POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Studio di fattibilità e valutazione delle prestazioni di un sistema di comunicazione low-latency distribuito mediante OpenDDS

Relatori
prof. Enrico Masala
prof. Guido Marchetto

Candidato
Marco Toscano

Aprile 2018

Indice

Elenco delle figure

Sommario

La tesi si focalizza sullo studio di fattibilità e valutazione di un sistema software real-time utilizzando OpenDDS. Esso è un'implementazione di un Data Distribution Service for Real Time Systems (DDS), esso è uno standard emanato dall'Object Management Group (OMG) che utilizzando il paradigma publish/subscribe definisce un middleware per la distribuzione dei dati in tempo reale.

Pochi studi si concentrano esclusivamente sulla bassa latenza nella trasmissione dei dati audio con un framework open source. In questa tesi si propone un esempio funzionante di architettura distribuita con una determinata struttura di messaggio mediante gli standard utilizzati da OpenDDS. Inoltre, grazie all'utilizzo di numerosi standard di protocolli per sistemi distribuiti, OpenDDS fornisce una completa interoperabilità e utilizzo dello stesso codice per un largo numero di hardware e sistemi operativi differenti.

Il risultato dello studio è stato quello di mostrare la fattibilità nella creazione di un sistema di comunicazione low-latency utilizzando un' implementazione open source di DDS.

Struttura tesi Capitolo 1 espone una descrizione più approfondita degli obiettivi del progetto di tesi e i sistemi di comunicazione Multimedia.

Capitolo 2 introduce gli strumenti software utilizzati per la realizzazione del lavoro di tesi e lo standard principale utilizzato per il funzionamento del progetto che è DDS per i sistemi di distribuzione di dati in tempo reale secondo il paradigma publish/subscribe.

Capitolo 3 vengono introdotti alcuni protocolli sfruttati per i dati real-time che sono utilizzati generalmente nei sistemi distribuiti per la trasmissione multimediale.

Capitolo 4 viene descritta la fase finale di progettazione dei vari componenti per il funzionamento di OpenDDS. Spiegando l'utilizzo delle varie classi e interfacce generate dal compilatore.

Capitolo 5 sono le varie fasi di sviluppo del lavoro di tesi per il raggiungimento dell'obiettivo di tesi. Spiegando quali strumenti di OpenDDS si possono sfruttare per risolvere i problemi che si sono affrontati durante lo sviluppo del software.

Capitolo 6 in questo capitoli sono mostrati le stampe dei messaggi scambiati, le prestazioni e il consumo delle risorse del sistema.

Capitolo 7 avremo gli obiettivi raggiunti, i vantaggi, gli svantaggi del sistema utilizzato e i possibili sviluppi futuri.

Capitolo 1

Introduzione

Questo lavoro di tesi è stato realizzato presso i laboratori del Dipartimento di Automatica e Informatica (DAUIN) nel Politecnico di Torino, sotto la visione del Prof. E. Masala e Prof. G. Marchetto. L'obiettivo principale del lavoro di tesi si è focalizzato sullo studio di valutazione di un sistema distribuito per una comunicazione real-time usando OpenDDS, un'implementazione open source dello standard Data Distribution Service (DDS).

Negli ultimi anni nel campo delle trasmissioni multimediali ha preso sempre più diffusione l'importanza di sfruttare sistemi distribuiti eterogenei che hanno interoperabilità. Affinché siano costituiti di queste caratteristiche si pone la necessità di creare uno strato di compatibilità dei sistemi. Attualmente con le tecnologie middleware come per esempio CORBA sono stati fortemente utilizzati in molte aree applicative per mascherare i problemi scaturiti in sistemi eterogenei e ridurre l'intrinseca complessità delle applicazioni distribuite. Negli ultimi anni è comparsa l'applicazione dei middleware in diverse aree di applicazione come nei sistemi embedded, real-time e multimedia. Queste nuove applicazioni ha portato nuove sfide che attualmente le piattaforme middleware non possono affrontare, perché richiedono una maggiore richiesta di condivisione delle risorse e più reattività. Quindi richiedono caratteristiche aggiuntive al middleware sottostante. DDS porta lo sviluppo per sistemi di architettura modulari e aperti, aiutando la creazione di interfacce tra componenti e sottosistemi ben definite. Questo porta a ridurre la complessità di sviluppo e i costi di integrazione e aggiornamento. OpenDDS è un'implementazione open source in C++ della specifica OMG DDS. OpenDDS possiede un meccanismo di configurazione basato su file. Attraverso la configurazione del file comune ai partecipanti, un utente può configurare il trasporto del Publisher o Subscriber, l'output del debug, la

posizione nella rete del Dispatcher e molte altre impostazioni.

Il capitolo successivo espone in dettaglio gli strumenti software utilizzati per la realizzazione del lavoro di tesi.

Capitolo 2

Background tecnologico

2.1 Introduzione

In questo capitolo verranno introdotte i vari strumenti software utilizzati per la realizzazione dello studio di fattibilità e valutazione del sistema distribuito middleware OpenDDS. In un sistema distribuito, il middleware è un software che si trova tra il livello di trasporto e quello applicativo del modello ISO/OSI. Per lo svolgimento del lavoro di tesi è stato sfruttato il sistema di distribuzione di dati in tempo reale DDS secondo il paradigma publish/subscribe.

2.2 DDS

Il Data Distribution Service for Real Time Systems (DDS) è una specifica di uno standard middleware, che consente di comunicare tra i nodi distribuiti tra loro usando un paradigma di publish/subscribe, e uno standard Application Interface (API) per la connettività di un sistema incentrato sui dati definito da Object Management Group (OMG). Esso integra i componenti in un sistema, fornendo connettività di dati a bassa latenza, avendo una forte affidabilità e un'architettura scalabile che può essere sfruttata per applicazioni aziendali e mission-critical per Internet of Things (IoT). DDS adotta un paradigma publish/subscribe e un approccio data-centric definito in[1]. Le specifiche DDS definiscono lo standard in due livelli di interfacce e funzionalità separate:

- Data-centric Publish-Subscribe (DCPS): è lo strato inferiore che ha il compito di consegnare in maniera efficiente e corretta le informazioni ai destinatari correttamente. Nel senso che dà la capacità ai publisher di riconoscere i data-object che vogliono pubblicare e quindi fornire i loro valori. Inoltre, permette ai subscriber di riconoscere i data-object interessati e quindi di accederci, per questo consente un efficiente trasmissione dei dati dal publisher al subscriber.

- Data Local Reconstruction Layer (DLRL): strato superiore, ipoteticamente è un livello facoltativo, è disposto sopra il DCPS. Il suo utilizzo è quello di fornire una visione più diretta ai dati scambiati, dando la possibilità ad uno sviluppo software con maggiore integrazione a livello applicativo [2].

L'utilizzo di DDS porta dei vantaggi per lo sviluppatore rispetto agli altri middleware che sono qui elencati.

2.2.1 Data-centricity

DDS ha un funzionamento fortemente data-centric. Nei middleware tradizionali incentrati sui messaggi, i programmatori devono scrivere il codice che invia i messaggi. Invece utilizzando la programmazione di un middleware data-centric dovremo specificare come e quando condividere i dati e quindi direttamente i valori dei dati. Tuttavia, in un sistema incentrato sui dati, gli scambi di informazione si riferiscono ai valori scambiati in un data-object globale. Dato che le nuove informazioni sovrascrivono in genere i valori precedenti, sia l'applicazione che il middleware devono identificare univocamente l'istanza nell' oggetto globale che viene chiamato Global Data Space (GDS) a cui si applica il valore. Il GDS è diviso in domini, i quali sono identificati da un numero univoco. Un publisher che scrive il valore in un data-object deve identificare in un modo univoco il dato che sta scrivendo. Cosicché il middleware può distinguere l'istanza che viene scritta e decidere, ad esempio se mantenere solo il valore più recente[3].

2.2.2 Topic-based

Il modello dei dati di DDS publish/subscribe (pub/sub) è topic-based: i messaggi sono pubblicati su Topic e ogni di essa è identificato univocamente. Per avere una grande interoperabilità, i tipi di dati di DDS sono un sottoinsieme dello standard OMG Interface

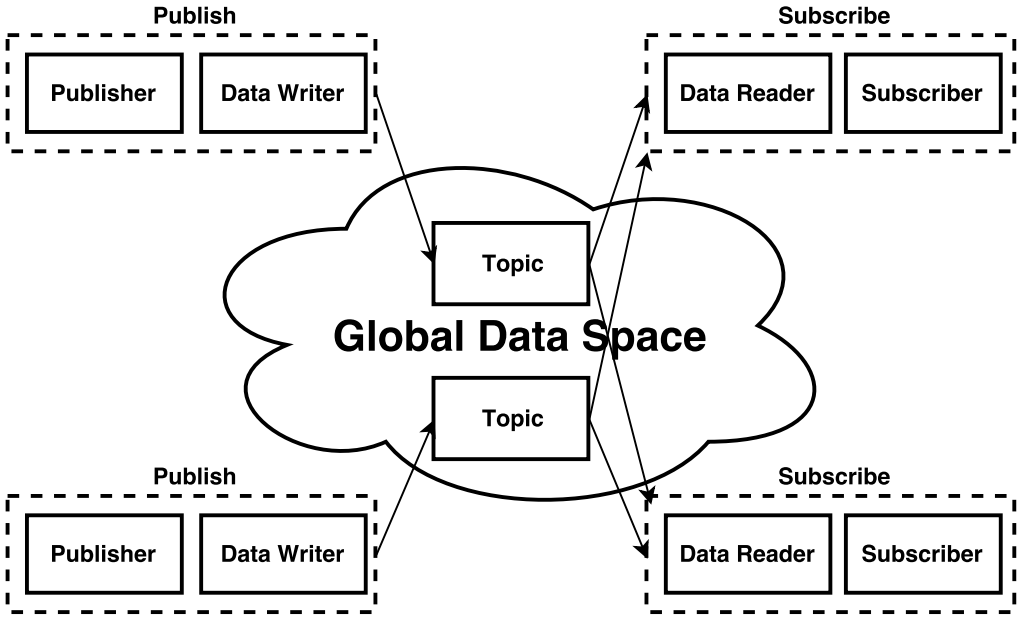

Figura 2.1: Global Data Space.

Definition Language (IDL). Ogni Topic è definito in un determinato dominio specifico che è una comunicazione logica e separata dalle diverse applicazioni distribuite e tutte indipendenti in esecuzione sullo stesso gruppo di computer. Ogni dominio DDS può essere diviso in diverse partizioni per dare la possibilità di dividere il partizionamento del traffico. In ogni dominio possono coesistere diversi Topic. Due Topic con lo stesso nome possono esserci in DDS ma soltanto definendoli in diversi domini.

2.2.3 Architettura scalabile

Lo standard DDS è progettato per essere molto scalabile per dispositivi piccoli come in Internet of Think (IoT) e sistemi molto grandi. Esso consente di scalare migliaia di partecipanti utilizzando IoT, trasmettendo dati a bassa latenza e gestendo molti oggetti di dati, fornendo disponibilità e sicurezza. DDS semplifica gran parte della complessità nello sviluppo di sistemi distribuiti.

2.3 Design pattern publish/subscribe

Publish-subscribe è un pattern di messaggistica dove il mittente dei messaggi, viene chiamato publisher, non invia direttamente i dati ai destinatari specifici, chiamati subscriber, se sempre sono presenti. Allo stesso tempo, i subscriber esprimono interesse ad uno o più tipi di messaggi e ricevono dati solo di loro interesse, senza avere la conoscenza di quale publisher ci sono, se ce ne sono attivi.

2.4 Data-Centric Publish-Subscribe

Le specifiche DDS descrivono un modello Data-centric Publish-Subscribe (DCPS) per la comunicazione di applicazioni distribuite e integrate. Questa specifica definisce sia le API che la semantica della comunicazione, cioè comportamento e Quality of Service (QoS) che consentono l'efficiente trasmissione di informazione dai produttori di informazioni ai consumatori corrispondenti[4].

2.4.1 DomainParticipantFactory

È un singleton, uno dei pattern fondamentali descritti dalla Gang of Four (GoF), per garantire una classe che sia creata una sola istanza [5]. Viene utilizzato per accedere al GDS, cioè lo spazio di comunicazione del DDS. Esso per poter comunicare in un dominio avrà bisogno del punto di accesso che si chiama DomainParticipant, il quale può essere creato soltanto da questo DomainParticipantFactory.

2.4.2 DomainParticipant

DomainParticipant è il punto di accesso per ogni dominio e dopo essere stato creato dalla DomainParticipantFactory a sua volta funge da factory, uno dei pattern fondamentali descritti dalla GoF [5], per la creazione e la distruzione dei Topic, Publisher e Subscriber dello stesso dominio DDS. Ogni DomainParticipant ha il proprio gruppo di thread e strutture dati interne che gestiscono le informazioni sulle Entità create da sé e altri DomainParticipant nello stesso dominio DDS.

2.4.3 Topic

Topic fornisce il punto di connessione base tra publisher e subscriber [6]. Per avvenire il collegamento, il Topic di un publisher deve corrispondere al Topic associato ad ogni subscriber. Se i Topic non corrispondono, non avrà luogo la comunicazione. I Topic sono definiti in un linguaggio standard da OMG che definisce la struttura dati delle interfacce, questo linguaggio si chiama Interface Definition Language (IDL). Un Topic è definito all'inizio della struttura da un Topic Name e un Topic Type. Il Topic Name è una stringa che identifica in modo univoco il Topic nel proprio dominio, mentre il Topic Type è la definizione del tipo di dati contenuti nel Topic. All'interno della struttura di un Topic possiamo aver scelto una chiave. Questa chiave verrà utilizzata dal sistema distribuito per ordinare i dati in arrivo. Il contenitore del valore di una chiave dati è definito come un'istanza.

2.4.4 Publisher

Publisher ha il compito di distribuire i dati provenienti dai DataWriter, che possono essere uno o molti, ad esso associati. Determina quando i dati devono essere inviati ai subscriber. A seconda delle impostazioni dei publisher e dataWriter i dati possono essere memorizzati in un buffer per essere mandati con i dati di altri dataWriter o non inviati completamente. Esso funge da factory per la creazione e distruzione dei DataWriter.

2.4.5 DataWriter

È l'entità che viene usata dall'applicazione per comunicare con un publisher, per l'esistenza dei valori di oggetti con un tipo determinato. Durante la creazione è vincolato ad un solo Topic, specificando prima della creazione il nome del Topic e il tipo di dato associato ai dati. Dopo che il publisher ha ricevuto i valori dei data-object dell'appropriato DataWriter, il publisher ha il compito di eseguire la distribuzione.

2.4.6 Subscriber

Subscriber ha il compito di ricevere i dati del publisher e li mette a disposizione [3]. Esso può ricevere e inviare dati di vario tipo. Per avere i dati ricevuti deve usare un determinato DataReader per ogni tipo di topic sottoscritto.

2.4.7 DataReader

È l'entità che viene usata dall'applicazione per accedere ai dati che sono stati ricevuti dal subscriber. Notifica il sistema distribuito che il dato dal GDS sono disponibili. Per leggere i dati ricevuti, l'applicazione deve utilizzare un tipo di dataReader collegato al subscriber.

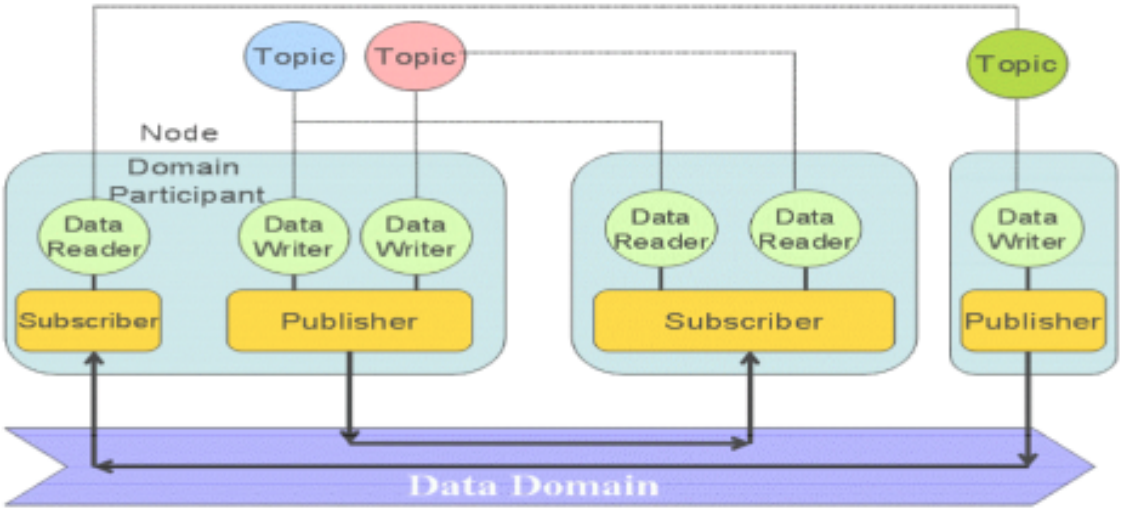

Figura 2.2: Entità di DDS. [7]

2.5 OpenDDS

OpenDDS è un'implementazione open source del protocollo middleware OMG DDS, sponsorizzato dalla Object Computing In. [8]. OpenDDS grazie al fatto che è disponibile sotto un modello di software open source, il codice sorgente è stato progettato per essere compilato ad un largo numero di hardware e sistemi operativi differenti. Esso utilizza anche altri prodotti open source come Make Project Creator (MPC), the ADAPTIVE Communication Enviroment (ACE) e The ACE ORB (TAO). MPC serve per generare file di un progetto software per vari ambienti di sviluppo. MPC è stato implementato in script Perl e quindi indipendente dalla piattaforma.

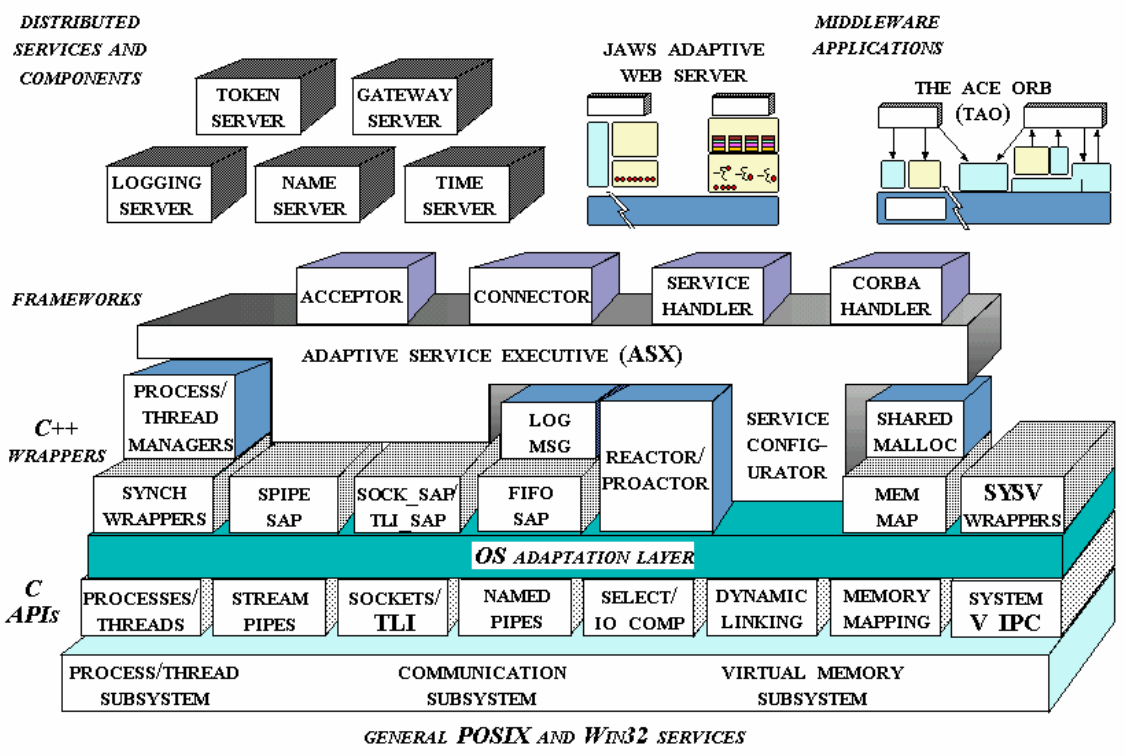

Figura 2.3: Struttura e funzionalità di ACE. [9]

2.5.1 ACE e TAO

ACE fornisce un gruppo di classi C++ per semplificare la programmazione di rete utilizzando lo stesso codice in più sistemi operativi differenti. È orientato a oggetti e implementa molti dei pattern per la trasmissione concorrente. Questo porta ad incrementare la portabilità, efficienza e una maggiore facilità di transizione al middleware standard di livello superiore. Infatti, ACE fornisce pattern e componenti riutilizzabili per TAO.

TAO è un'implementazione C++ per sistemi in tempo reale e conforme agli standard in CORBA basata su ACE. Utile per uno sviluppo di applicazioni distribuite che hanno elevate richieste di prestazioni.

2.6 CORBA

Lo standard CORBA definito dall'OMG consente l'interoperabilità delle applicazioni distribuite scritte in vari linguaggi di programmazione, eseguite in diversi sistemi operativi e diverse piattaforme hardware [10]. OpenDDS genera le classi definite nel codice Interface Definition Language (IDL) dalle interfacce CORBA. Un compilatore IDL esterno traduce il codice IDL in un linguaggio ad alto livello, come ad esempio in questo lavoro di tesi in C++.

2.6.1 GIOP, IIOP e ORB

General Inter-ORB Protocol (GIOP) è un'architettura di interoperabilità ORB generale introdotto da CORBA 2.0. GIOP è un protocollo in cui definisce la specifica sintassi del trasferimento e i messaggi per consentire la comunicazione di altri ORB sviluppati in modo indipendente. Internet Inter-ORB Protocol (IIOP) specifica come in GIOP è implementato nel TCP/IP [11]. GIOP definisce le strutture e i formati dei messaggi scambiati tra dispositivi diversi con architetture eterogenee. GIOP ha due differenti versioni: 1.0 e 1.1, quindi i messaggi possono essere di formato differente in base alla versione che si utilizza.

Object Request Broker (ORB) è una componente di un middleware che consente di effettuare le chiamate di programmi da differenti dispositivi a un altro tramite una rete, garantendo la trasparenza delle chiamate di procedura remota.

Nella Figura 2.4 mostra il modello OSI confrontato alla pila protocollare usato dall'OMG.

2.6.2 IDL

L'OMG Interface Definition Language (OMG IDL) è il linguaggio che si usa per definire le interfacce del client e forniscono le implementazioni degli oggetti chiamati. Perciò CORBA definisce una mappatura da IDL a un linguaggio ad oggetti come C++ o Java. Ci sono anche mappature standard per molti altri linguaggi. Per definire un IDL bisogna mettere almeno una variabile in chiave primaria e per identificare una variabile deve sarà case sensitive e tutte le definizioni devono terminare con un punto e virgola, come avviene in Java e in C/C++. Possono essere definiti uno dentro l'altro mediante moduli o interfacce.

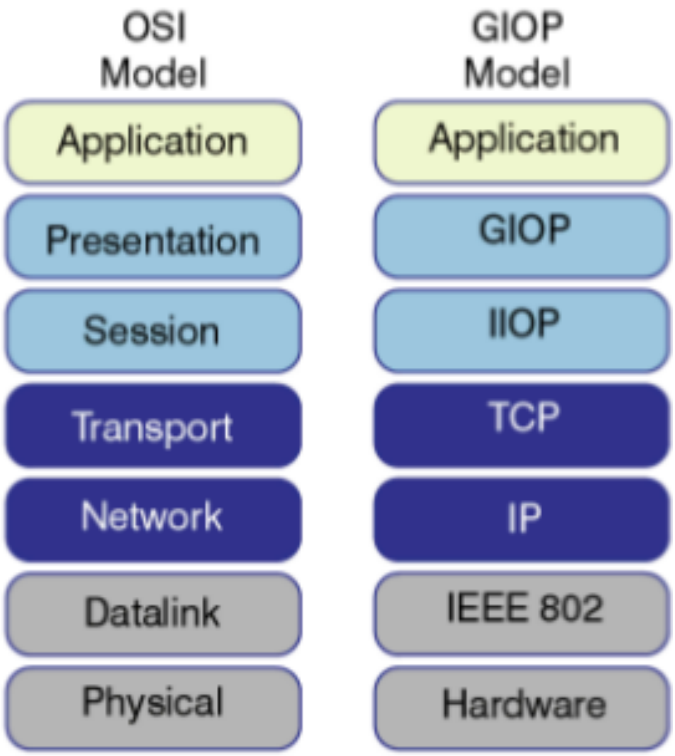

Figura 2.4: Pila protocollare tra OSI e GIOP. [12]

2.6.3 Tipi utilizzabili nei Topic

Un Topic lo definiamo con il linguaggio OMG IDL ed ha tre categorie generali di tipo di dato:

- IDL basic types: sono i tipi semplici e hanno lo stesso significato come in buona parte dei linguaggi di programmazione: short, unsigned short, long, unsigned long, float, double, char, boolean, char, octet ed any. Fanno eccezione gli ultimi due basic type che rappresentano rispettivamente octet un valore da 8 bit ed any per esprimere un qualsiasi IDL type arbitrario.

- IDL complex types: sono gli unici tipi complessi utilizzabili in DDS e sono: enum, struct, union, string, sequence e array. Anche questi hanno lo stesso comportamento in un linguaggio come C ad eccenzione fatta della sequence che dichiara una sequenza di qualsiasi IDL type. È simile ad un array unidimensionale eccetto che non ha una dimensione fissa, molto utile in questo progetto di tesi per poter trasferire una sequenza di byte.

Capitolo 3

Protocolli per la comunicazione di dati real-time

In questo capitolo vengono introdotti alcuni protocolli per i dati multimediali sfruttati per la trasmissione real-time che sono utilizzati generalmente nei sistemi distribuiti.

Questi protocolli si posizionano nel livello applicativo nella pila protocollare TCP/IP e in particolare sono utilizzati anche per la gestione della sessione che esso è una delle parti più importanti nell'architettura di una trasmissione multimediale. Per lo streaming audio abbiamo bisogno di vari protocolli per avere l'inizializzazione e la trasmissione dei dati multimediali, quelli più utilizzati sono:

- RTP (Real Time Transport Protocol) è un protocollo di trasporto in tempo reale dei dati.

- RTCP (Real Time Control Protocol) è un protocollo ideato per lavorare con RTP per il monitoraggio e non di controllo nella qualità della distribuzione.

- RTSP (Real Time Streaming Protocol) è un protocollo di controllo ma non di monitoraggio per la trasmissione dei dati.

- SIP (Session Description Protocol) è un protocollo per la creazione e gestione delle trasmissioni a basso ritardo.

3.1 Utilizzo del TCP o UDP per il real-time

Per la trasmissione di dati real-time può portare diversi problemi se si vuole utilizzare protocolli più comuni come il protocollo TCP o UDP.

Il TCP crea collegamenti bidirezionali per il trasporto di dati, garantisce la consegna dei pacchetti senza perdere informazioni, controlla le congestioni, il flusso dati e adatta automaticamente la comunicazione di rete e lo stato del ricevente. Però i dati ricevuti attraverso il protocollo TCP può subire latenze che non possono essere gestite direttamente da un sistema distribuito e non è per niente adatto a comunicazioni multimediali low-latency se sono presenti congestioni. Invece nel protocollo UDP avremo poche funzionalità in aggiunta dell'IP, ogni pacchetto trasmesso è indipendente dalle altre, fornisce il controllo di integrità per il carico utile. Però lascia molti aspetti all'applicazione, è più adatto per le comunicazioni multimediali però in condizioni difficili come per esempio le congestioni di rete.

La soluzione che si può seguire è quella di utilizzare protocolli progettati per la comunicazione che ha la proprietà della bassa latenza come ad esempio RTP. Insieme a questa soluzione si potrebbe modificare l'architettura di rete con router o un dispatcher come nel caso di questo lavoro di tesi oppure inserire una forma di Quality of Service (QoS).

3.2 RTP

Il protocollo RTP fornisce i servizi di consegna point-to-point per i dati in tempo reale, per esempio, come nell'audio e nel video interattivi [13]. Definito nell'RFC appena citato, affronta molti problemi che non vengono affrontati nell'UDP. Ogni sequenza di bit ha determinati riferimenti temporali. Rileva problemi di comunicazione come i pacchetti mancanti o ridondanti. Monitora il traffico con statistiche, sono previsti l'utilizzo di differenti formati multimediali. Però si deve gestire autonomamente problemi come l'adeguazione della larghezza di banda, la velocità di trasmissione, quindi dovrà essere gestito da un altro protocollo. In conclusione, è stato scelto l'UDP come imbustamento per il protocollo RTP per varie ragioni: RTP è stato ideato per il multicast e non è appropriato l'uso del TCP che è connesso e non scala bene come l'UDP, il secondo fatto è che la trasmissione real-time, l'affidabilità non è di principale importanza rispetto la

trasmissione veloce perché se ci fosse qualche perdita di pacchetto è necessario continuare con il flusso di trasmissione.

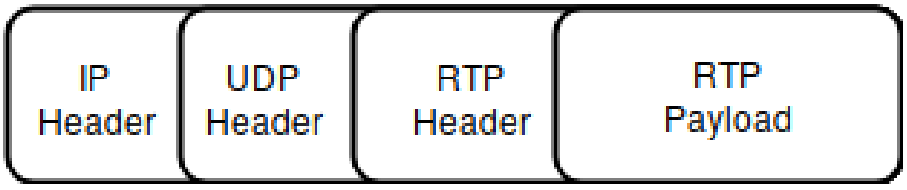

Figura 3.1: RTP pacchetto.

In Figura 3.1 abbiamo i vari imbustamenti che vengono fatti per la trasmissione di un pacchetto RTP. Questo protocollo aggiunge all'UDP: il timestamp, il numero di sequenza, l'identificazione del payload e la possibilità di identificare le sorgenti e per la sincronizzazione.

3.3 RTCP

RTCP fornisce informazioni all'applicazione per la riproduzione multimediale di diverse sessioni RTP. Ogni sessione RTP avremo sempre per il monitoraggio una sessione RTCP che non trasporta dati multimediali, ma solo informazioni di monitoraggio e statistiche. I pacchetti RTCP vengono periodicamente inviati da ciascun partecipante a tutti gli altri partecipanti per comunicare l'informazione sulla qualità della trasmissione dei dati.

In questi pacchetti d'informazione, fornisce varie servizi. Il più importante è il monitoraggio della congestione e la qualità del servizio, così che l'applicazione potrebbe valutare le performance della rete per la trasmissione multicast. Inoltre, permette una informazione di ritorno sulla sessione RTP, utile per avere un feedback all'applicazione per adattarsi alle condizioni della rete come cambiare la codifica o ridurre il tasso di trasmissione dei dai.

3.4 RTSP

RTSP è un protocollo che si trova nel livello applicativo che durante la trasmissione dei dati controlla il flusso trasmesso. È strutturato con un paradigma Client-Server che ha il compito di controllare la trasmissione dei dati solo quando è necessario. Fornisce un

insieme di comandi dedicati al controllo ed alla trasmissione distribuita delle risorse nella rete.

RTSP ha il dovere di trasportare le informazioni e non di distribuire i dati stessi, fatto per esempio dal protocollo RTP. Principalmente supporta il controllo dei dati sul trasporto e il recupero delle informazioni da un server multimediale.

3.5 SIP

Il protocollo SIP è un servizio di livello applicazione, può essere eseguito sopra TCP, UDP o SCTP (intermedio tra TCP e UDP) ed è simile a HTTP. Il suo utilizzo è quello di creare, modificare e terminare sessioni tra uno o più partecipanti [14].

SIP ha un'architettura Client-Server, è facilmente programmabile, indipendente dal protocollo di trasporto e i suoi componenti principali sono User Agent, Proxy Server, Redirect server e Register.

User Agent è il client software installato in un computer che viene usato dall'utente direttamente come mostrato in Figura 3.2 .

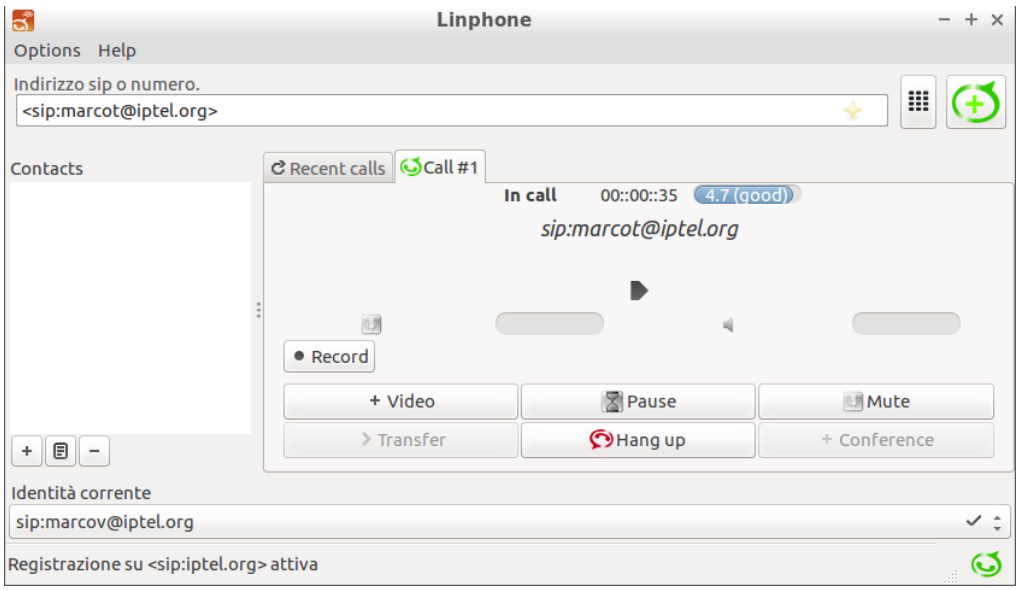

Figura 3.2: Esempio di User Agent di SIP.

Proxy Server ha il compito di inoltrare le richieste dell'User Agent al SIP server più vicino o un altro User Agent della stessa rete interna.

Register è un server dedicato oppure si trova in un proxy che gestisce le richieste di registrazione e memorizza le informazioni dell'utente.

Redirect server riduce la mole di lavoro al Proxy Server perché risponde alle richieste del Client e lo aggiorna sull' indirizzo del server richiesto successivo.

Capitolo 4

Progettazione

In questo capitolo verrà descritta la fase finale di progettazione dei vari componenti per il funzionamento di OpenDDS.

Inizialmente il lavoro di tesi è iniziato utilizzando l'esempio StockQuoter rilasciato dagli sviluppatori, ma dopo varie fasi di lavoro è stato fortemente cambiato lasciando la struttura delle classi fondamentali per il funzionamento della comunicazione e le classi astratte generate durante la compilazione. Sono stati usati vari oggetti astratti per il funzionamento di OpenDDS, per la creazione di più trasmissioni tra diversi publisher e subscriber, inoltre sono state utilizzate le interfacce di C++ generate dal compilatore per l'utilizzo dei tipi dichiarati in due Topic differenti, definiti per lo scambio dei messaggi e il funzionamento corretto di OpenDDS. È stato sfruttato questo esempio che era stato scritto per funzionare tutto in uno stesso computer, questo non affrontava le problematiche dovute al funzionamento completo in un sistema distribuito e la valutazione delle prestazioni di trasmissione, infine nell'esempio non dava una comprensione chiara di come il sistema distribuito riuscisse a creare la comunicazione tra i vari componenti di OpenDDS senza trovarsi in localhost della stessa macchina, secondo non meno banale si è dovuto apportare delle modifiche in varie classi per eseguire trasferimento di dati audio in thread concorrenti tra più publisher e subscriber.

Infine, i Topic per lo scambio dei messaggi nella soluzione finale è stato completamente riscritto rispetto agli esempi svolti trovati nei repository di OpenDDS per valutare lo scambio di un flusso dati audio a bassa latenza. Inoltre, si è dovuto affrontare il problema che gran parte dei tipi complessi sono stati deprecati nelle ultime versioni e quindi una parte degli esempi non potevano essere provati.

4.1 Strumenti utilizzati

L'ambiente utilizzato per analizzare il sistema middleware è realizzato utilizzando varie macchine virtuali create tramite Oracle VM VirtualBox con sistema operativo Lubuntu 16.04.1 a 64 bit. I test sono stati fatti su macchine virtuali differenti che comunicavano con una scheda virtuale configurata ad hoc in ogni VM per la trasmissione isolata in LAN per lo scambio dei messaggi tra le varie VM, invece è stata utilizzata una seconda scheda di rete virtuale collegata ad internet esclusivamente per apportare gli aggiornamenti del Sistema Operativo e il software delle VM. Si è utilizzato la versione 3.9 di OpenDDS, compilata su GNU/Linux direttamente in ogni VM.

Infine, si è dovuto intercettare e analizzare l'intero traffico tra le VM utilizzando: TCP-Dump per la cattura del traffico e successivamente analizzati i pacchetti con Wireshark per lo studio della traccia dei pacchetti.

4.2 Requisiti dell'applicazione

L'inizializzazione delle connessioni tra i componenti del sistema avviene innanzitutto con l'avvio del dispatcher che fa da mediatore tra il publisher e subscriber. Il dispatcher con opportuna configurazione decide e informa che tipo di protocollo a livello di trasporto utilizzare, quale versione di CORBA comunicare, quali e quanti domini creare. Successivamente è possibile eseguire i publisher, prima comunicano con il dispatcher e dopo pubblicano il dato. Infine, i subscriber che prima comunicano con il dispatcher e dopo si sottoscrivano ad un dominio per ricevere i Topic. È importante sottolineare che il dispatcher dopo aver instaurato la connessione tra i dispositivi non riceve le informazioni che vengono pubblicate o quelle che ricevono i subscriber. Questo porta dopo vari scambi di messaggi la creazione delle connessioni e una comunicazione a bassa latenza di trasmissione tra il publisher e subscriber.

Analizzando le classi di OpenDDS mostra un forte utilizzo del linguaggio C++11 che ha in sé gran parte dei paradigmi di concorrenza e per la gestione multi-thread che porta uno scambio dei messaggi a bassa latenza anche più publisher e subscriber, tutto questo sotto i metodi per il funzionamento del middleware.

4.3 Architettura dell'applicazione

Per sviluppare e testare il funzionamento del middleware si è sfruttato Oracle VM VirtualBox per utilizzare cinque macchine virtuali (Virtual Machine VM) nello stesso computer. Sono state usate due schede di rete virtuali per ogni VM: la prima serviva per creare una rete isolata tra le VM ed avere delle tracce pulite dei pacchetti scambiati, invece la seconda scheda di rete è stata usata per collegarsi ad Internet per l'installazione e l'aggiornamento del Sistema Operativo e di OpenDDS.

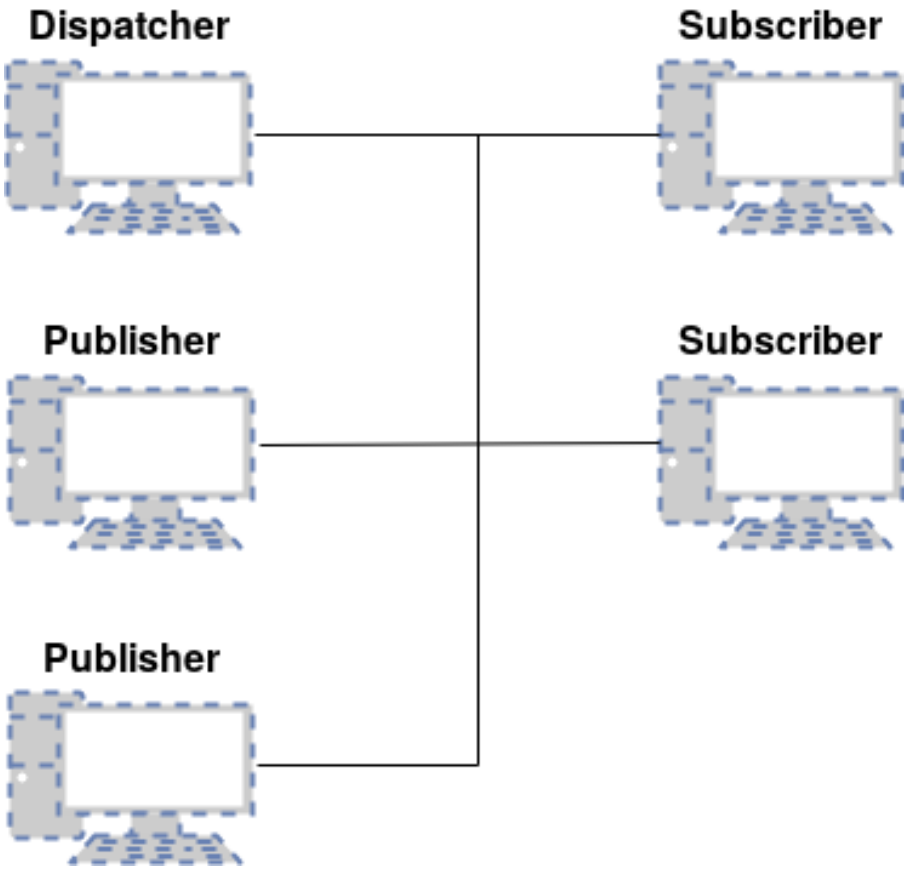

Figura 4.1: Architettura di rete.

Nella Figura 4.1 abbiamo due VM come publisher per simulare più messaggi differenti pubblicati, due VM come subscriber per mostrare che potessero ricevere Topic differenti e una VM come dispatcher per poter creare e distruggere le connessioni tra i vari publisher/subscriber, componente importante per poter pubblicare e sottoscrivere più Topic da parte di più publisher e subscriber.

Si evidenzia il fatto che nella trasmissione dei pacchetti tra i partecipanti devono avere hostname differenti e appartenere allo stesso nome di dominio, questo perché è stato appurato che le comunicazioni non vengono create utilizzando solo gli indirizzi IP ma anche con l'hostname correttamente configurato. Questo è per avere la massima astrazione durante lo sviluppo dei vari componenti, ma dall'altra parte porta una grossa complicazione se c'è l'esigenza di capire quale indirizzi IP vengono utilizzati per instaurare le comunicazioni delle varie VM.

4.4 Descrizione delle Classi del progetto

Il progetto creato di OpenDDS è strutturato da varie classi fondamentali per il suo funzionamento, alcune generate dal compilatore IDL per il Topic e altre sono interfacce e classi create durante la compilazione dell'intero progetto.

- AudioSystemC.cpp
- AudioSystemC.h
- AudioSystemC.inl
- AudioSystemCommon_Export.h
- AudioSystem.idl
- AudioSystem.mpc
- AudioSystem.mwc
- AudioSystemS.cpp
- AudioSystemS.h
- AudioSystemTypeSupportC.cpp
- AudioSystemTypeSupportC.h
- AudioSystemTypeSupportC.inl
- AudioSystemTypeSupport.idl

- AudioSystemTypeSupportImpl.cpp
- AudioSystemTypeSupportImpl.h
- AudioSystemTypeSupportS.cpp
- AudioSystemTypeSupportS.h
- dds_udp_conf.ini
- ExchangeEventDataReaderListenerImpl.cpp
- ExchangeEventDataReaderListenerImpl.h
- libAudioSystemCommon.so.3.9.0
- publisher.cpp
- RadioDataReaderListenerImpl.cpp
- RadioDataReaderListenerImpl.h
- subscriber.cpp

Quando si crea il progetto la prima cosa da fare è compilare con il comando opendds_idl il file AudioSystem.idl, in cui sono definiti i Topic ed esso genera AudioSystemTypeSupportImpl.cpp ed AudioSystemTypeSupportImpl.h in cui sono definiti i metodi per utilizzare tutte le variabili definite dentro ad AudioSystem.idl. Dopo la creazione di questi file vengono generati i file che iniziano con il nome di AudioSystemTypeSupport in cui rispettivamente sono: quelli che finiscono con C sono per lo stub, invece i file che finisco con S sono per lo skeleton, entrambi paradigmi di ingegneria del software utilizzati soprattutto nei sistemi distribuiti. AudioSystemCommon_Export.h è un interfaccia generata per le DLL di Windows dato che la compilazione è multipiattaforma, invece libAudioSystemCommon sono le librerie dinamiche condivise per il Sistema Operativo GNU/Linux. Successivamente abbiamo AudioSystem.mpc e AudioSystem.mwc fondamentale per il software MPC, strumento open source per la compilazione del codice del progetto, sviluppato da ACE e TAO per un utilizzo indipendente dalla piattaforma, infatti supportato dal compilatore GCC che di Microsoft Visual Studio. Dopo questi abbiamo

le classi e le interfacce ExchangeEventDataReaderListenerImpl e RadioDataReaderListenerImpl che implementano i metodi generati per la gestione dei dati da inserire nei messaggi scambiati e le stampe dei messaggi spediti e ricevuti dai publisher e subscriber, inoltre, si trovano il Mutex e una lock_guard per gestire la concorrenza.

Infine, abbiamo publisher.cpp e subscriber.cpp in cui rispettivamente eseguono il publisher e il subscriber, in queste due classi ci sono le operazioni per instaurare la connessione tra il dispatcher e il publisher \subscriber definiti in OpenDDS.

4.5 Definizione del tipo di dato Radio ed ExchangeEvent

Il primo passo per l'implementazione di un middleware con standard OMG DDS è quello di definire ogni tipo di dato utilizzando IDL. Si vede in Figura 4.2 l'utilizzo della direttiva #pragma per permettere a OpenDDS di identificare i tipi di dato che DDS distribuisce nel sistema e gestisce. I tipi di dato nel Topic sono processati dai compilatori IDL di TAO e di OpenDDS, con il risultato di generare il codice necessario di inviare e ricevere i tipi semplici e complessi definiti in DDS. In questo caso abbiamo un file AudioSystem.idl che definisce due Topic tramite IDL.

4.5.1 AudioSystem.idl

In AudioSystem.idl abbiamo i due Topic, esso è effettivamente la definizione delle strutture dei messaggi scambiati. Viene definito in un file e compilato prima delle altre classi dal fatto che i metodi generati saranno fondamentali per richiamare la struttura del topic ed inserire nel codice tutti i dati che si vogliono pubblicare o sottoscrivere. Queste funzioni generate dalla compilazione sono l'unico modo per poter inserire i valori nei messaggi scambiati. Nella trasmissione di un flusso di dati audio a bassa latenza ha portato la scelta implementativa di questa struttura:

- Una Struct di nome Radio, in cui abbiamo la String id per identificare univocamente ogni messaggio scambiato, la String full_name per identificare quale publisher sta pubblicando e una sequence<octet> in cui rappresenta la sequenza di byte in cui inseriamo il dato nello streaming audio.
- Una Struct ExchangeEvent, in cui decidiamo quando aprire, chiudere, sospendere o riesumare la connessione tra i dispositivi usando una enum.

Apri | AudioSystem.idl | Salva

```
1  #include "orbsvcs/TimeBase.idl"
2
3  module AudioSystem
4  {
5  #pragma DCPS_DATA_TYPE "AudioSystem::Radio"
6  #pragma DCPS_DATA_KEY "AudioSystem::Radio id"
7
8  typedef sequence<octet> OctetSeq;
9
10 struct Radio {
11         string id;
12         string full_name;
13         OctetSeq packet; };
14
15 interface InterfaceSeq {
16         OctetSeq getPacket(inout OctetSeq byte);
17 };
18
19 #pragma DCPS_DATA_TYPE "AudioSystem::ExchangeEvent"
20 #pragma DCPS_DATA_KEY "AudioSystem::ExchangeEvent exchange"
21
22 enum ExchangeEventType { OPENED,
23                          CLOSED,
24                          SUSPENDED,
25                          RESUMED };
26
27 struct ExchangeEvent {
28         string exchange;
29         ExchangeEventType event;
30         short nome_publisher; };
31 };
```

Figura 4.2: AudioSystem.idl

Tale scelta della struttura di questi due Topic è dovuta da vari fattori. Innanzitutto ogni Topic definito deve contenere una variabile di valore univoco per identificare ogni singolo messaggio trasmesso, infatti ci sono due chiavi primarie una per ciascun Topic: Id ed Exchange, questi sono fondamentali per il funzionamento di OpenDDS per motivi di definizione dello standard OMG OpenDDS. Dentro la Struct è stata definita una sequence<octet> questo perché provando le varie possibili tipi complessi, la sequence è l'unica struttura complessa non deprecata nelle ultime versioni del framework, molto probabilmente dovuto al fatto che è l'unico tipo complesso in cui si deve definire e inviare prima la dimensione della sequenza evitando così vari tipi di attacchi di sicurezza informatica e una maggiore robustezza nella trasmissione dei dati. Infine, abbiamo la stringa che identifica quale publisher sta spedendo il messaggio.

Nel secondo Topic è stato definito una enum che esegue le varie azioni di apertura e chiusura delle connessioni. La enum è stata scelta per il fatto di avere il minor consumo di risorse per migliorare le prestazioni nell'esecuzione del programma, questo Topic è stato

importante per aprire e chiudere più connessioni avendo usato più publisher e subscriber in concorrenza durante l'esecuzione. In conclusione, il nome_publisher definiamo quale publisher stiamo trasmettendo l'evento di apertura o chiusura ai subscriber.

4.5.2 Classi Publisher e Subscriber

La creazione del codice delle classi Publisher e Subscriber è stato inizialmente utilizzato l'esempio di StockQuoter, fornito liberamente dagli sviluppatori di OpenDDS. Le modifiche apportate al codice sono sostanzialmente:

- Il publisher che il subscriber stampano nel videoterminale la dimensione del messaggio inviato e ricevuto, il numero di pacchetti scambiati dall'inizio dell'avvio dell'applicazione, il tempo impiegato di trasmissione tra il tempo attuale e il precedente messaggio trasmesso. Nelle due VM in cui viene eseguito il subscriber viene mostrato inoltre da quale publisher è stato pubblicato il messaggio e alla fine della trasmissione viene dato le informazioni di valutazione dei tempi come il tempo minimo, massimo, media e deviazione standard di trasmissione tra i vari messaggi ricevuti.

- È stato implementato la gestione concorrente tra più publisher e subscriber, utilizzando un Mutex e una lock definiti nelle API di ACE.

- I Topic trasmessi dai publisher sono stati riempiti da una sequenza di byte opportunamente definiti come misurazione di esempio delle prestazioni.

- Nel publisher è stata inserita una sleep subito prima della connessione per dare il tempo di pubblicare tutti i dati audio trasmessi.

4.5.3 File di configurazione

Il file di configurazione è stato preso inizialmente dall'esempio di StockQuoter ma è stato in parte cambiato a causa del fatto questo esempio come tutti quelli messi pubblici dai sviluppatori di OpenDDS non è prevista una prova con la comunicazione in computer differenti, sono scritti soltanto per comunicare tutti nello stesso computer non dovendo così impostare un indirizzo IP o un hostname, rischiando così di una incompleta comprensione interna del funzionamento di OpenDDS. Inoltre, la documentazione non è molto chiara

sulla parte di configurazione di questo file. In dds_udp_conf.ini sono state fatte diverse prove per avere la bassa latenza e il miglior funzionamento tra le cinque VM utilizzate.

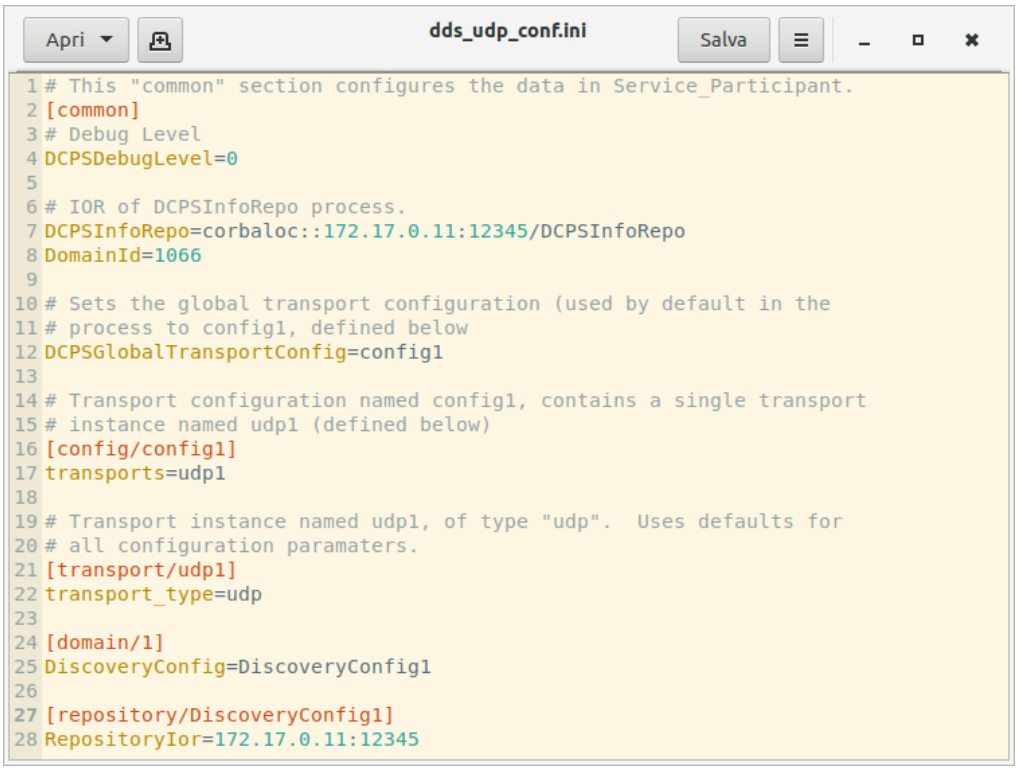

Figura 4.3: dds_udp_conf.ini

Come mostrato in alcuni commenti in Figura 4.3 abbiamo:

- DCPSDebugLevel è un valore che va da 0 a 10, è da i log in dieci livelli di profondità. Parametro molto utile per il primo periodo del lavoro di tesi per la comprensione degli errori di trasmissione.
- DCPSInfoRepo definisce l'indirizzo, la porta e il demone del dispatcher.
- DCPSGlobalTransportConfig imposta la configurazione di trasporto.
- transports definisce quale protocollo utilizzare per la trasmissione dei dati audio in questo caso è impostato in UDP ma può essere anche messo in TCP.
- DiscoveryConfig definisce in che modo gli oggetti di DDS vengono rilevati nel dominio.

- RepositoryIor viene usato per acquisire le informazioni che verrebbero fornite su una riga di comandi del terminale per puntare al servizio del dispatcher.

4.6 Dispatcher

OpenDDS ha un servizio stand-alone che è il Dispatcher dell'applicazione, chiamato DCPS Information Repository (DCPSInfoRepo) ed è necessario per consentire ai componenti OpenDDS di poter comunicare tra loro individuandosi a vicenda, pertanto è il primo componente da avviare.

Quando un subscriber richiede una sottoscrizione per un topic, il repository di informazioni DCPS individua il topic e lo notifica agli eventuali publisher della presenza del nuovo subscriber. Non ha il ruolo di propagazione dei dati, ma soltanto di discovery tra publisher e subscriber. La riga di comando del terminale che è stata usata per l'esecuzione del servizio è la seguente:

```
$ bin/DCPSInfoRepo -ORBListenEndpoints
                          iiop://172.17.0.11:12345 -d 1066
```

DCPSInfoRepo è il nome del servizio per l'esecuzione, -ORBListenEndpoints è per definire l'indirizzo e la porta del Dispatcher, mentre -d 1066 è il nome del dominio. Tutto questo potrebbe essere messo in un file di configurazione come viene fatto nel publisher e il subscriber.

4.7 Messaggi scambiati per instaurare la comunicazione

Analizzando varie tracce con Wireshark si è compreso che OpenDDS utilizza per lo scambio dei pacchetti per instaurare la connessione del Dispatcher con Publisher \Subscriber, la sintassi e gli specifici messaggi definiti in GIOP introdotto da CORBA 2.0.

Nella Figura 4.2 mostra tutti i pacchetti di OpenDDS che vengono scambiati da quando viene attivato il dispatcher e i due publisher (pacchetti identici vengono scambiati anche tra il dispatcher e i due subscriber) fino a quando gli oggetti hanno tutte le informazioni per comunicare direttamente senza intermediazione con il dispatcher. Il primo pacchetto trasmesso è mandato dal publisher per comunicare la sua presenza al dispatcher, e sarà un pacchetto TCP con dentro solo gli indirizzi IP e le porte sorgente e del

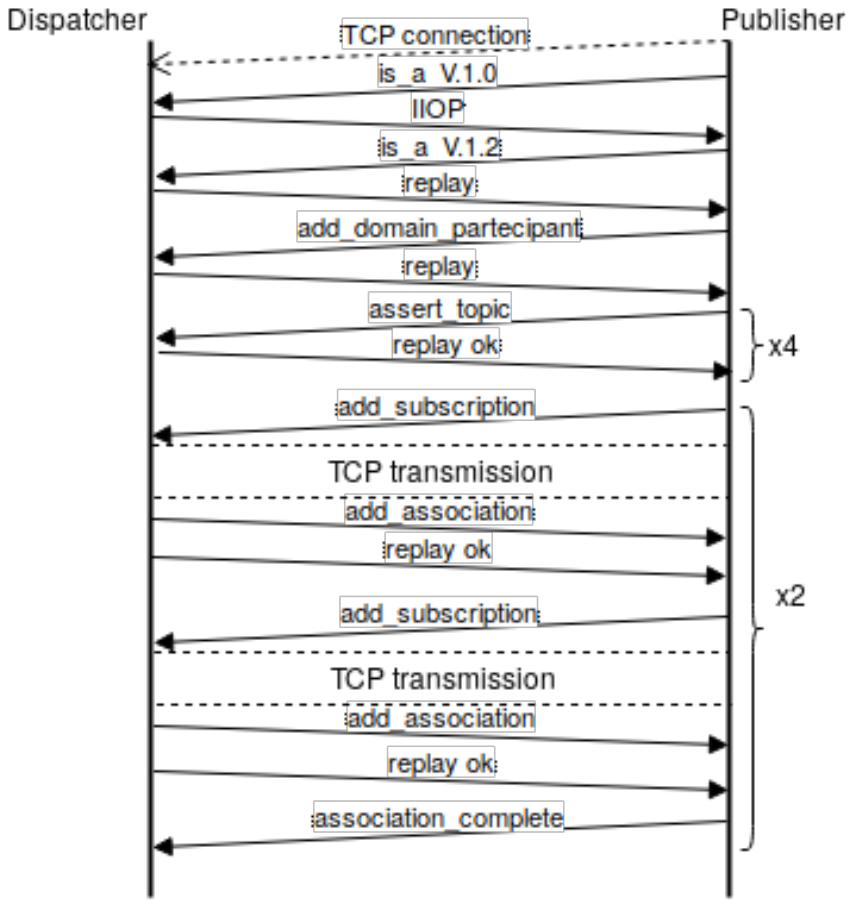

Figura 4.4: Apertura connessione con protocollo GIOP.

destinatario, l'informazione dell'indirizzo IP e la porta del dispatcher sono letti dal file di configurazione dds_tcp_conf.ini da parte del publisher. Subito dopo questo vengono trasmessi i pacchetti GIOP. I primi quattro messaggi scambiati viene negoziata quale versione di sintassi GIOP deve essere utilizzata per la comunicazione, in questo caso la versione 1.2. Il successivo viene richiesto da parte del publisher di aggiungersi come partecipante al dominio sempre definito nel file di configurazione. Poi vengono mandate le richieste assert_topic per la trasmissione delle informazioni sui topic. Infine, viene mandato add_subscription, add_association e association_complete per poter iniziare la vera e propria trasmissione dei dati dal publisher al subscriber.

Alla fine dello scambio dei dati in TCP, vengono trasmessi i messaggi per la chiusura della comunicazione come mostrato in Figura 4.5 con i seguenti messaggi in GIOP:

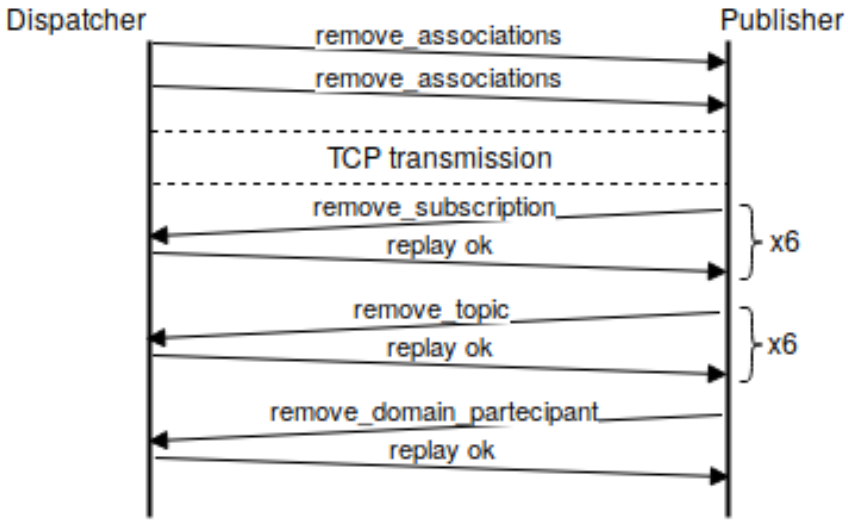

Figura 4.5: Chiusura connessione con protocollo GIOP.

remove_associations per segnalare la conclusione della trasmissione dei dati , remove_subscription per il rimuovere della sottoscrizione dalla trasmissione con il dispatcher, remove_domain_participant per la rimozione di partecipante dal dominio comunicato inizialmente. È evidente la notevole trasmissione di messaggi prima e dopo della comunicazione vera propria dei dati, ma questo porta maggiore robustezza e flessibilità di trasmissione.

Capitolo 5

Valutazione sperimentale di OpenDDS

L'utente per poter eseguire dei test deve fare una sequenza di operazioni e configurazioni:

1. Scelta dei parametri da provare, ponendo particolare attenzione che tipo di protocolli utilizzare.
2. Creazione del file di configurazione contenente le impostazioni.
3. Scelta del numero di publisher e subscriber da voler testare, scelta del loro indirizzo IP e hostname di ogni singolo nodo.
4. Installazione del middleware nei partecipanti.
5. Inserimento dei file necessari per i test sui nodi.
6. Esecuzione dei test.

Lo svolgimento di queste operazioni ha portato non pochi problemi a livello pratico. Nella fase di configurazione dei partecipanti, si ha alcuni problemi per l'impostazione dei nomi degli hostname, perché per creare il file di configurazione, si deve conoscere prima la struttura della rete. Nella scelta della configurazione di rete, per scegliere i partecipanti, deve conoscerne gli hostname, indirizzi IP e verificare la corretta installazione del middleware. Nel momento in cui la verifica dell'installazione non finisce correttamente si dovrà procedere all'installazione della piattaforma manualmente. La soluzione di questi problemi è stata nella configurazione in una macchina virtuale e di clonarla per il numero delle VM che erano richieste per le prove. Successivamente è stato preso un esempio fatto dagli sviluppatori di OpenDDS, modificarlo e fare delle aggiunte in varie fasi dello sviluppo.

5.1 Fase 1: Funzionamento e comportamento in TCP e in UDP

L'esempio che è stato sfruttato inizialmente illustra lo scambio di dati tra un publisher e un subscriber con il dispatcher che pubblicano e sottoscrivano dei dati campione attraverso il livello DCPS. L'esempio contiene due topic, entrambi relativi al mercato azionario. Un publisher pubblica le quotazioni azionarie, con il suo valore azionario e un timestamp. Inoltre, il publisher invia un altro topic per segnare l'apertura e la chiusura della borsa. Invece, un subscriber sottoscrivere e stampa a video i risultati. Questa fase mostra che non modificando il codice del subscriber e del publisher, ma cambiando soltanto un parametro di configurazione si può comunicare tramite protocollo TCP e UDP. Questo perché la configurazione del protocollo di trasporto è isolata in un gruppo di file, permettendo di cambiare protocollo di trasporto senza apportare modifiche al codice.

OpenDDS include un meccanismo di configurazione basato su file. Un utente di OpenDDS potrebbe configurare il protocollo di trasporto in un publisher e di un subscriber attraverso il file di configurazione come in questo caso.

5.1.1 Registrazione degli eventi in OpenDDS

Il framework OpenDDS registra un errore soltanto quando si verifica un problema grave non indicato nel codice di ritorno. Però lo sviluppatore può aumentare la quantità di registrazione tramite i controlli che avvengo tra il livello del DCPS e di trasporto.

La registrazione a livello di DCPS di OpenDDS è regolata da un'opzione di configurazione inseribile nel terminale prima dell'esecuzione del dispatcher. Oppure può essere impostato a livello di codice nella programmazione [8]. Il livello predefinito è 0 e ha valori che vanno da 0 a 10 come di seguito evidenzio i più importanti:

- 0 indica gli errori gravi non indicati nel codice di ritorno, quasi niente.
- 6 indica la possibile lettura/scrittura di esempio che dovrebbe accadere.
- 10 registra tutti gli eventi che si verificano per ogni tipo di lettura/scrittura.

Applicando nel servizio dispatcher l'opzione: -ORBDebugLevel si è giunti inizialmente a risolvere molti problemi non banali e a comprendere il funzionamento iniziale di questo esempio.

5.2 Fase 2: Secondo subscriber e modifica dei messaggi scambiati

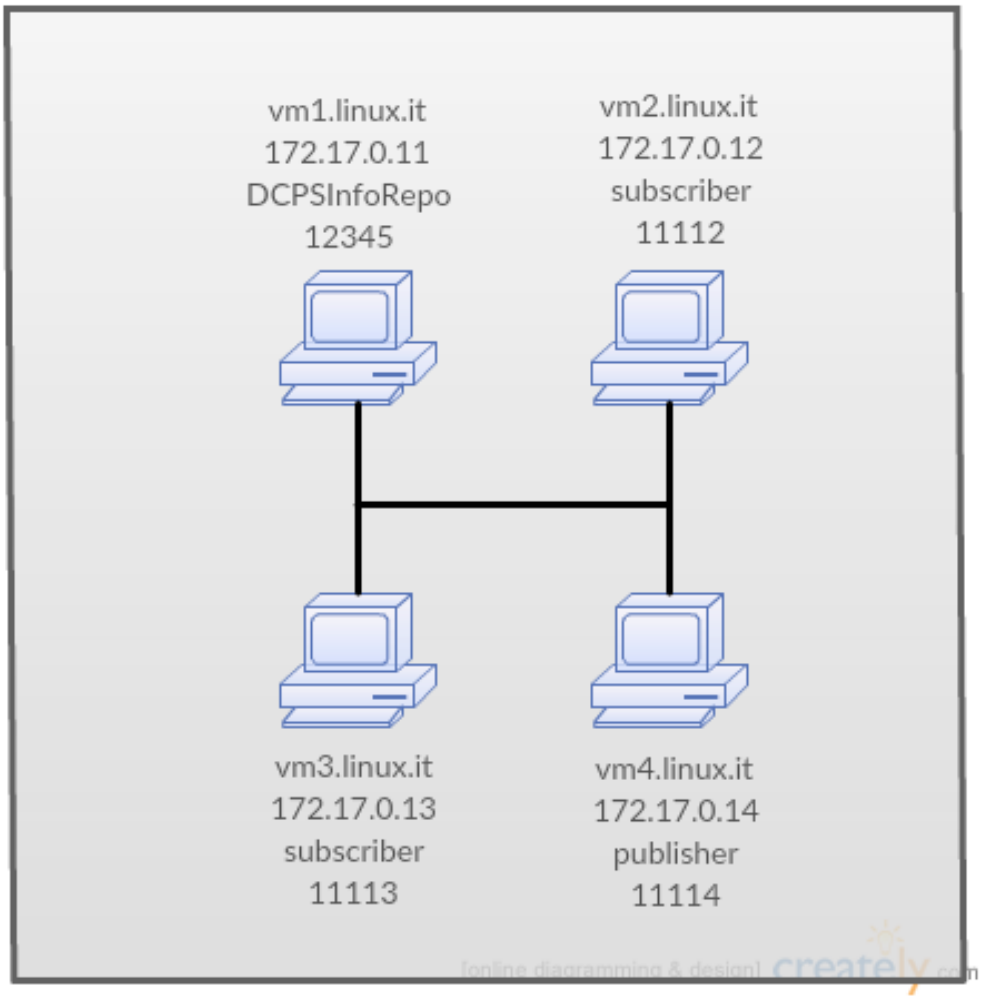

Figura 5.1: Test con secondo Subscriber.

In Figura 5.1 mostra una prova con l'aggiunta di un secondo Subscriber. Dato che nella fase precedente si è visto il corretto funzionamento nel livello di Trasporto del protocollo TCP che sia l'UDP, si è deciso di utilizzare soltanto da ora in avanti il protocollo UDP per il proprio obiettivo di tesi, dato che l'interesse principale è quello di sfruttare una comunicazione low-latency. Infine, si è cercato di capire come potessero ricevere informazioni differenti tra i due subscriber. In conclusione, si è visto che i subscriber possono sottoscriversi a flussi di dati differenti se si iscrivano a domini diversi. Avendo

modificato uno dei due Topic in modo tale da ricevere Publishing segnale operatore 1 e Publishing segnale operatore 2, nella Figura 5.2 abbiamo l'inizio della modifica di uno dei due Topic trasmesso da un publisher e ricevuti gli stessi messaggi a due subscriber differenti.

```
valore:segnale          = 1
       dominio          = Segnale operatore 1
       valore           = 1210
       timestamp        = 34eae3f094c495
CampioneInfo.campione_rank        = 0
CampioneInfo.instanza_handle      = 13
CampioneInfo.pubblicazione_handle = 10
valore:segnale          = 2
       dominio          = Segnale operatore 2
       valore           = 1410
       timestamp        = 34eae3f0bb2777
CampioneInfo.campione_rank        = 0
CampioneInfo.instanza_handle      = 14
CampioneInfo.pubblicazione_handle = 10
ScambioEvento: scambiato = Task Segnali Scambiati
               segnale     = CHIUSURA_CONNESSIONE
               timestamp = 34eae3f0e19673
CampioneInfo.campione_rank        = 0
CampioneInfo.instanza_handle      = 12
CampioneInfo.pubblicazione_handle = 11
INFO: lettura completa dopo 1 campioni.
ScambioEvento: scambiato = Task Segnali Scambiati
CampioneInfo.campione_rank        = 0
CampioneInfo.instanza_handle      = 12
CampioneInfo.pubblicazione_handle = 11
INFO: lettura completa dopo 1 campioni.
Ricevuto segnale CLOSED dal publisher all'uscita...
```

Figura 5.2: Test stampe con secondo Subscriber.

5.3 Fase 3: Utilizzo di un secondo Publisher e gestione della concorrenza

La fase successiva del lavoro ha portato alla decisione di togliere il secondo subscriber e di mettere invece in più un publisher per trasmettere gli stessi due Topic che si trovano nel dominio definito in DDS ma trasmettere messaggi differenti tra primo e secondo

Publisher per la pubblicazione. Da notare che si è preso l'architettura di rete Figura 5.1 ma la vm3.linux.it si è convertita in publisher.

Si è fatto uno studio del funzionamento dello standard CORBA per mantenere l'interoperabilità del codice e si è potuto sfruttare le API di ACE che implementano metodi per la gestione dei Mutex e lock_guard. Con il Mutex si è potuto inserire nel codice del progetto un contatore per contare il numero di chiusure e aperture di canale delle varie trasmissioni avendo così la possibilità di gestire più publisher in modo efficiente e con bassa latenza di trasmissione dati. Per gestire il Mutex si è usato un booleano per il lock in cui veniva chiuso e rilasciato quando si richiamava il metodo di inizio e fine della connessione.

Figura 5.3: Test stampe Publisher.

Infine, si vede nella Figura 6.2 uno dei due Publisher, e in Figura 6.3 il Subscriber con i vari messaggi scambiati. Nelle stampe dei terminali abbiamo la trasmissione del Publishing segnale operatore 1 da parte della vm4 e di Publishing segnale operatore 2 dell'altro publisher, infine abbiamo a destra la stampa della sottoscrizione del subscriber in cui mostra i differenti informazioni da parte dei publisher. In questa fase si è cercato di provare il funzionamento del multicast, che nella documentazione di OpenDDS ci sono le specifiche per poter utilizzare un multicast affidabile o non affidabile ma durante le prove

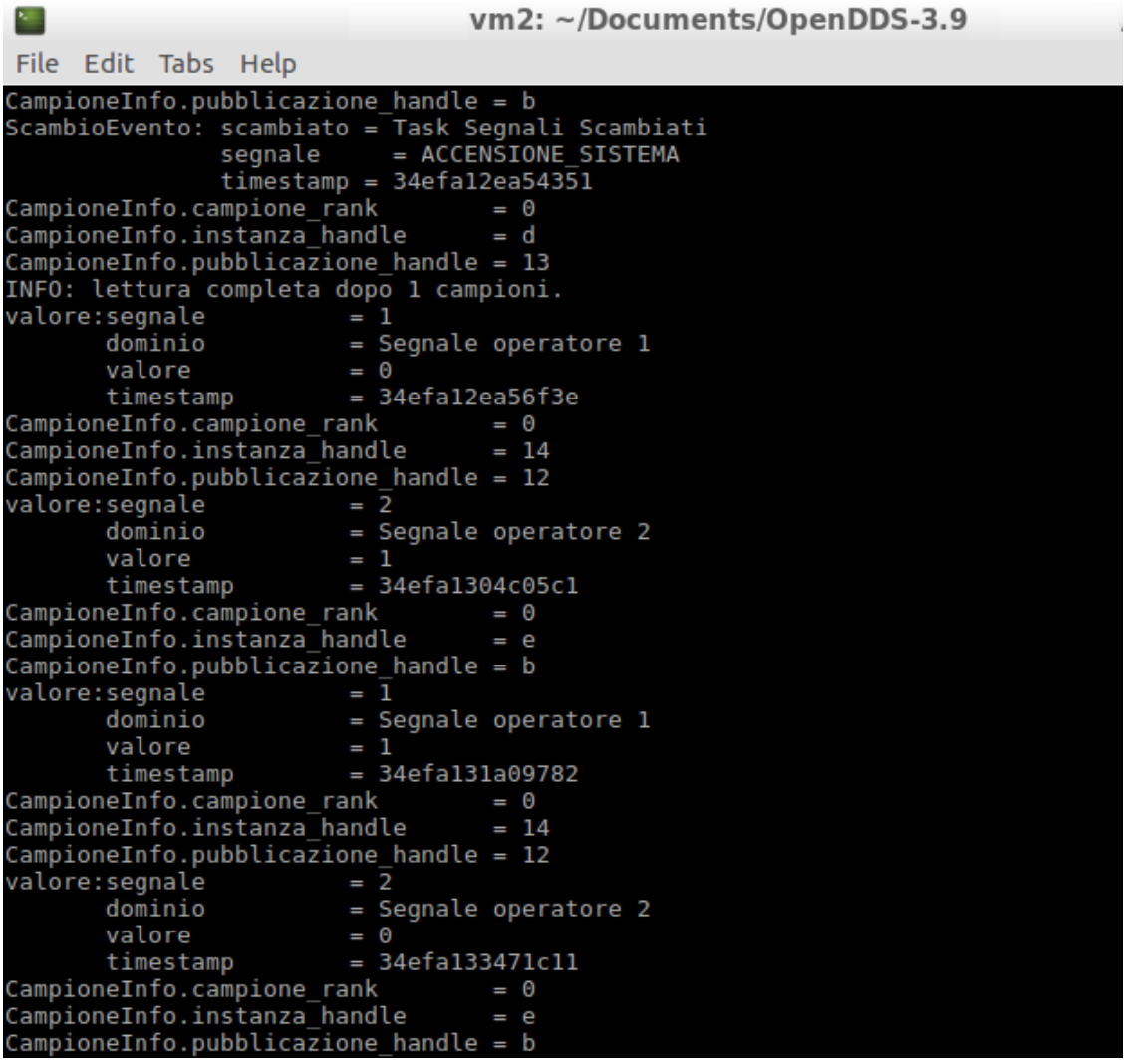

Figura 5.4: Test stampe Subscriber.

non si è riuscito a trasmettere i dati anche se la fase iniziale di connessione tra dispatcher e i partecipanti avvenivano. Si ipotizza che sia dovuto al fatto che fosse necessario una nuova progettazione del codice di programmazione, infatti successivamente si sono trovate delle funzioni specifiche per la trasmissione del multicast.

In conclusione, questa prova mostra una bassa latenza per la trasmissione dei dati audio, un'ottima gestione delle risorse del sistema utilizzando poca potenza di calcolo come si ha nella macchina virtuale utilizzata.

5.4 Fase 4: Trasporto pacchetti RTP

La quarta e ultima fase del lavoro di tesi è stata quella di cambiare completamente tutta la struttura dei due topic per scambiare il dato audio che era prefissato nell'obiettivo della tesi. È stato definito un topic con una sequenza di byte per la trasmissione di pacchetti dati che contenessero la dimensione minima di una codifica PCM di una sequenza di byte che rappresentasse il campionamento di un segnale audio.

La sequenza di byte che contengono il flusso audio di ogni singolo pacchetto trasmesso è di 160 Byte in cui è stato imbustato in un header RTP da 12 byte. Il protocollo RTP è stato scelto per il fatto che è un protocollo progettato per il trasporto di dati multimediali, che affronta gran parte dei problemi dovuti alla gestione, monitoraggio dello scambio di dati real-time, in cui i dati non sono solo un flusso di bit: ogni pezzo di sequenza ha specifici riferimenti temporali. Inoltre, per sviluppi successivi al lavoro di tesi si possono rilevare problemi nella comunicazione come i pacchetti mancanti o riordinati. Si ha pure con RTP la capacità di poter adeguare la larghezza di banda, velocità di trasmissione e non devono essere gestiti dal protocollo nei strati protocollari più bassi, scalabilità, possibilità di supportare unicast e multicast con un grande numero di partecipanti, supporto di applicazioni multimediali senza dare limiti alla loro logica, ma con nessun algoritmo di adattamento, mancanza di supporto per la Quality Of Service (QoS) o prenotazione delle risorse. È stato aggiunto una quinta macchina virtuale per creare uno scambio dati tra due publisher, due subscriber e un dispatcher come mostrato in Figura 4.1. Dato che è stata cambiata la struttura dei topic, sono state cambiate le stampe completamente in ogni macchina virtuale. Inoltre, si è aumentato il numero di pacchetti e la velocità di trasmissione portando l'invio di 3000 dati con una latenza media tra un uno e l'altro di 20 ms. Simulando così il traffico dati per 1 minuto di trasmissione. Viene calcolato anche la latenza nella ricezione dei messaggi da parte del subscriber stampando il tempo della media, la deviazione standard, il minimo e il massimo alla fine della trasmissione dei dati. E per mandare il tempo esatto di ogni pacchetto è stato usato nel publisher la funzione gettimeofday() in cui stampo il timestamp di ogni pacchetto mandato nel Publisher come mostrato in Figura 5.6 e ricevuto dal Subscriber come in Figura 5.7 e nel Dispatcher non avremo nessuna stampa perché ha solo il compito di instaurare la connessione tra i partecipanti come in Figura 5.5.

Infine, il calcolo dei tempi di latenza nella trasmissione è stato fatto calcolando i tempi inserendo nel codice sorgente un timestamp del pacchetto ricevuto e un altro timestamp

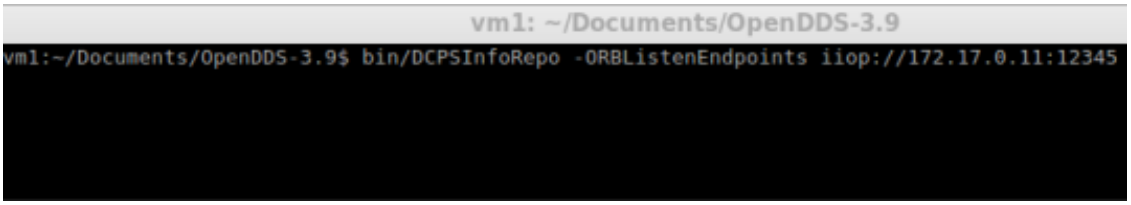

Figura 5.5: Stampa terminale del Dispatcher.

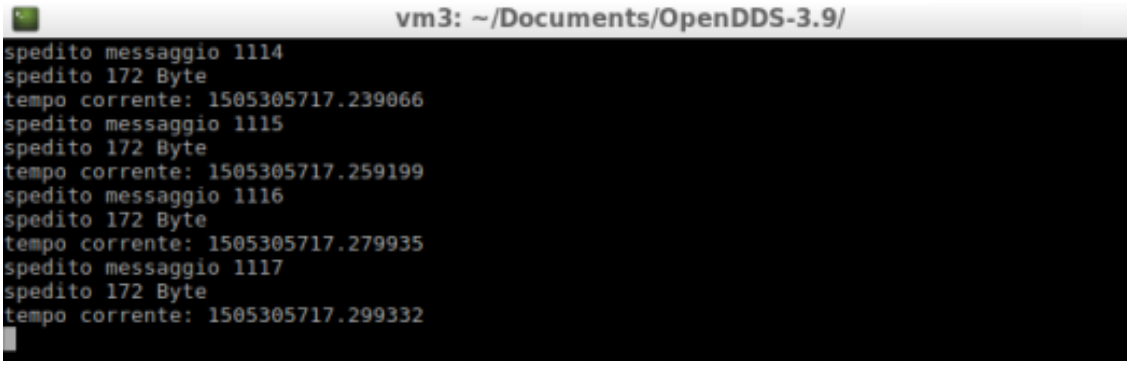

Figura 5.6: Stampa terminale di uno dei due Publisher.

nel codice sorgente per la ricezione del pacchetto, inoltre i tempi di trasferimento da un messaggio all'altro è stato regolato in base al consumo di banda cioè se dipende dal tempo impiegato alla trasmissione viene tolto o aggiunto tempo per il trasferimento del messaggio successivo per mantenere la latenza di 20 ms, infatti avremo una media del traffico totale di 20-19 ms per ogni topic trasmesso, un valore massimo tra due topic di 31 ms, ad un altro estremo minimo di 7 ms e una deviazione standard di 2,5 ms.

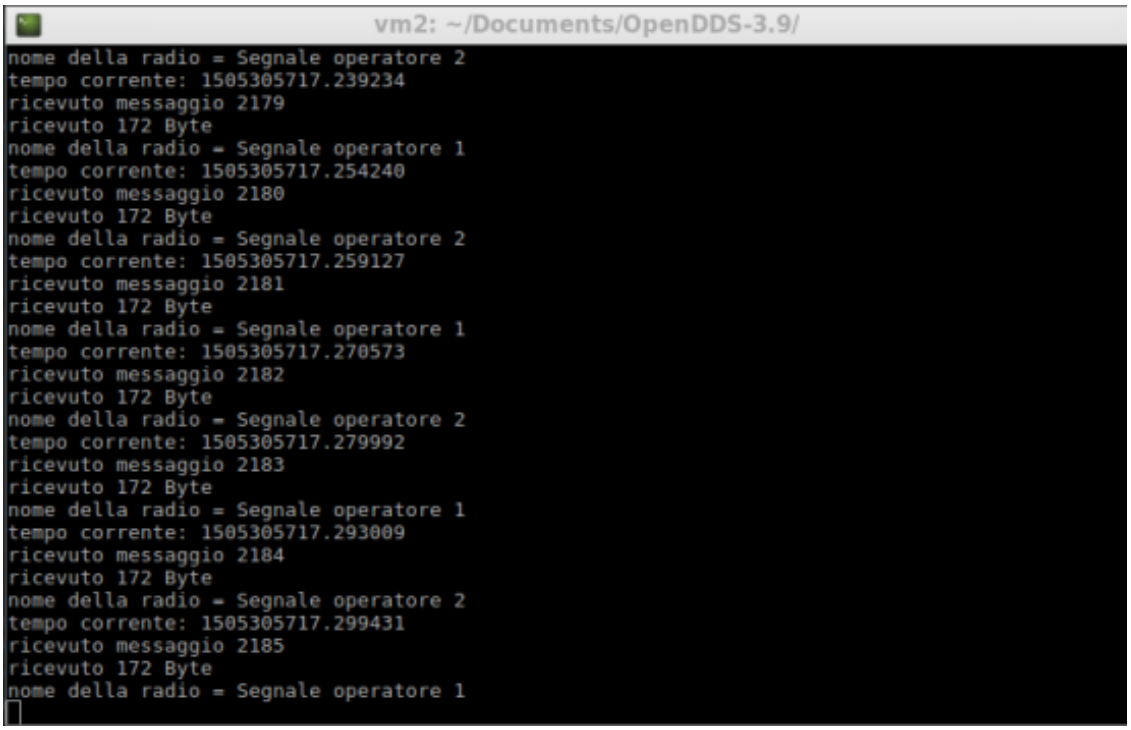

Figura 5.7: Stampa terminale di uno dei due Subscriber.

Capitolo 6

Risultati

Il sistema distribuito realizzato durante lo svolgimento del lavoro di tesi è stato creato per lo studio della fattibilità di utilizzare un sistema distribuito come OMG DDS e che fosse implementato con software open source come OpenDDS. Questo ha portato la comprensione dello scambio dei messaggi anche senza la completa conoscenza della struttura interna del middleware, mantenendo così la flessibilità rispettando come requisito fondamentale la consegna a bassa latenza dei messaggi.

Dalle analisi effettuate delle tracce dei pacchetti con l'utilizzo del protocollo UDP si è potuto osservare che lo stesso, soddisfacendo i requisiti di interoperabilità, consente di soddisfare le esigenze di low-latency. Questo risultato è dovuto anche grazie dal fatto che si è limitato la dimensione di ogni singolo messaggio scambiato, non superando i 160 Byte di dati utili trasmessi e cercando sempre di trasmettere i messaggi ad una latenza accettabile tra un pacchetto e il successivo. Oltre ai dati utili è stato messo un header RTP da 12 Byte per gli sviluppi futuri.

Durante lo svolgimento dell'applicazione sono state fatte due demo, una a metà del lavoro e un'altra alla fine utilizzando il task manager della distribuzione GNU/Linux usata per visualizzare il consumo delle risorse del processore e della RAM. Nella fase finale è stato utilizzato delle funzioni in C per catturare il timestamp sia di ogni messaggio spedito dal publisher e sia per ognuno ricevuto dal subscriber. Mediante queste informazioni si sono ottenute tutti i dati per comprendere i consumi delle risorse di sistema e le prestazioni di trasmissione dei pacchetti per capire se rispettasse la latenza che deve avere un sistema low-latency.

6.1 Prestazioni della fase 3

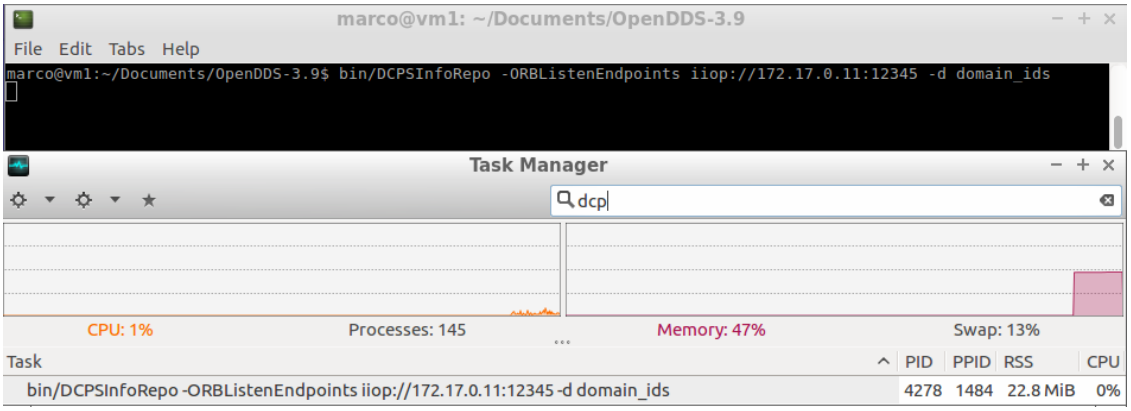

Figura 6.1: Prestazioni Dispatcher nella fase 3.

In questa fase del processo si è modificato il progetto in modo tale da poter gestire più Publisher in concorrenza, il problema più grosso è stato comprendere il funzionamento delle classi di OpenDDS per poter prendere e rilasciare le risorse nel momento opportuno. In dettaglio il Subscriber ha una variabile condivisa in cui poter contare quante connessioni aprire e alla fine chiudere per poter rilasciare le risorse.

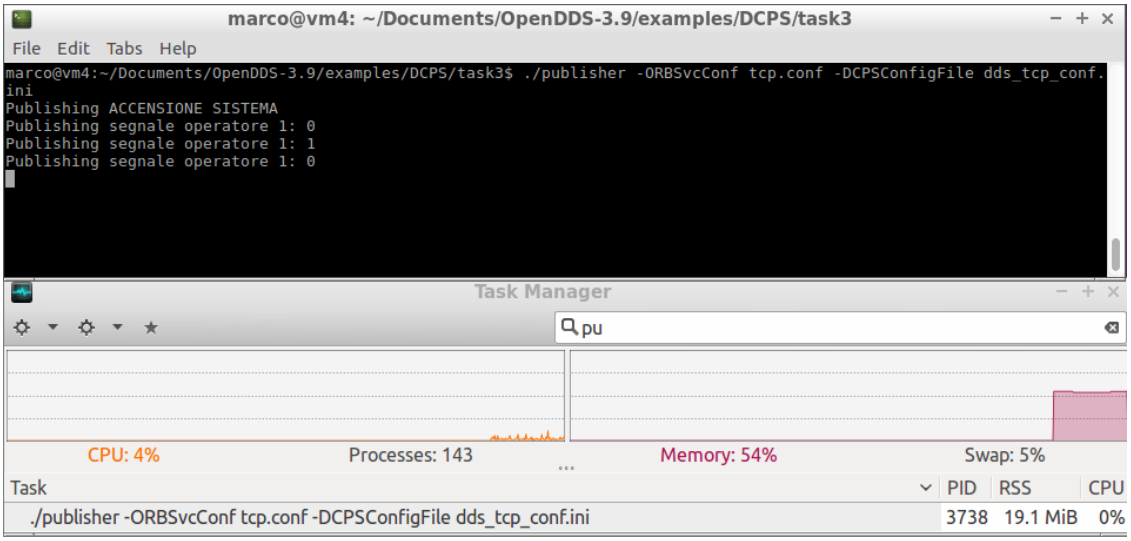

Figura 6.2: Prestazioni Publisher nella fase 3.

Nella Figura 6.1 abbiamo il dispatcher che dato dal fatto che deve solo creare la comunicazione iniziale tra i partecipanti ha una bassa richiesta di risorse di memoria e di

processore, ha solo alcuni MB per mantenere le informazioni dei partecipanti.

Nella Figura 6.2 mostra la connessione di un Publisher e l'informazione di vari messaggi trasmessi, anch'esso il task manager di GNU/Linux da un basso uso delle risorse della macchina virtuale.

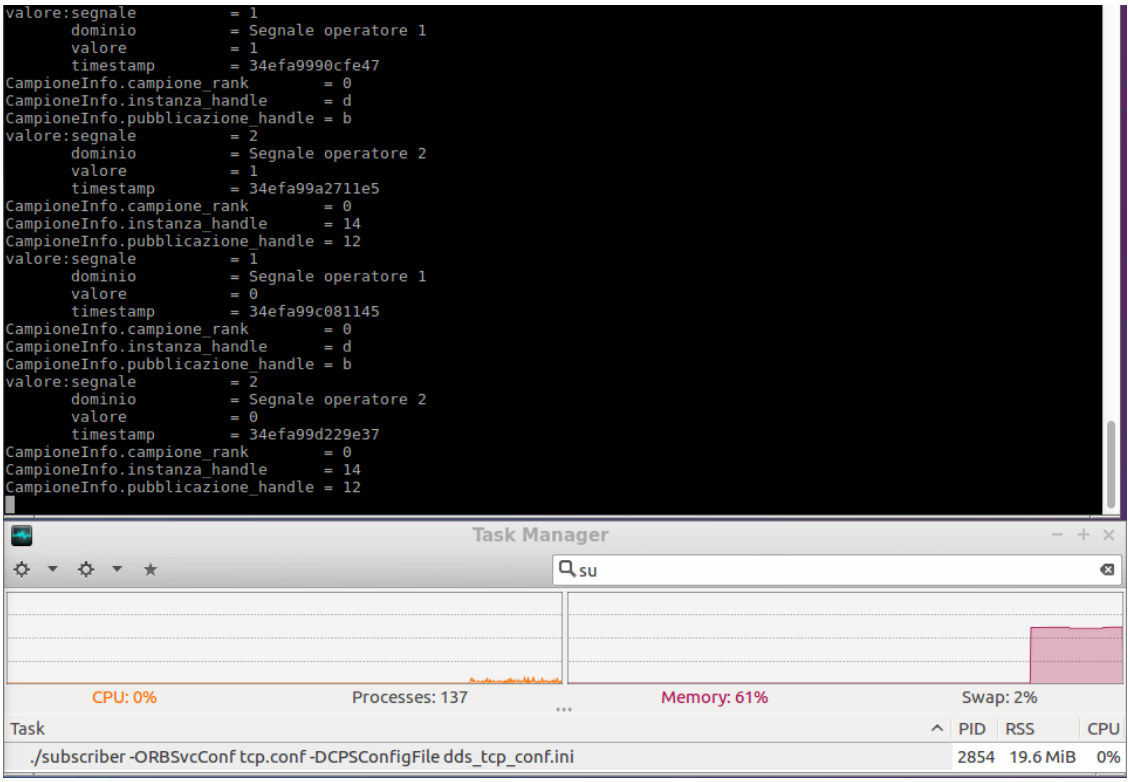

Figura 6.3: Prestazioni Subscriber nella fase 3.

In conclusione, nella Figura 6.3 mostra le stampe dei messaggi sottoscritti dal Subscriber, nel task manager abbiamo sempre valori di consumi relativamente contenuti.

6.2 Prestazioni della fase 4

Nell'ultima fase del sistema distribuito sono stati usati degli scambi di dati tra due publisher e due subscriber con il relativo dispatcher per informare e instaurare la connessione tra i partecipanti.

I pacchetti sono stati trasmessi con una latenza di circa 20 ms ma trasmettendo pacchetti di dimensione fissa di 172 Byte, questo ha portato dei ottimi risultati nel

consumo di risorse che si discostano davvero poco la fase 3 che sono rappresentate nel paragrafo precedente.

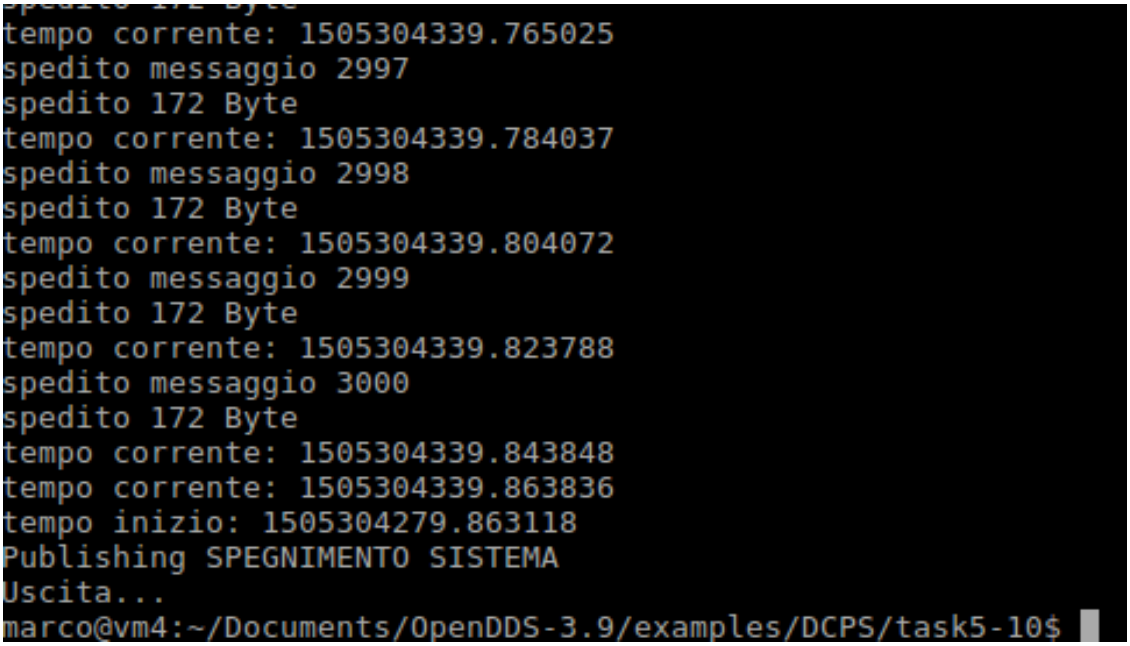

Figura 6.4: Stampe Publishing.

Nella Figura 6.4 abbiamo il conteggio dei messaggi spedititi da parte del Publisher fino a 3000 pacchetti come è stato precedente definito, abbiamo la dimensione del messaggio e il timestamp di ogni Topic spedito. Invece in Figura 6.5 mostra uno dei due subscriber le stampe dei messaggi ricevuti dalla pubblicazione dei due Publisher perciò avremo 6000 topic arrivati a destinazione. Alla chiusura della connessione con gli altri Publisher abbiamo le stampe delle informazioni di latenza da parte dei due partecipanti che hanno trasmesso.

In ordine delle stampe rappresentate in figure abbiamo il valore dei tempi di ricezione minimi, massimi, la media di tutti i messaggi ricevuti e la deviazione standard del primo e secondo publisher. I valori risultanti dimostrano la fattibilità e l'efficienza del sistema distribuito utilizzato, anche se c'è stato un minimo di distanza temporale tra due messaggi inaspettatamente molto bassa, la latenza massima non ha mai raggiunto i 40 ms che è una prestazione ottima anche dal fatto che la media è intorno i 19-20 ms.

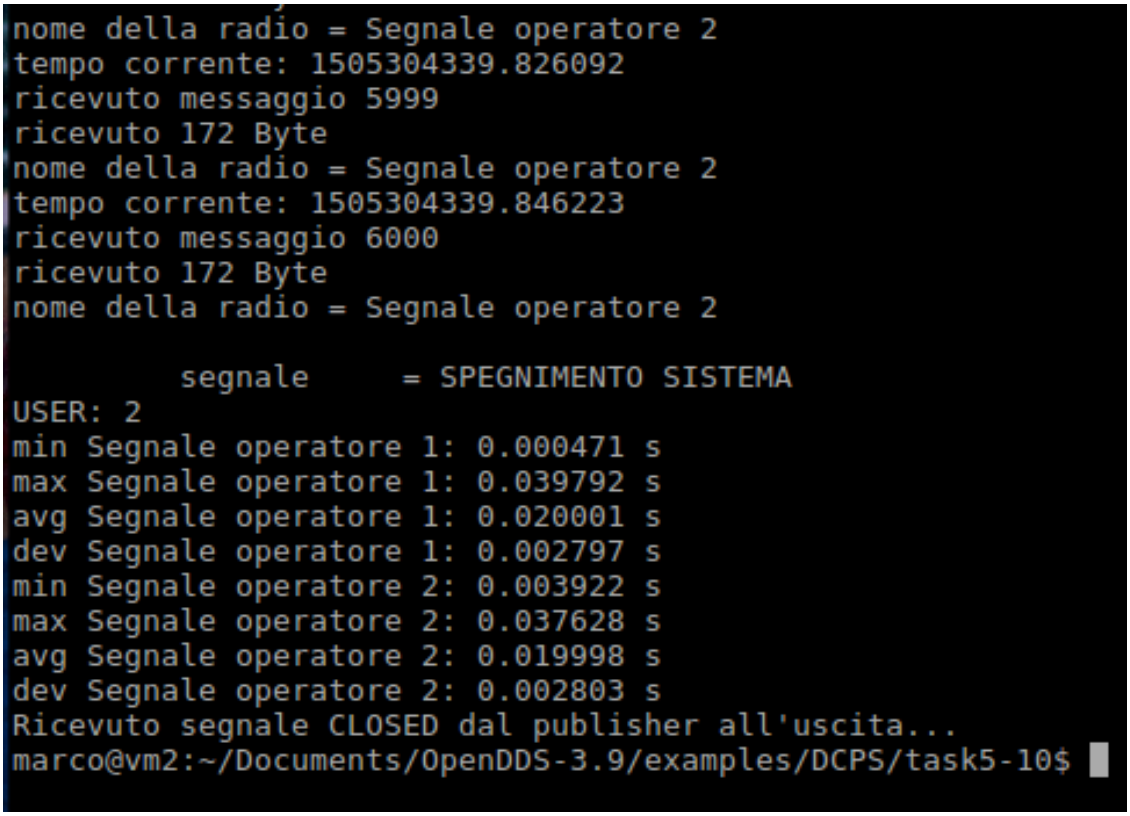

Figura 6.5: Stampe Subscriber.

Capitolo 7

Conclusioni

L'obiettivo principale del presente elaborato è stato quello di realizzare un sistema distribuito per una comunicazione real-time usando del software open source. Per lo sviluppo di tale sistema si è fatto uso della documentazione e gli esempi per sviluppatori di OpenDDS [8], la quale espone i componenti per l'utilizzo dell'implementazione dello standard e alcuni esempi funzionanti in rete locale, in particolare si è preso come esempio StockQuoter.

Si è riusciti a comprendere il funzionamento e nella fase finale si è potuto creare un Topic per la trasmissione dei dati utili per la comunicazione multimediale. Infine, sono state fatte delle stampe del terminale e il task manager per mostrare i risultati ottenuti. I vantaggi di questa implementazione sono la grande scalabilità, performance e interoperabilità, tutto questo grazie all'applicazione di una licenza open source, un utilizzo di C+11 e i paradigmi di gestione della concorrenza e il rispetto dello standard OMG DDS. Gli svantaggi sono il numero di buona parte dei tipi complessi in IDL, in cui si può scegliere solo sequence e i tipi primitivi. La costrizione di usare un Dispatcher per mediare le connessioni tra i partecipanti e ultimo ma non meno importante la difficoltà di riconoscere correttamente il protocollo GIOP da parte dei analizzatori di traffico.

Possibili sviluppi futuri potrebbero riguardare l'utilizzo di librerie audio per la registrazione e trasmissione in tempo reale con OpenDDS, a fine di mostrare il comportamento nelle prestazioni utilizzando la complessità d'uso di altre librerie al di fuori del framework. Invece, per analizzare il traffico si potrebbe implementare i plugin appositi per rappresentare correttamente il traffico creato da OpenDDS. Durante la fase di sperimentazione sono state utilizzate diverse macchine virtuali nello stesso computer con

Oracle VM VirtualBox potrebbe essere un'ottima soluzione continuare le valutazioni sperimentali usando anche computer differenti.

Appendice A

Configurazione Macchine Virtuali

A.1 Requisiti del sistema distribuito utilizzato

Hardware:

- CPU:i5 1a generazione
- Memoria RAM: 4GB
- OS: Debian GNU/Linux o derivate

Software:

- VirtualBox 5.1
- Si consiglia di installare VirtualBox Extension Pack per una migliore gestione del sistema ospite. Disponibile in: https://www.virtualbox.org/wiki/Downloads

A.2 Creazione delle macchine virtuali

Requisiti consigliati per ogni Macchina Virtuale:

- RAM: 256 MB
- SO utilizzato: Lubuntu 16.04.1 64 bit

- Scaricare l'immagine ISO della versione alternate del Sistema Operativo dall'indirizzo:
  cdimages.ubuntu.com/lubuntu/releases/

- Prima dell'esecuzione della VM andate sulle impostazioni e fate le seguenti modifiche:

- Tasto destro sulla VM: Impostazioni → Archiviazione
  aggiungete l'ISO del Sistema Operativo da installare (lubuntu-16.04.1-alternate-amd64.iso in questo caso).

- Adesso impostiamo due reti virtuali, una per la comunicazione tra le varie VM e l'altra usata per scaricare gli aggiornamenti del software: Impostazioni → Rete

- Create due reti virtuali:
  Scheda 1 → Rete Interna
  Scheda 2 → NAT (abilitando la Network Adapter)

A.3 Primo avvio della macchina

- Adesso possiamo eseguire la VM ed iniziare l'installazione del Sistema Operativo nella VM. Dopo di che apriamo il terminale e andiamo a configurare l'hostname della VM quanto segue:

```
$ sudo nano /etc/network/interface

allow-hotplug enp0s3
iface enp0s3
address 192.168.1.2
netmask 255.255.255.0
broadcast 192.168.1.255

allow-hotplug enp0s8
iface enp0s8 inet dhcp
```

Adesso avremo le due schede virtuali correttamente configurate. Da controllare prima se il nome delle schede di rete ha un nome differente, ad esempio potrebbero essere eth0 e eth1, si può controllare con ifconfig.
Per essere sicuri che sono avvenute correttamente le modifiche sulle due schede di rete virtuali, riavviamo il Sistema Operativo e facciamo alcune prove con il servizio ping per essere sicuri che le macchine virtuali comunicano.

- Soddisfiamo le dipendenze per installare Oracle VM VirtualBox Extension Pack e OpenDDS:

```
$ sudo apt-get update && sudo apt-get upgrade
$ sudo apt-get install build-essential
$ sudo apt-get install linux-headers-$(uname -r)
```

- Installiamo Oracle VM VirtualBox Extension Pack andando sul menu di Oracle VM VirtualBox
  Dispositivi → Inserisci l'immagine del CD della Guest Additions.

- Successivamente dal terminale eseguiamo:

```
$ cd /media/NOME-UTENTE-DELLA-VM/VBOXADDITIONS_5.*/
$ sudo -s
# ./VBoxLinuxAdditions.run
```

- Bisogna impostare l'hostname modificando /etc/hostname mettendo

```
vm1
```

- Infine, modificare /etc/hosts configurando in questo modo:

```
172.17.0.11        vm1.linux.it vm1
```

- Riavviamo la VM per applicare le modifiche.

Appendice B

Riprodurre esempio di comunicazione

Questo esempio è stato utilizzato per poter trasferire una sequenza di byte tra due publisher e due subscriber. In questa comunicazione avremo i 160 Byte per contenere le informazioni di campionamento per una trasmissione audio, sommato a 12 Byte di payload del pacchetto RTP per un totale di 172 Byte.

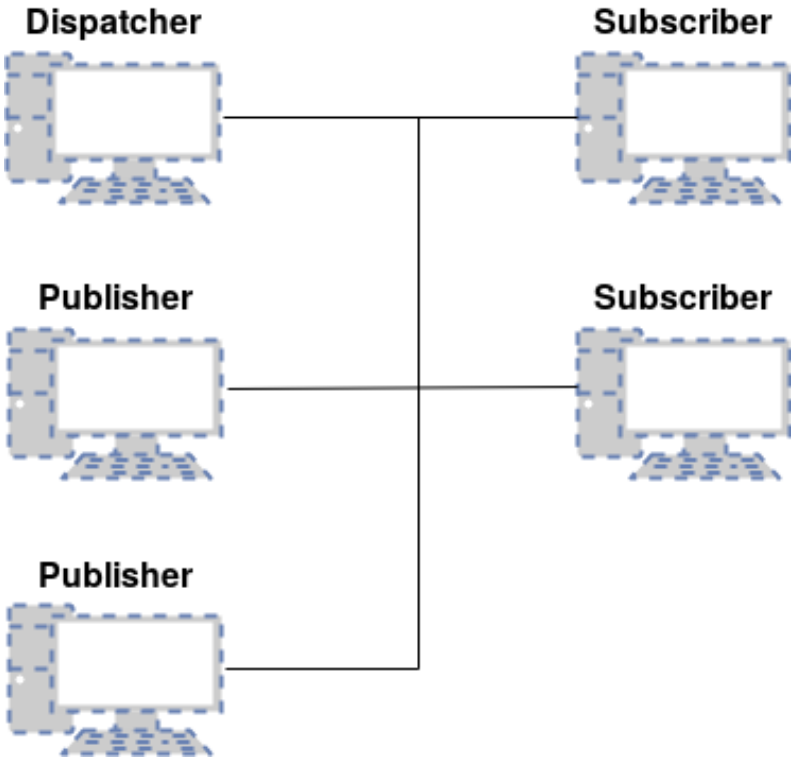

Figura B.1: Architettura di rete esempio.

B.1 Architettura di rete

Nella Figura B.1 qui sopra mostra la struttura della rete come configurazione di ogni singola macchina virtuale:

- Nome del dominio delle VM (fondamentale per la trasmissione): linux.it
- Hostname: vm1 per il dispatcher, vm2 e vm4 per i subscriber, vm3 e vm4 per i publisher
- Le porte per i canali di comunicazione sono stati scelti differenti in ogni VM solo per poter capire con quale macchina si sta comunicando, quando si ha bisogno di avere una rapida visione della traccia del traffico di rete, quindi non è obbligatorio usare porte differenti.

B.2 Esecuzione di ogni VM

Ipotizzando che OpenDDS è stato installato nella cartella /home/NOME-UTENTE-VM/Documenti .

Avremo le seguenti operazioni da eseguire dal terminale in ogni VM:

- vm1

```
$ cd ~/Documenti/OpenDDS-3.9 && source setenv.sh
$ bin/DCPSInfoRepo
        -ORBListenEndpoints iiop://172.17.0.11:12345 -d 1066
```

- vm2 e vm5

```
$ cd ~/Documenti/OpenDDS-3.9 && source setenv.sh
$ cd examples/DCPS/esempio
$ ./subscriber -DCPSConfigFile dds_udp_conf.ini
```

- vm3 e vm4

```
$ cd ~/Documenti/OpenDDS-3.9 && source setenv.sh
$ cd examples/DCPS/esempio
$ ./publisher -DCPSConfigFile dds_udp_conf.ini
```

B.3 File di configurazione

Il file di configurazione dds_udp_conf.ini è mostrato qui sotto:

```
# This "common" section configures the data in Service_Participant.
[common]

# Debug Level
DCPSDebugLevel=0

# IOR of DCPSInfoRepo process.
#DCPSInfoRepo=corbaloc::localhost:12345/DCPSInfoRepo
DCPSInfoRepo=corbaloc::172.17.0.11:12345/DCPSInfoRepo
DomainId=1066

# Sets the global transport configuration (used by default in the
# process to config1, defined below
DCPSGlobalTransportConfig=config1


# Transport configuration named config1, contains a single
# transport instance named udp1 (defined below)
[config/config1]
transports=udp1


# Transport instance named udp1, of type "udp".
Uses defaults for
# all configuration paramaters.
[transport/udp1]
transport_type=udp
#local_address=172.17.0.2

[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=172.17.0.11:12345
```

B.4 Compilazione dei file di configurazione

- Compilazione del Topic, dal terminale nella cartella dove si trova il progetto:

```
$DDS_ROOT/bin/opendds_idl StockQuoter.idl
```

- Compilazione di tutte le classi:

```
$ACE_ROOT/bin/mwc.pl -type gnuace StockQuoter.mwc && make
```

- Prima di ogni compilazione se sono stati modificati le classi o le interfacce:

```
make realclean
```

Bibliografia

[1] Patrick Eugster et al. “The Many Faces of Publish/Subscribe”. In: 35 (giu. 2003), pp. 114–131.

[2] Insup Lee, Joseph YT Leung e Sang H Son. *Handbook of real-time and embedded systems*. CRC Press, 2007.

[3] Gerardo Pardo-Castellote. “Omg data-distribution service: Architectural overview”. In: *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*. IEEE. 2003, pp. 200–206.

[4] *Data Distribution Service*. 1.4. Mar. 2015. URL: http://www.omg.org/spec/DDS/1.4.

[5] John Vlissides et al. “Design patterns: Elements of reusable object-oriented software”. In: *Reading: Addison-Wesley* 49.120 (1995), p. 11.

[6] Joseph M Schlesselman, Gerardo Pardo-Castellote e Bert Farabaugh. “OMG data-distribution service (DDS): architectural update”. In: *Military Communications Conference, 2004. MILCOM 2004. 2004 IEEE*. Vol. 2. IEEE. 2004, pp. 961–967.

[7] Gerardo Pardo-Castellote, Bert Farabaugh e Rick Warren. “An introduction to DDS and data-centric communications”. In: *RTI, Aug* (2005).

[8] Object Computing. *Inc. OpenDDS Developer’s Guide*.

[9] Douglas Schmidt e Stephen D Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley Professional, 2002.

[10] *Rhapsody CORBA Development - User Guide*. Release 2.3. I-Logix Inc. 2000.

[11] Michi Henning e Steve Vinoski. *Advanced CORBA® programming with C++*. Pearson Education, 1999.

[12] Dave Bartlett. “Under the hood: IORs, GIOP and IIOP”. In: *Aug* 1 (2000), pp. 1–7.

[13] H Schulzrinne et al. “RFC 3550”. In: *RTP: a transport protocol for real-time applications* 7 (2003).

[14] Wikipedia. *Session Initiation Protocol*. [Online; controllata il 22-febbraio-2018]. 2018. URL: https://it.wikipedia.org/wiki/Session_Initiation_Protocol.