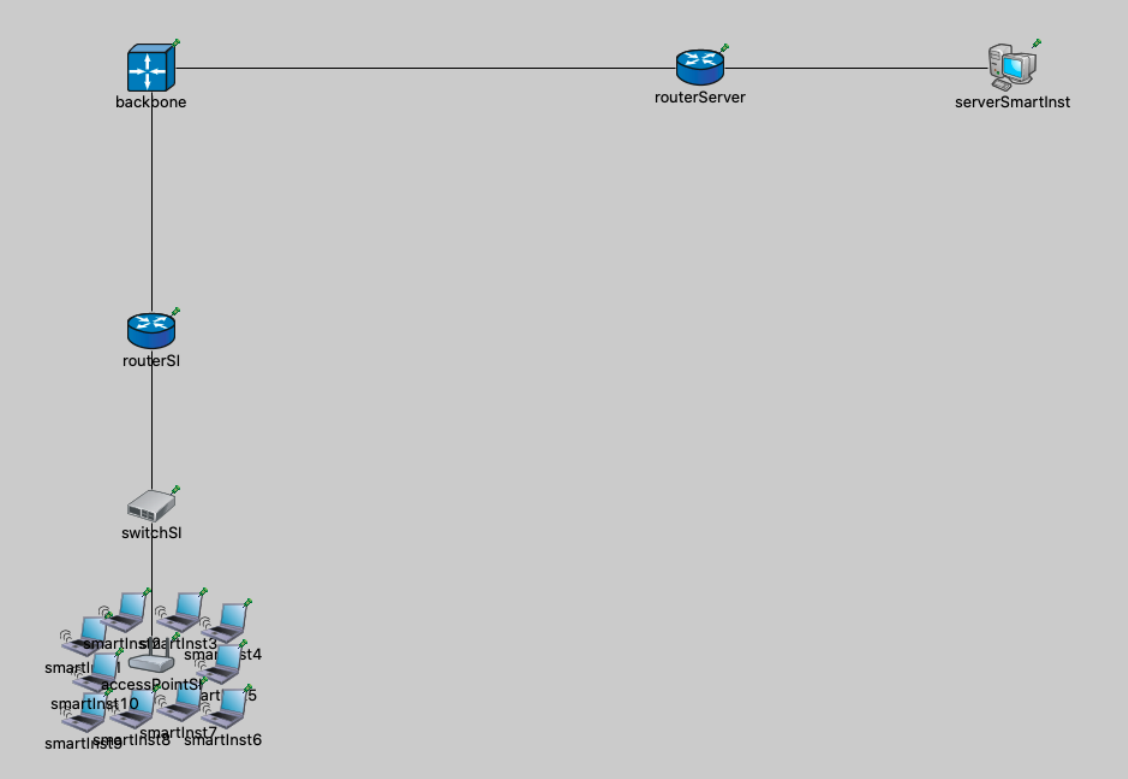

Figura 2.2: Screenshot della struttura della rete Client-Server presa dall'ambiente usato per le simulazioni

L'intera rete può essere considerata *locale* in quanto a livello di estensione si parla di circa 1 Km (distanza dalla sala degli SMI al Server centrale). Analizzando la figura che mostra la topologia della rete possiamo notare varie regioni e categorie di elementi di rete che verranno brevemente descritti qui di seguito:

- **Smart Musical Instrument:** tutti gli SMI, sia quelli per principianti sia per professionisti, si trovano in una sala da concerto di circa 100 m^2.

  Sono modellati come *Wireless Host* che si connettono ad un Access-Point, posto al centro della stanza, grazie al quale inviano e ricevono pacchetti UDP.

- **Server per SMI:** è un semplice *Standard Host* che si occupa di ricevere i dati dagli SMI, analizzarli e inviare le opportune correzioni, ovviamente attraverso pacchetti UDP.

- **Access-Point:** è un dispositivo di rete che mette a disposizione una rete WiFi al quale è possibile collegarsi con gli SMI per poter accedere alla rete locale.

  Il collegamento alla LAN, viene effettuato tramite un cavo collegato al *Router* principale, il quale svolge il compito di instradare il traffico.

  Nella rete è stato aggiunto anche uno *Switch* per poter avere scalabilità (possibilità di avere più LAN diverse connesse allo stesso Router).

- **Collegamento Cablato:** sono stati utilizzati dei cavi *Ethernet cat. 8* che permettono velocità fino a 100 Mbps.

- **Router:** è un dispositivo di rete usato come interfacciamento tra sottoreti diverse, eterogenee e non, che lavorando a livello logico come nodo interno di rete deputato alla commutazione di livello 3 del modello OSI o del livello 2 Internet nel modello TCP/IP, si occupa di instradare i pacchetti dati fra tali sottoreti permettendone l'interoperabilità a livello di indirizzamento.

  Nella rete sono presenti due Router: quello nella LAN degli SMI e quello nella LAN del Server.

- **Backbone Router:** è un Router ad alta velocità necessario ad avvicinare due server o due Router che a loro volta hanno la funzione di smistare informazioni.

  Nella rete è stato utilizzato per il collegamento tra i due Router principali.

2.3.2 Procedure della Rete

Gli SMI inviano ogni 100 ms il valore attualmente misurato dal sensore rilevatore di BPM al Server centrale, il quale elabora i dati facendo una media sui campioni della battuta appena passata e invia i pacchetti correttivi agli SMI all'inizio di ogni nuova battuta.
A questo punto se il musicista che riceve la notifica è un principiante si ha una correzione, che può essere automatica o meno come descritto in precedenza. In caso contrario, ossia per musicisti professionisti, la notifica avrà soltanto un ruolo informativo sull'andamento.

2.4 Rete Jam Session per Smart Musical Instruments

La rete descritta qui di seguito realizza lo scenario d'esecuzione di una Jam Session, dove musicisti professionisti suonano assieme improvvisando, utilizzando come paradigma di alto livello il *Peer-to-Peer*.

2.4.1 Topologia e Componenti della Rete

Figura 2.3: Screenshot della struttura della rete Peer-to-Peer presa dall'ambiente usato per le simulazioni

Tutti gli SMI si trovano in una sala da concerto di circa 100 m^2 con un Access-Point centrale per la comunicazione tra essi e sono modellati come *Wireless Host*.

2.4.2 Procedure della Rete

Esattamente come nel caso precedente gli SMI inviano ogni 100 ms il valore attualmente misurato dal sensore rilevatore di BPM a un altro SMI (invece che ad un Server) che a differenza degli altri svolge il ruolo di *Sink*, il quale elabora i dati facendo una media sui campioni della battuta appena passata, confronta il valore con il proprio BPM attuale e invia i pacchetti correttivi agli SMI all'inizio di ogni nuova battuta.
A questo punto i musicisti che ricevono la notifica devono adattarsi, a seconda delle correzioni, per uniformare il proprio BPM con quello del Sink SMI.

Capitolo 3

Ambiente di Sviluppo e Framework

In questo capitolo andremo ad analizzare l'ambiente di sviluppo con il quale la rete è stata modellata: *OMNeT++*.
In particolare per la realizzazione è stato utilizzato il framework *INET* che estende le capacità di OMNeT++ aggiungendone elementi fondamentali come ad esempio la possibilità di utilizzare collegamenti wireless e sopratutto applicazioni UDP che hanno consentito lo sviluppo della parte applicativa del progetto.
Infine andremo a vedere più nel dettaglio l'implementazione vera e propria focalizzandoci sulle varie classi del progetto.

3.1 OMNeT++

Andiamo brevemente a descrivere l'ambiente in questione e le sue caratteristiche d'utilizzo.

3.1.1 Descrizione dell'Ambiente

OMNeT++ è una piattaforma di simulazione di reti:

- **Modulare** con una gerarchia tra i vari moduli.
- **Ad Eventi** il flusso del programma è largamente determinato dal verificarsi di eventi esterni.
- **Orientata agli Oggetti** il linguaggio utilizzato è infatti il C++.
- **Open Source** non protetta da copyright e liberamente modificabile dagli utenti.

Il fatto di utilizzare come linguaggio di base il C++ porta con sé alcuni vantaggi come l'ereditarietà multipla tra classi ed il fatto di poter riutilizzare tutto il codice già presente per il linguaggio C.
D'altra parte il C++ ha una sintassi molto pesante ed a differenza di linguaggi come il Python, che è un linguaggio interpretato, necessita la compilazione (operazione spesso onerosa in progetti grandi).
OMNeT++ mette a disposizione una libreria di classi C++, da cui normalmente lo sviluppatore estende le proprie come è essenzialmente avvenuto in questo progetto.

3.1.2 Tipologie di Files

I files che vengono utilizzati in OMNeT++ sono essenzialmente di 3 tipologie: .cc, .ned ed .ini.
Andiamo brevemente ad analizzare cosa essi rappresentano:

- **File .cc** sono le classi C/C++ che possono estendere quelle della libreria nativa di OMNeT++, o come vedremo in seguito di INET, oppure classi C/C++ "standard".

- **File .ned** acronimo di Network Description Language, serve a dare una descrizione di un determinato modulo nella rete, dove per modulo si può intendere:

  - la rete stessa
  - un elemento della rete
  - un particolare tipo di connessione
  - un particolare tipo di messaggio

- **File .ini** servono a descrivere le simulazioni nelle quali vengono specificati i parametri di moduli, connessioni e le caratteristiche della simulazione stessa come la durata, il nome identificativo, ecc.
  All'interno dello stesso file .ini si possono avere più tipi di configurazione diversa, ad esempio nel nostro caso sono stati descritti i due scenari d'utilizzo che abbiamo visto precedentemente (paradigma Client-Server e P2P).

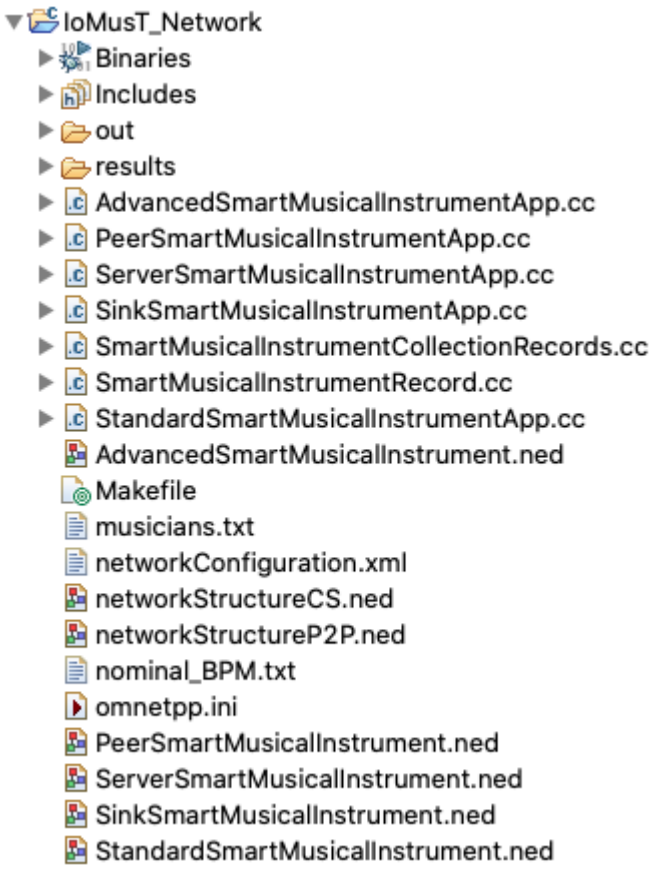

Figura 3.1: Elenco dei files del progetto

Come si può notare dall'immagine, nel progetto sono presenti anche alcuni file .txt che sono stati utilizzati per l'inizializzazione di parametri della rete, come il BPM iniziale e la lista degli SMI presenti.
Molto importante è il file *networkConfiguration.xml* che è stato utilizzato per l'assegnazione statica degli IP della rete e per le tabelle statiche di routing interne ai router.

3.2 INET Framework

INET Framework è una libreria di modelli open source per l'ambiente di simulazione OMNeT ++. Fornisce protocolli, agenti e altri modelli per ricercatori e studenti che lavorano con reti di comunicazione.
INET è particolarmente utile quando si progettano e convalidano nuovi protocolli o si esplorano scenari nuovi o esotici.
INET contiene modelli per lo stack Internet (TCP, UDP, IPv4, IPv6, OSPF, BGP, ecc.), protocolli di livello di collegamento cablato e wireless (Ethernet, PPP, IEEE 802.11, ecc.), supporto per la mobilità, protocolli MANET, DiffServ, MPLS con segnalazione LDP e RSVP-TE, diversi modelli di applicazioni e molti altri protocolli e componenti.
Nel progetto il framework è stato utile sia per quanto riguarda la componentistica di rete, pensiamo all'Access-Point o ai *Wireless Host* che modellano gli SMI, sia per quanto riguarda la parte applicativa.
A bordo di ogni SMI è presente un'applicazione UDP che estende la *UdpBasicApp* fornita da INET e realizza l'intera logica applicativa che vedremo successivamente nella sezione dedicata all'implementazione.
Ad ogni classe che rappresenta un SMI è necessario associare il suo corrispettivo modulo descritto in un file .ned (come si può notare dall'immagine).
Qui di seguito andiamo a vedere alcuni metodi della classe *UdpBasicApp* di INET sui quali poi verrà fatto override dalle classi derivate nel progetto.

```
// UdpBasicApp.cc
#include "inet/applications/base/ApplicationPacket_m.h"
#include "inet/applications/udpapp/UdpBasicApp.h"
#include "inet/common/packet/Packet.h"
#include "inet/networklayer/common/FragmentationTag_m.h"
#include "inet/networklayer/common/L3AddressResolver.h"
#include "inet/transportlayer/contract/udp/UdpControlInfo_m.h"

Define_Module(UdpBasicApp);

void UdpBasicApp::initialize(int stage)
{
    ApplicationBase::initialize(stage);

    if (stage == INITSTAGE_LOCAL) {
        numSent = 0;
        numReceived = 0;
        WATCH(numSent);
        WATCH(numReceived);

        localPort = par("localPort");
        destPort = par("destPort");
        startTime = par("startTime");
        stopTime = par("stopTime");
        //----CONVERTING RETURN VALUE OF par("packetName") TO A
            NON-CONST TYPE----
        const char *const_string = par("packetName");
        const size_t n = strlen(const_string);
```

```
        char *new_string = new char[n + 1]{};
        std::copy_n(const_string, n, new_string);
        packetName = new_string;
        //-------------------------------------------------------
        dontFragment = par("dontFragment");
        if (stopTime >= SIMTIME_ZERO && stopTime < startTime)
            throw cRuntimeError("Invalid startTime/stopTime
                parameters");
        selfMsg = new cMessage("sendTimer");
    }
}

L3Address UdpBasicApp::chooseDestAddr() {
    int k = intrand(destAddresses.size());
    if (destAddresses[k].isUnspecified() ||
        destAddresses[k].isLinkLocal()) {
        L3AddressResolver().tryResolve(destAddressStr[k].c_str(),
            destAddresses[k]);
    }
    return destAddresses[k];
}

void UdpBasicApp::sendPacket()
{
    std::ostringstream str;
    str << packetName << "-" << numSent;
    Packet *packet = new Packet(str.str().c_str());
    if(dontFragment)
        packet->addTag<FragmentationReq>()->setDontFragment(true);
```

```
    const auto& payload = makeShared<ApplicationPacket>();
    payload->setChunkLength(B(par("messageLength")));
    payload->setSequenceNumber(numSent);
    payload->addTag<CreationTimeTag>()->setCreationTime(simTime());
    packet->insertAtBack(payload);
    L3Address destAddr = chooseDestAddr();
    emit(packetSentSignal, packet);
    socket.sendTo(packet, destAddr, destPort);
    numSent++;
}

void UdpBasicApp::processPacket(Packet *pk)
{
    emit(packetReceivedSignal, pk);
    EV_INFO << "Received packet: " <<
        UdpSocket::getReceivedPacketInfo(pk) << endl;
    delete pk;
    numReceived++;
}
```

3.3 Implementazione

Per la realizzazione della rete IoMusT nei due possibili scenari d'utilizzo sono state scritte classi C++, file Ned e file Ini.
Grazie alle enormi potenzialità offerte dell'ambiente OMNeT++ e sopratutto da INET, la mappatura del progetto della rete descritta nel Capitolo 2 è stata quasi 1:1.
Vediamo adesso alcuni dettagli implementativi per quanto riguarda i due diversi scenari.

3.3.1 Implementazione Rete Standard

Nel caso della rete standard dove tutti gli SMI eseguono un brano predefinito e vi è un Server centrale per le correzioni, paradigma Client-Server, oltre alla modellazione della rete, fatta nel corrispettivo file .ned, sono state realizzate delle classi per i vari SMI, sia suonati da principianti sia suonati da professionisti, che estendono la *UdpBasicApp* ed altre classi fondamentali ai fini della logica applicativa.
Le principali classi in questo scenario sono:

- **StandardSmartMusicalInstrumentApp** è l'applicazione UDP presente sugli strumenti suonati da musicisti principianti.
- **AdvancedSmartMusicalInstrumentApp** è l'applicazione UDP presente sugli strumenti suonati da musicisti professionisti.
- **ServerSmartMusicalInstrumentApp** è l'applicazione UDP presente sul Server centrale per SMI.
- **SmartMusicalInstrumentRecord** rappresenta il singolo record inviato dagli SMI.

- **SmartMusicalInstrumentCollectionRecords** rappresenta la collezione di records dell'ultima battuta per ogni SMI.

Per capire meglio la logica applicativa, è utile la visione di un *UML Class Diagram* che sintetizza le relazioni che intercorrono tra le varie classi.

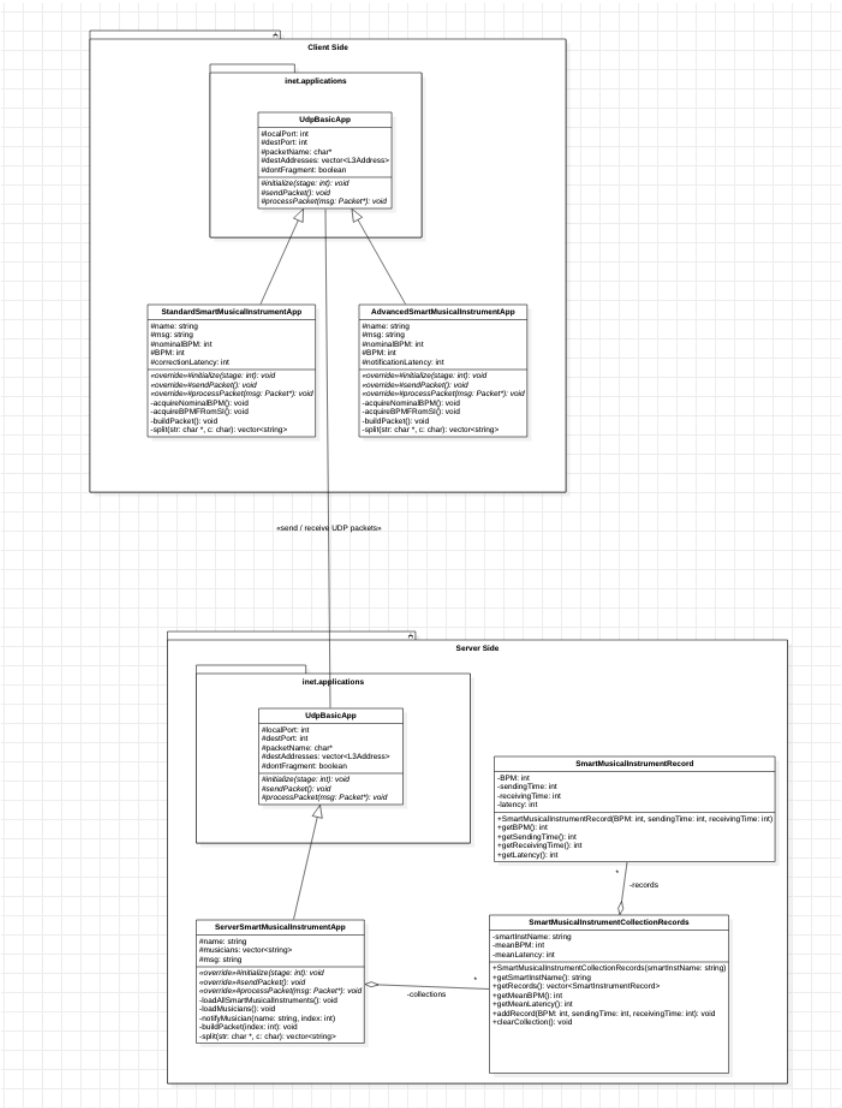

Figura 3.2: UML Class Diagram

Qui di seguito dei frammenti importanti di codice dove viene fatto l'override di alcuni metodi della *UdpBasicApp* da parte delle classi derivate.

```
//StandardSmartMusicalInstrumentApp.cc
#include "inet/applications/udpapp/UdpBasicApp.h"
#include <fstream>
#include <string.h>

virtual void sendPacket() override {
    buildPacket();
    UdpBasicApp::sendPacket();
}

virtual void processPacket(inet::Packet *pk) override {
    std::vector<std::string> result = split(pk->getName(), '-');
    UdpBasicApp::processPacket(pk);
    BPM -= (std::stoi(result.at(1))-nominalBPM);
    correctionLatency = int(omnetpp::simTime().dbl()*1000) -
        std::stoi(result.at(2));
}

void buildPacket() {
    msg = "-";
    acquireBPMFromSI();
    msg+= std::to_string(BPM);
    msg+= "-";
    msg += std::to_string(int(omnetpp::simTime().dbl()*1000));
    strcpy(packetName, (name + msg).c_str());
}
```

```
//ServerSmartMusicalInstrumentApp.cc
#include "inet/applications/udpapp/UdpBasicApp.h"
#include "inet/networklayer/common/L3AddressResolver.h"
#include <fstream>
#include <string.h>
#include "SmartMusicalInstrumentCollectionRecords.cc"

virtual void sendPacket() override {
   for(int i = 0; i < musicians.size(); i++){
        notifyMusician("smartInst" + std::to_string(i+1), i);
        collections.at(i).clearCollection();
    }
    firstSent = false
}

virtual void processPacket(inet::Packet *pk) override {
    std::vector<std::string> result = split(pk->getName(), '-');
    for(int i = 0; i < musicians.size(); i++)
        if(collections.at(i).getSmartInstName() == result.at(0)) {
            collections.at(i).addRecord(std::stoi(result.at(1)),
                std::stoi(result.at(2)),
                int(omnetpp::simTime().dbl()*1000));
        }
    UdpBasicApp::processPacket(pk);
}
```

3.3.2 Implementazione Rete Jam Session

Nel caso della rete Jam Session dove tutti i musicisti professionisti suonano insieme seguendo un musicista nominato come *Sink*, paradigma Peer-to-Peer, esattamente come nel caso precedente, oltre alla modellazione della rete, fatta nel corrispettivo file .ned, sono state realizzate delle classi per i vari SMI che estendono la *UdpBasicApp* ed altre classi fondamentali ai fini della logica applicativa.
Le principali classi in questo scenario, oltre a quelle necessarie alla logica applicativa (ad esempio la gestione dei records) che abbiamo già visto nello scenario precedente sono:

- **PeerSmartMusicalInstrumentApp** è l'applicazione UDP presente sugli strumenti suonati dai musicisti.
- **SinkSmartMusicalInstrumentApp** è l'applicazione UDP presente sullo strumento del musicista che in quel momento ha il "comando".

Qui di seguito dei frammenti di codice della classe *SinkSmartMusicalInstrumentApp* che estende la classe *UdpBasicApp*.

```
//SinkSmartMusicalInstrumentApp.cc
#include "inet/applications/udpapp/UdpBasicApp.h"
#include "inet/networklayer/common/L3AddressResolver.h"
#include <fstream>
#include <string.h>
#include "SmartMusicalInstrumentCollectionRecords.cc"

virtual void sendPacket() override {
   changeBPM();
    for(int i = 0; i < (musicians.size()-1); i++){
```

```
        notifyMusician("smartInst" + std::to_string(i+1), i);
        collections.at(i).clearCollection();
    }
    firstSent = false;
}

void changeBPM() {
    oldBPM = BPM;
    BPM = (initialBPM-10) + (rand() %
        static_cast<int>((initialBPM+10) - (initialBPM-10) + 1));
}
```

A differenza del caso precedente, dove il Server non aveva un BPM proprio, in questo caso i pacchetti correttivi che invierà il Sink dipenderanno dal suo BPM.

Capitolo 4

Test e Risultati

In questo capitolo andremo a effettuare alcuni test sui vari scenari della rete realizzata, commentando e confrontando i risultati ottenuti.

4.1 Latenza di Ricezione delle Correzioni

Un primo parametro cruciale che possiamo andare ad analizzare è il *tempo di latenza* dei pacchetti correttivi, inviati dal Server o dal musicista Sink. Se vogliamo che la rete funzioni in modo corretto la latenza deve essere molto bassa, in quanto è necessario che i pacchetti arrivino all'inizio di ogni battuta con un ritardo di pochi ms.

4.1.1 Latenza Rete Standard

Per valutare la latenza nel caso della rete standard con paradigma Client-Server sono state effettuate alcune prove come variare la distanza tra gli SMI ed il Server centrale oppure aumentare il numero degli SMI.

Per quanto concerne la scalabilità spaziale, se manteniamo la rete con tecnologia cablata le distanze possono arrivare nell'ordine di qualche decina di Km con ritardi non significativi.
Un punto sicuramente più critico è il numero degli SMI, infatti come si può notare dal grafico riportato di seguito, all'aumentare del numero di essi, aumenta notevolmente la latenza di ricezione dei pacchetti correttivi.

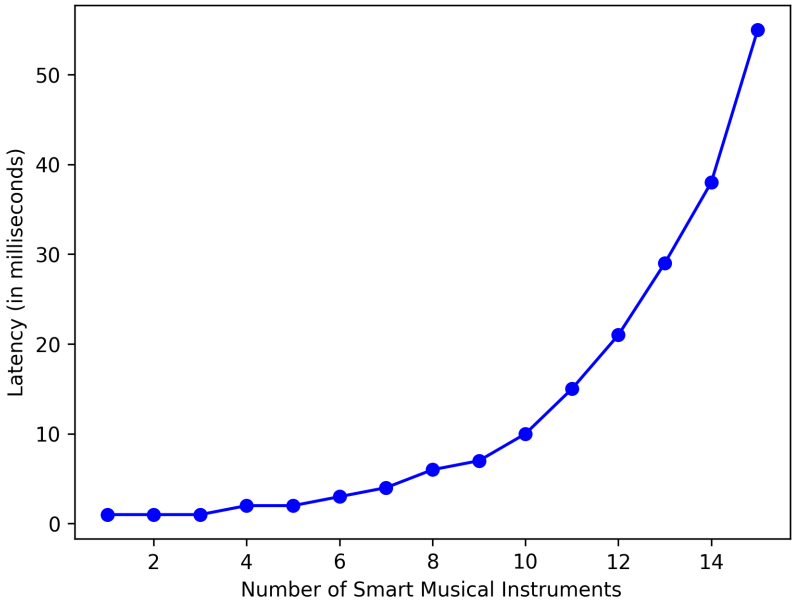

Figura 4.1: Latenza dei pacchetti correttivi al variare del numero di SMI

Questo avviene principalmente a causa dell'Access-Point wireless che fa da "collo di bottiglia" all'aumentare del numero di nodi connessi ad esso.

4.1.2 Confronto tra Latenza Rete Standard e Rete Jam Session

Interessante è il confronto tra la latenza precedentemente vista nel caso della rete standard e la latenza nel caso della Jam Session (con i corrispettivi paradigmi di alto livello Client-Server e P2P).
Vediamo adesso un grafico che mette a confronto i due paradigmi.

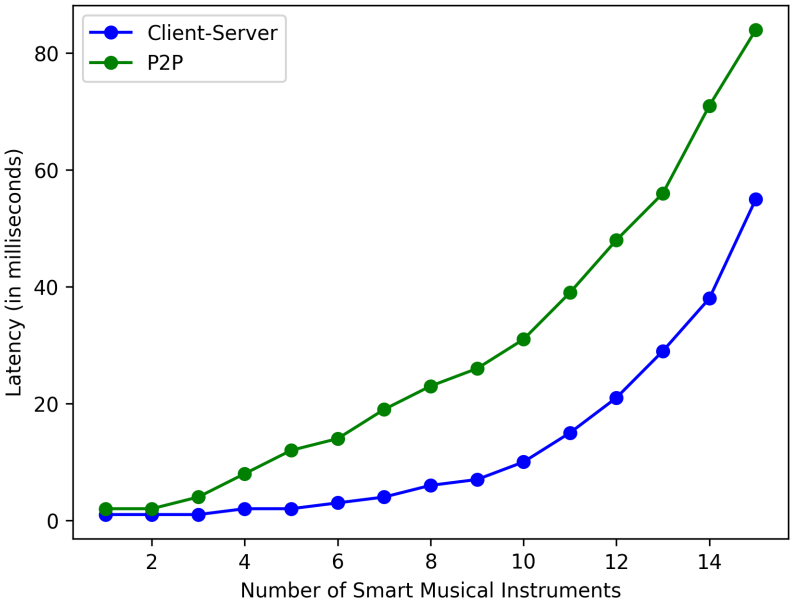

Figura 4.2: Latenza dei pacchetti correttivi al variare del numero di SMI nei diversi scenari

Come possiamo osservare dal grafico nel caso della Jam-Session (P2P) la latenza è più alta a causa del maggior numero di messaggi che transitano all'interno della rete tramite l'Access-Point locale.

4.2 Correzione di uno Smart Musical Instrument

Supponendo di avere un BPM nominale di 80 con un metro di 4/4, ogni battuta durerà 3 secondi, quindi il Server, o il Sink, dovrà inviare il pacchetto correttivo ad ogni SMI ogni 3 secondi basandosi sui campioni ricevuti alla battuta precedente.
Ricordiamo che ogni SMI invia il proprio valore di BPM rilevato ogni 100 ms, quindi in linea teorica, al netto di eventuali perdite di pacchetto, il Server per la correzione si baserà sugli ultimi 30 campioni ricevuti.

4.2.1 Correzione di un Musicista Principiante

Vediamo di seguito alcuni grafici che mostrano l'andamento temporale di uno SMI suonato da un musicista principiante con e senza correzione da parte del Server centrale per SMI, con brano assegnato a priori avente BPM nominale di 80.
Ricordiamo che, per come è stato modellato il musicista principiante, egli tenderà ad accelerare andando ad aumentare i propri BPM se non opportunamente corretto.

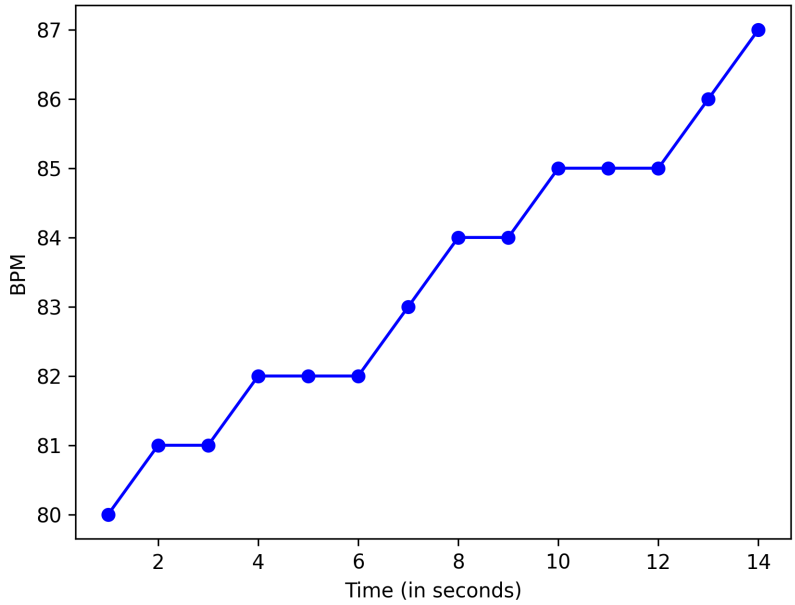

Figura 4.3: Andamento temporale del BPM di un musicista principiante senza correzione

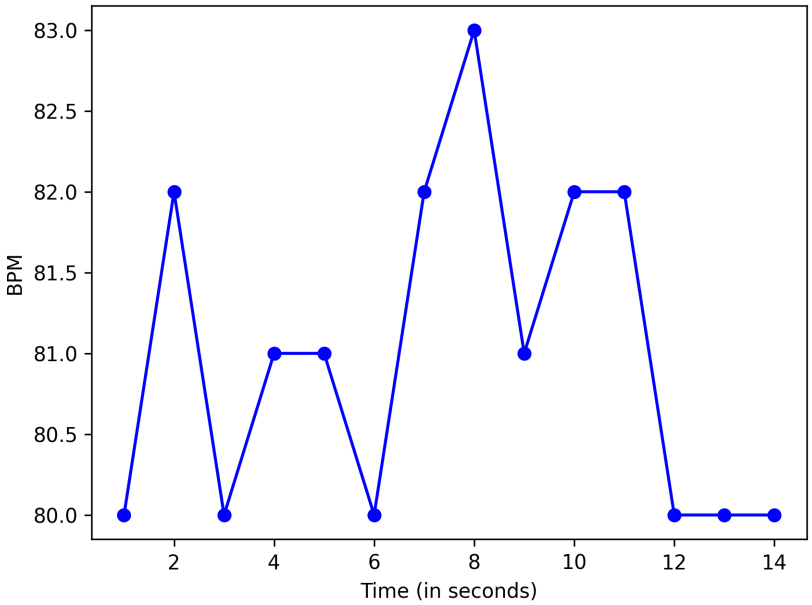

Figura 4.4: Andamento temporale del BPM di un musicista principiante con correzione

Come possiamo notare dai seguenti grafici la correzione viene eseguita in modo praticamente istantaneo ad inizio battuta, trascurando i tempi di reazione del musicista, questo poiché la latenza dei pacchetti correttivi se abbiamo un numero adeguato di SMI, come abbiamo visto precedentemente, è dell'ordine delle decine di ms e quindi trascurabile rispetto al tempo di battuta.
Dai grafici possiamo anche notare come, grazie alle correzioni del Server, il BPM tenda a stabilizzarsi intorno al BPM nominale, possono esserci tuttavia degli errori dovuti al fatto che il musicista principiante potrebbe riaccelerare, tuttavia è un errore trascurabile rispetto a quello che commetterebbe senza alcun tipo di correzione.

4.2.2 Correzione di un Musicista Professionista in una Jam Session

Vediamo qui di seguito il grafico che mostra l'andamento temporale di un SMI nella rete per Jam Session, ossia un Peer.

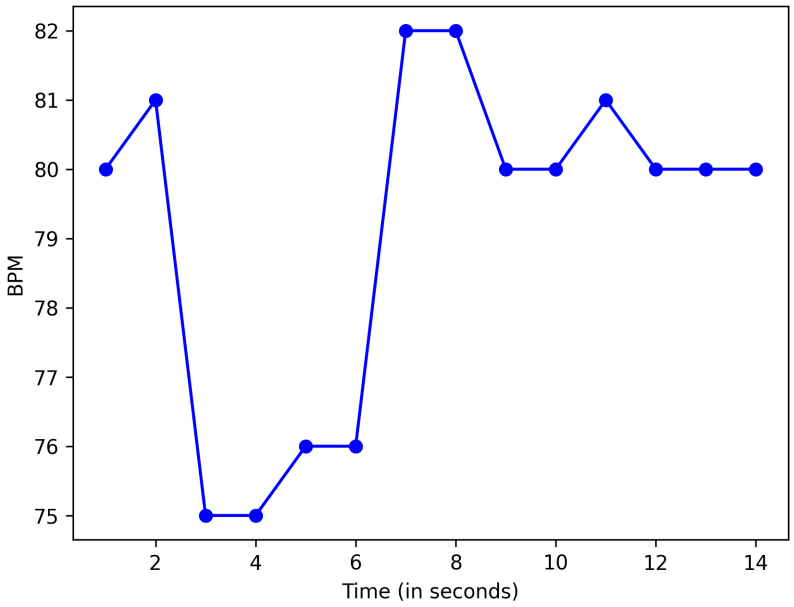

Figura 4.5: Andamento temporale del BPM di un musicista professionista in una Jam Session con correzione

Come si nota dal grafico, lo SMI Peer tenderà a inseguire quello di riferimento del musicista nominato come Sink, che nell'esempio in figura varia da 80-75-82-80 BPM, ovviamente con qualche incertezza data dal fatto che lo stesso Peer può commettere degli errori.
Tuttavia queste variazioni, rispetto al BPM che dovrebbe avere, saranno mol-

to piccole, ricordiamo infatti che l'errore che il musicista professionista può commettere è stato modellato con una varianza molto bassa.

Capitolo 5

Conclusioni

Alla luce dei risultati ottenuti nel capitolo precedente, possiamo andare ad analizzare la rete IoMusT progettata da due diverse prospettive: *Qualità del Servizio* (QoS) e *Qualità dell'Esperienza* (QoE).

5.1 Qualità del Servizio

Per quanto concerne la parte di Qualità del Servizio (QoS), come abbiamo visto il collo di bottiglia è dato dall'Access-Point wireless, sia nello scenario Client-Server (Fog Computing) sia in quello P2P (Edge Computing), in misura ancor più maggiore.
Il problema della latenza può essere tuttavia arginato pensando ad una possibile implementazione senza Access-Point, ad esempio utilizzando come standard di comunicazione tra gli SMI della rete locale il *Wireless Ad-Hoc*.
Un'altra problematica, se non ci trovassimo nel caso di rete cablata, è la scalabilità spaziale, che nel caso di reti mobili richiederebbe tempi di accesso e latenze molto basse, spesso non garantite anche dagli attuali standard come *LTE*.

Per arginare questo problema si potrebbe pensare all'utilizzo della tecnologia *5G* che riesce a garantire un tempo di accesso alla rete molto basso e latenze tipicamente di qualche decina di ms.

5.2 Qualità dell'Esperienza

Per quanto concerne invece la Qualità dell'Esperienza (QoE) è stata valutata andando a vedere, ad esempio, in che modo le correzioni effettuate andassero a beneficio degli SMI, come dai grafici del capitolo precedente.
Nel caso in cui si dovesse avere un BPM nominale di 80 ed uno spartito in 4/4, i pacchetti correttivi vengono inviati ogni 3 secondi, tempo di una singola battuta, e ci danno un'indicazione istantanea (latenza di ricezione della correzione come abbiamo visto è di pochi ms) su come è andata la battuta appena avvenuta e su eventuali comportamenti correttivi.
Anche aumentando di molto il BPM portandolo, ad esempio, ad un valore di 240 BPM, con conseguente invio dei pacchetti correttivi ogni secondo e non più ogni 3 secondi per rispettare il tempo di battuta, si ha che la latenza con cui arrivano le notifiche, una decina di ms, è molto inferiore, circa due ordini di grandezza, rispetto al tempo della battuta stessa, e quindi come nel caso ad 80 BPM possiamo approssimarlo istantaneo all'inizio della battuta.

Ringraziamenti

In prima istanza vorrei ringraziare il professor Francesco Chiti il quale grazie alla sua dedizione, alla sua disponibilità e alle numerose conoscenze che mi ha trasmesso ha reso possibile lo svolgimento di questo elaborato.
Ringrazio i miei genitori Fabio e Barbara, mia sorella Alessia ed in generale ogni singolo componente della mia famiglia senza la quale non sarei la persona che sono adesso.
Un ringraziamento speciale a mia nonna Gloria, per essermi sempre stata d'aiuto sopratutto durante il percorso universitario.
Infine ringrazio immensamente tutti i miei amici, sia esterni all'Università sia compagni di corso, i quali hanno reso questi anni sicuramente più leggeri e spensierati.

Bibliografia

[1] Francesco Chiti. *Internet Prospettive, Architetture e Applicazioni*. Società Editrice Esculapio, 2020.

[2] OMNeT. Inet user's guide, 2020.

[3] OMNeT. Omnet++ simulation manual, 2020.

[4] Luca Turchet, Carlo Fischione, Damian Keller, and Mathieu Barthet. Internet of musical things: Vision and challenges. *IEEE Access*, 2018.

[5] Luca Turchet, Fabio Viola, Gyorgy Fazekas, and Francesco Antoniazzi. C minor: a semantic publish/subscribe broker for the internet of musical things. *IEEE Conference Paper*, 2018.

[6] Luca Turchet, Fabio Viola, Gyorgy Fazekas, and Mathieu Barthet. Towards a semantic architecture for the internet of musical things. *IEEE Conference Paper*, 2018.