

UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
GIOVANNI DEGLI ANTONI



CORSO DI LAUREA IN INFORMATICA

Progettazione e sviluppo dell'orchestratore
dell'architettura ETSI MEC nel simulatore
OMNeT ++

Relatore: Prof. Gian Paolo Rossi

Correlatore: Dott. Christian Quadri

Tesi di Laurea di:

Leonardo Menti

Matricola N. 908947

ANNO ACCADEMICO 2019/2020

Ringraziamenti

Grazie al mio relatore Prof. Gian Paolo Rossi e al mio correlatore Dott. Christian Quadri per il supporto che mi hanno fornito durante il lavoro di tesi, per la loro disponibilità e flessibilità che ci hanno permesso di svolgere questo progetto durante questo particolare anno.

Grazie ai miei amici e compagni di corso per ogni singola opportunità di confronto in ambito accademico, sociale e lavorativo.

Grazie infine ai miei genitori e a mio fratello per ogni consiglio che hanno sempre saputo fornirmi.

Indice

Ringraziamenti	i
Indice	ii
Elenco delle figure	v
Introduzione	1
1 Multi-access Edge Computing	4
1.1 Multi-access Edge Computing framework	4
1.2 Architettura di riferimento	7
1.2.1 Architettura di riferimento generica	7
1.2.2 Architettura di riferimento in NFV	8
1.2.3 NFV - Network Function Virtualization	9
1.3 Entità principali	10
1.3.1 MEC host	10
1.3.2 MEC platform	10
1.3.3 MEC applications	10
1.3.4 MEC orchestrator	10
1.3.5 Altre entità	11
2 OMNeT ++	12
2.1 OMNeT ++ IDE	13
2.2 Il linguaggio NED	14
2.2.1 Caratteristiche del linguaggio	14
2.2.2 The Network	16
2.2.3 Simple Modules	17
2.2.4 Compound Modules	18

2.2.5	Channels	20
2.2.6	Parametri	20
2.2.7	Gates	22
2.2.8	Submodules	23
2.2.9	Connections	24
2.2.10	Metadata Annotations	25
2.2.11	Packages	26
2.3	Simple Modules	27
2.3.1	Overview	28
2.3.2	Ciclo di vita	29
2.3.3	Inviare e ricevere messaggi	30
2.3.4	Parametri dei moduli	32
2.3.5	Gates	32
2.3.6	Gerarchia del modulo	33
2.3.7	Creazione dinamica di moduli	33
2.4	Messaggi e pacchetti	34
2.4.1	cMessage	34
2.4.2	Self-Messages	34
2.4.3	cPacket	35
2.4.4	Definire messaggi e pacchetti	35
2.5	Configurazione della simulazione	36
2.6	Esecuzione della simulazione	37
2.7	Analisi dei risultati	38
2.7.1	Libreria di simulazione	38
2.7.2	Signals	38
3	INET	39
3.1	Componenti della rete	41
3.2	Livello fisico	43
3.2.1	Wired Network	43
3.2.2	Wireless Network	44
3.3	Livello data-link	45
3.3.1	Interfacce Wired	45
3.3.2	Interfacce Wireless	45
3.4	Livello di rete	46
3.5	Livello di trasporto	47
3.5.1	TCP	47
3.5.2	UDP	48
3.5.3	SCTP	48
3.5.4	RTP	48

3.6	Livello di applicazione	49
3.6.1	Applicazioni TCP	49
3.6.2	Applicazioni UDP	49
4	Progetto	50
4.1	Componenti MEC	51
4.1.1	MEC Orchestrator	52
4.1.2	MEC Host e MEC Platform	52
4.1.3	MEC User	53
4.1.4	Sequence Diagram della simulazione	53
4.1.5	Collegamenti	54
4.2	La rete	56
4.2.1	Types	57
4.2.2	Submodules	58
4.3	StandardHost	59
4.4	MecControlApp	61
4.4.1	Modello	61
4.4.2	Implementazione	61
4.5	MecOrchestratorApp	65
4.5.1	Modello	65
4.5.2	Implementazione	66
4.6	MecPlatformApp	68
4.6.1	Modello	68
4.6.2	Implementazione	69
4.7	MecUser	71
4.7.1	Modello	71
4.7.2	Implementazione	71
4.8	Messaggi	73
4.9	Configurazione e Simulazione	75
	Conclusioni	78
	Bibliografia	80

Elenco delle figure

1	Edge Computing	2
2	Multi-access Edge Computing framework	5
3	Architettura di riferimento generica MEC	7
4	Architettura di riferimento MEC in NFV	8
5	Approccio tradizionale vs approccio NFV	9
6	OMNeT ++ IDE	13
7	Metadata annotations	15
8	Esempio di network	16
9	Simple Module	17
10	Utilizzo dell'opzione <code>@namespace</code>	17
11	Subclassing	18
12	Subclassing - compound modules	19
13	Assegnamento parametro tramite <code>submodules</code>	20
14	Assegnamento parametro tramite <code>parameters</code>	21
15	Parametri XML	22
16	Parametric Submodules Types	23
17	Multiple connections	24
18	Metadata Annotations	25
19	Ereditarietà delle classi	27
20	Simple Module	28
21	Handle Message	31
22	Funzioni <code>gate()</code> e <code>gateHalf()</code>	32
23	Creazione ed eliminazione dinamica	33
24	Costruttore della classe <code>cMessage</code>	34
25	Message definitions	35
26	File <code>omnetpp.ini</code>	36
27	Esecuzione della simulazione	37
28	NED File <code>src/inet/node/inet/Router.ned</code>	42

29	Wired Network	43
30	Wireless Network	44
31	Protocolli di livello di trasporto in StandardHost	47
32	File MEC.ned	51
33	Sequence Diagram della simulazione	54
34	Edge Computing	55
35	Channel	56
36	Channel Backbone	57
37	MEC.ned - submodules	58
38	Subclassing di StandardHost	59
39	ApplicationLayerNodeBase	60
40	IMecControlApp.ned	61
41	MecControlApp.h - parte 1	62
42	MecControlApp.h - parte 2	64
43	Subclassing di MecControlApp	65
44	MecOrchestratorApp.ned	66
45	Classe Host	66
46	MecOrchestratorApp.h	67
47	MecPlatformApp.ned	68
48	MecPlatformApp.h"	69
49	MecControlMessages.msg - parte 1	73
50	MecControlMessages.msg - parte 2	74
51	omnetpp.ini	75
52	Rete della simulazione	76
53	Console della simulazione	77

Introduzione

Il continuo sviluppo economico e tecnologico della società ha portato alla creazione di determinate situazioni in cui è comodo, ma molto spesso anche necessario, una comunicazione di dispositivi elettronici tra di loro a velocità molto elevata. Al giorno d'oggi la rete internet non ospita più soltanto dispositivi puramente ideati per la comunicazione, quali laptop, smartphones oppure calcolatori di più grande dimensione. La rete internet è composta da moltissimi oggetti che tutti i giorni utilizziamo, dando vita alla così detta *Internet of Things (IoT)*. Gli oggetti si rendono partecipi al dialogo sulla rete, comunicano dati su loro stessi e ricevono allo stesso tempo informazioni da parte di altri. Acquisiscono così intelligenza ed un ruolo attivo nella nostra vita, come una sveglia che suona prima in caso di traffico o come un'automobile che guida in modo autonomo. Parlando di automobili e di guida autonoma possiamo capire quanto sia importante il discorso precedente, ossia la necessità di comunicare con altri dispositivi in maniera estremamente veloce. Questo purtroppo non è sempre possibile, infatti la velocità di navigazione può dipendere dalle caratteristiche della rete a cui siamo agganciati. Inoltre un utilizzatore in movimento che sfrutta una rete cellulare, come appunto una vettura, può rendere ancora più complicato garantire una comunicazione veloce e stabile.

Nasce per risolvere questo tipo di problema l' *Edge Computing*: un sistema distribuito per effettuare elaborazioni di dati il più vicino possibile all'utilizzatore. I vantaggi principali sono una riduzione consistente del traffico sulla rete ed una minore latenza di comunicazione. Dall'altra parte dispone di una minore potenza di calcolo rispetto ad una soluzione cloud e la difficoltà principale da risolvere è il coordinamento di una struttura distribuita, quindi la gestione di problemi come la consistenza ed integrità dei dati oppure come la migrazione di servizi da un server ad un altro. Al contrario delle architetture di rete odierne, i sistemi di tipo Edge Computing possono essere integrati anche in architetture come quella del 5G. Il 5G infatti mira a supportare nativamente l'Edge Computing.

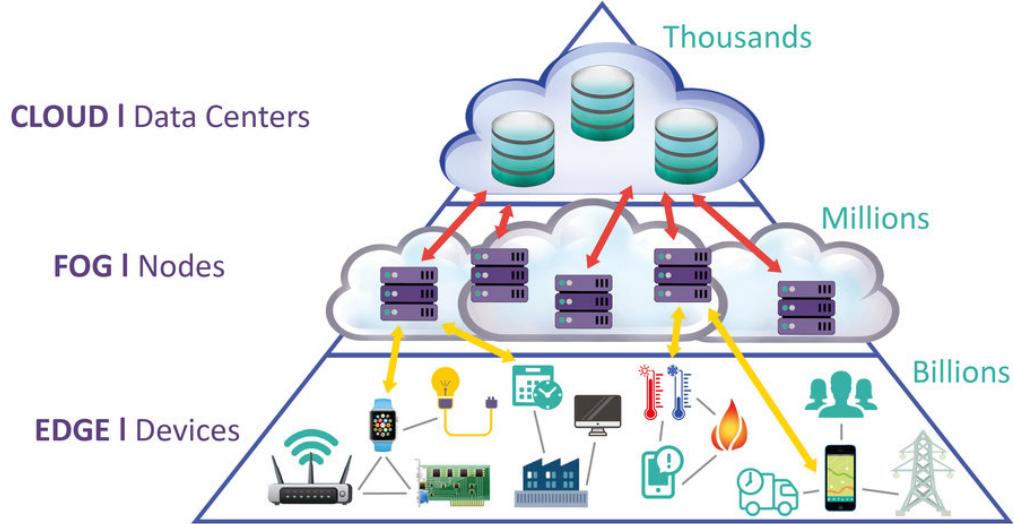


Figura 1: Edge Computing

Il termine Edge Computing viene spesso accostato al termine *Fog Computing*, per indicare bene o male lo stesso approccio di elaborazione dei dati. In Figura 1 viene illustrato una rappresentazione piramidale del sistema Edge, dove troviamo i devices ai margini della rete, seguiti dai nodi di elaborazione fisicamente vicini a loro. La base della piramide è composta da un numero elevato di nodi, come mostrato in figura. La punta della piramide rappresenta i data centers di gestione, che sono molto meno numerosi rispetto ai nodi nella parte bassa e che potenzialmente possono essere distanti dal resto del sistema.

Sulla base dell’Edge Computing, l’*European Telecommunications Standards Institute (ETSI)* ha definito l’architettura di rete *Multi-access Edge Computing (MEC)*. L’obiettivo principale di questa tesi è quello di ricreare l’infrastruttura ETSI MEC all’interno di un ambiente di simulazione, ovvero OMNeT ++, progettando e sviluppando lo scenario di lavoro del sistema. L’attenzione è posta su tutti gli aspetti di gestione dell’architettura, in particolare sul componente *MEC orchestrator*, entità che possiede una visione globale dell’infrastruttura. Partendo dallo studio dettagliato dell’architettura ETSI MEC, è stato ricreato un sistema per rappresentarla all’interno dell’ambiente di simulazione, adottando una serie di semplificazioni in ottica di porre l’enfasi sugli aspetti di controllo e sul componente *MEC orchestrator*, sui messaggi che scambia con le altre entità e sulle sue strutture dati di controllo. Le entità di rete sono state realizzate servendosi del framework INET, una libreria open source per il simulatore OMNeT ++, che fornisce protocolli, componenti ed altre strutture per la simulazione delle reti di comunicazione. Dopo l’architettura ETSI MEC ed il simulatore OMNeT ++, lo studio del framework INET è stato il punto di partenza

per ricreare le entità MEC all'interno del simulatore, comprendendo quali componenti messi a disposizione dal framework fossero più idonei per la realizzazione del progetto, utilizzandoli, estendendoli e collegandoli nella maniera più opportuna per ricreare il sistema MEC.

L'elaborato affronta prima di tutto il framework ETSI MEC; vengono approfonditi i suoi componenti, le loro caratteristiche ed i ruoli di ognuno all'interno dell'architettura. Successivamente viene presentato l'ambiente di simulazione OMNeT++, fornendo una breve documentazione in relazione alle funzionalità utilizzate durante la progettazione e simulazione del sistema. In seguito viene illustrato il framework INET, anche qui analizzando i componenti che sono stati utili alla realizzazione del progetto, necessari da assimilare per comprendere a pieno il lavoro che è stato svolto a partire da questi, ossia la realizzazione delle entità del framework ETSI MEC.

Capitolo 1

Multi-access Edge Computing

Multi-access Edge Computing (MEC) è un’architettura di rete definita dall’*European Telecommunications Standards Institute (ETSI)* [1]. Garantisce servizi IT ai margini della rete, quindi fornisce applicazioni ed esegue task vicino all’utilizzatore finale, fornendo latenze molto basse tali da rendere possibili le applicazioni quali realtà aumentata/virtuale, automotive e industria 4.0.

1.1 Multi-access Edge Computing framework

In Figura 2 vengono illustrate le principali entità coinvolte nell’architettura, che compongono il **Multi-access Edge Computing framework** [2]. Queste entità possono essere raggruppate in: **system level**, **host level** e **network level**, anche se per descrivere in maniera completa il framework è possibile limitarsi all’analisi del livello di sistema e del livello host.

- Il **system level** è composto dalle entità che sostanzialmente hanno una visione globale sul sistema MEC, svolgendo funzioni come il monitoraggio dei componenti oppure come il dialogo con gli utenti finali per l’attivazione di servizi e/o applicazioni. Il componente più importante di questo livello è il MEC orchestrator.
- L’**host level** contiene le entità che effettivamente offrono il servizio agli utilizzatori finali, ossia le MEC applications, che sono eseguite all’interno di un ambiente virtuale. Il livello host contiene sia i componenti di controllo che gli effettivi componenti che svolgono le richieste. Le principali entità di questo livello sono il MEC host, la MEC platform, le MEC applications e l’NFVI (Network Function Virtualization Infrastructure).

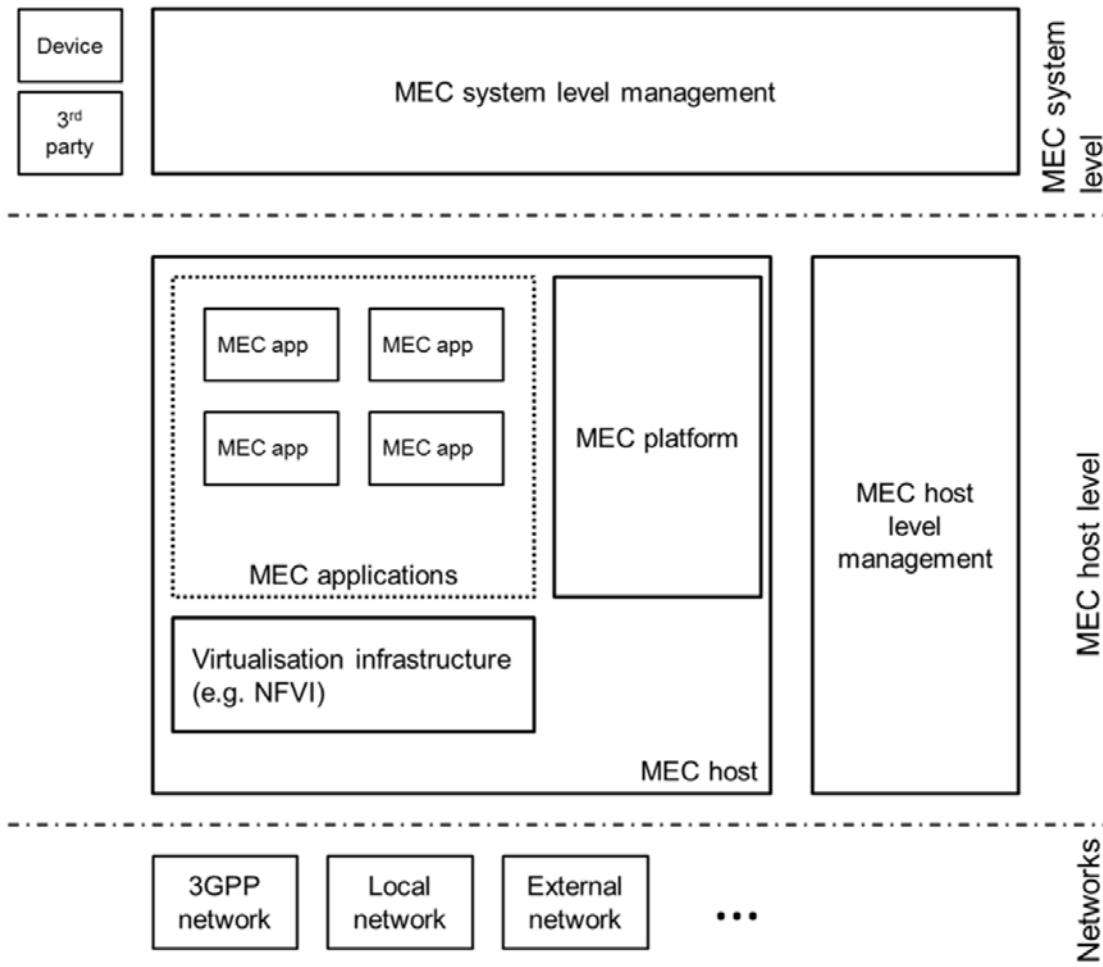


Figura 2: Multi-access Edge Computing framework

Di seguito una breve descrizione dei principali componenti:

- Il **MEC host** è un'entità che contiene una MEC platform ed una infrastruttura virtuale (NFVI), che provvede a fornire computazione, storage e risorse di rete per le MEC applications che deve ospitare.
- La **MEC platform** consiste nell'entità che fornisce le funzionalità per eseguire le MEC applications in un ambiente virtuale e permette loro di fornire e richiedere servizi MEC.
- Le **MEC applications** sono entità software istanziate nell'ambiente virtuale all'interno del MEC host, sulla base delle configurazioni e richieste validate dal MEC orchestrator. È importante distinguere quello che si intende per MEC

applications e **MEC services**. Un utente richiede al sistema l’attivazione di uno o più servizi tramite una device application che dialoga con il livello di sistema. Un servizio MEC è composto da una o più applicazioni. Le applicazioni possono essere istanziate in diverse posizioni geograficamente distanti tra di loro, ossia ospitate in diverse MEC platforms. Tutto questo viene gestito dall’orchestratore, sulla base di criteri come lo stato delle risorse delle varie MEC platforms oppure come particolari vincoli di distanza tra utente ed applicazione. Ovviamente la gestione delle applicazioni da parte del MEC orchestrator può complicarsi sulla base appunto di risorse, distanza e priorità, ma tutta la gestione del processo rimane trasparente all’utilizzatore finale, che semplicemente richiede ed utilizza un servizio MEC.

- Il **MEC orchestrator** è il cuore dell’architettura e possiede una visione globale del intero sistema MEC. Svolge funzioni come l’istanziazione dei servizi, la gestione del loro ciclo di vita ed il monitoraggio delle MEC platform.

Le principali entità dell’architettura MEC sono descritte in modo completo e dettagliato nella Sezione 1.3.

1.2 Architettura di riferimento

L’architettura di riferimento illustra la entità principali che compongono il sistema MEC ed i collegamenti tra di loro. ETSI ha definito due architetture di riferimento per MEC: l’architettura generica e l’architettura in NFV.

1.2.1 Architettura di riferimento generica

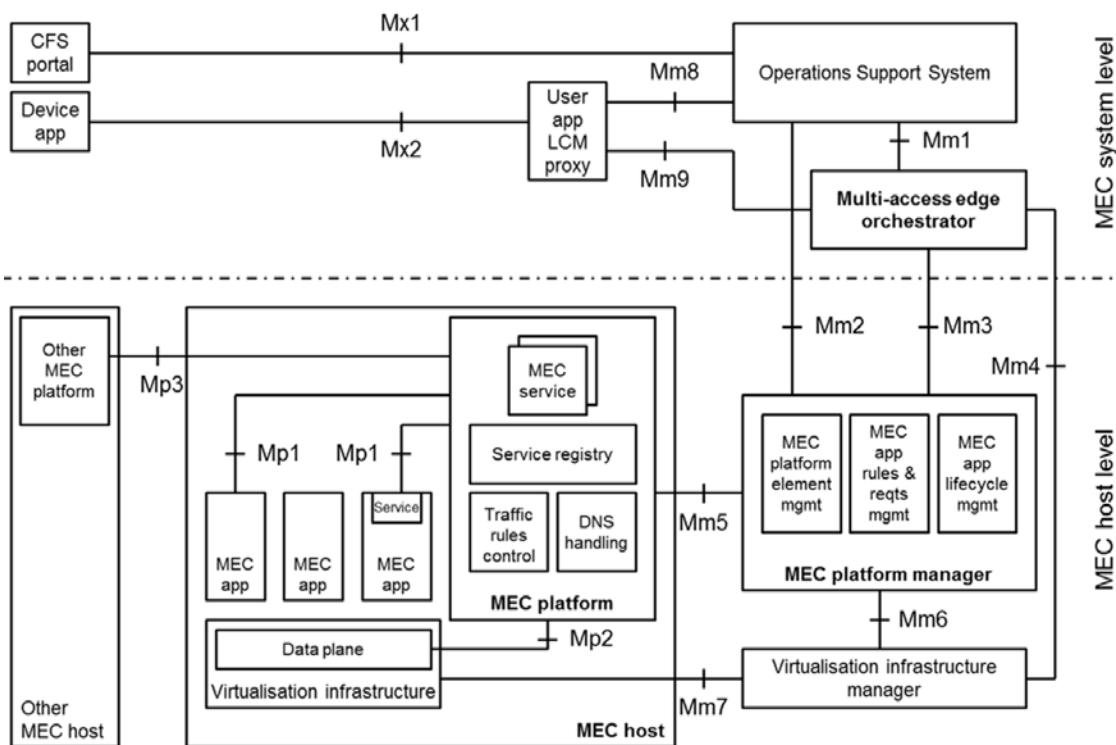


Figura 3: Architettura di riferimento generica MEC

In Figura 3 viene illustrata l’architettura di riferimento generica dell’architettura MEC. Ci sono tre tipologie di collegamenti tra le entità:

- Mp: collegamenti riguardanti le funzionalità della MEC platform
- Mm: collegamenti di gestione
- Mx: collegamenti con entità esterne

1.2.2 Architettura di riferimento in NFV

L’architettura MEC è stata inizialmente concepita in maniera tale che fosse stato possibile realizzare diverse opzioni di sviluppo del sistema stesso. Di seguito viene illustrata una variante dell’architettura MEC che permette di istanziare MEC applications e VNF (Virtualized Network Functions) nella stessa infrastruttura virtuale (Figura 4)

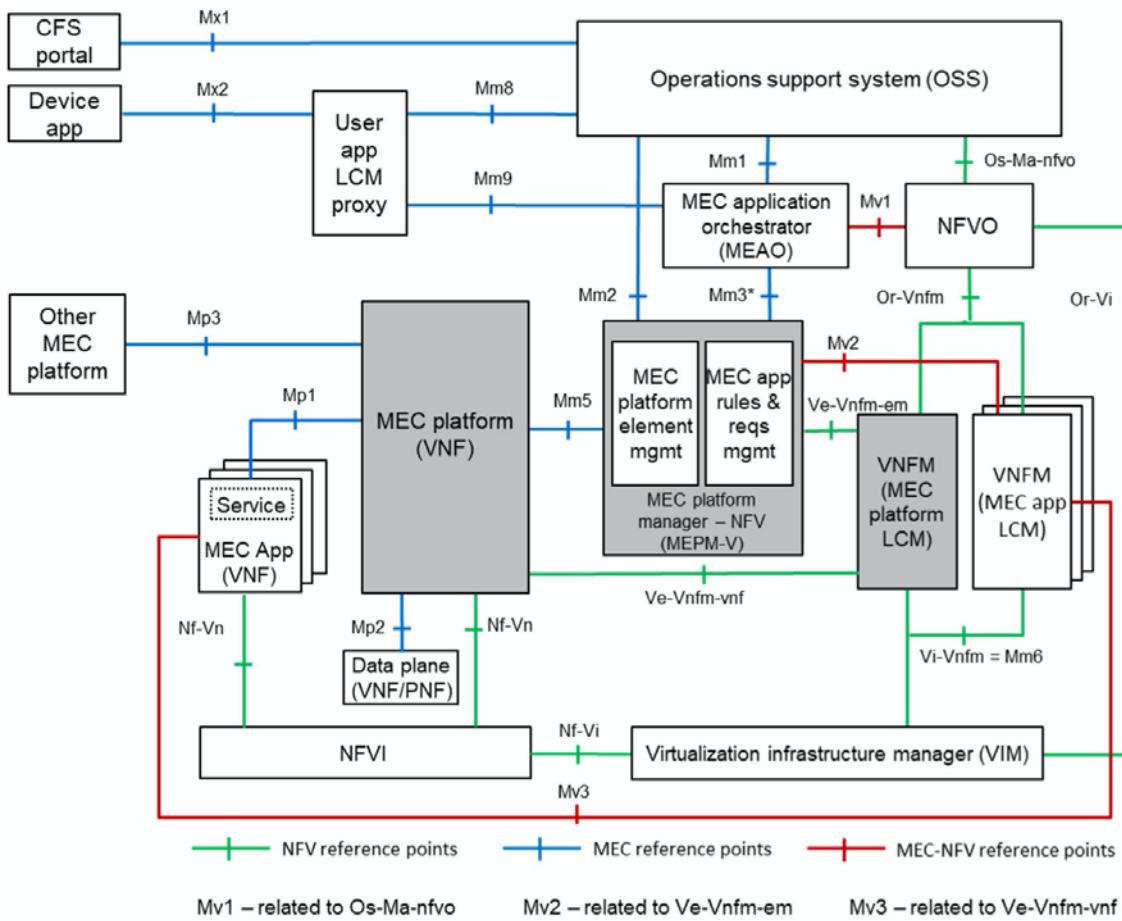


Figura 4: Architettura di riferimento MEC in NFV

L’architettura di riferimento in NFV mantiene la struttura base dell’architettura generica, modificano alcuni nomi ed aggiungendo delle componenti. La MEC platform e le MEC applications sono sviluppate come virtualized network functions, mentre l’infrastruttura virtualizzata diventa NFVI, gestita da VIM. Il concetto di NFV (Network Function Virtualization) viene illustrato in maniera più approfondita nella Sezione 1.2.3.

1.2.3 NFV - Network Function Virtualization

Grazie alle tecniche di virtualizzazione, ormai uno standard tecnologico, un calcolatore è in grado di eseguire al suo interno qualsiasi tipo di software sotto forma di macchina virtuale (Virtual Machine). Il termine **Network Function Virtualization (NFV)** nell'ambito delle reti di telecomunicazione indica appunto l'approccio che mira a virtualizzare nodi che compongono una rete, ad esempio switches, routers oppure firewalls. Come ogni processo di virtualizzazione si cerca di trasformare un'entità hardware in un'entità software.

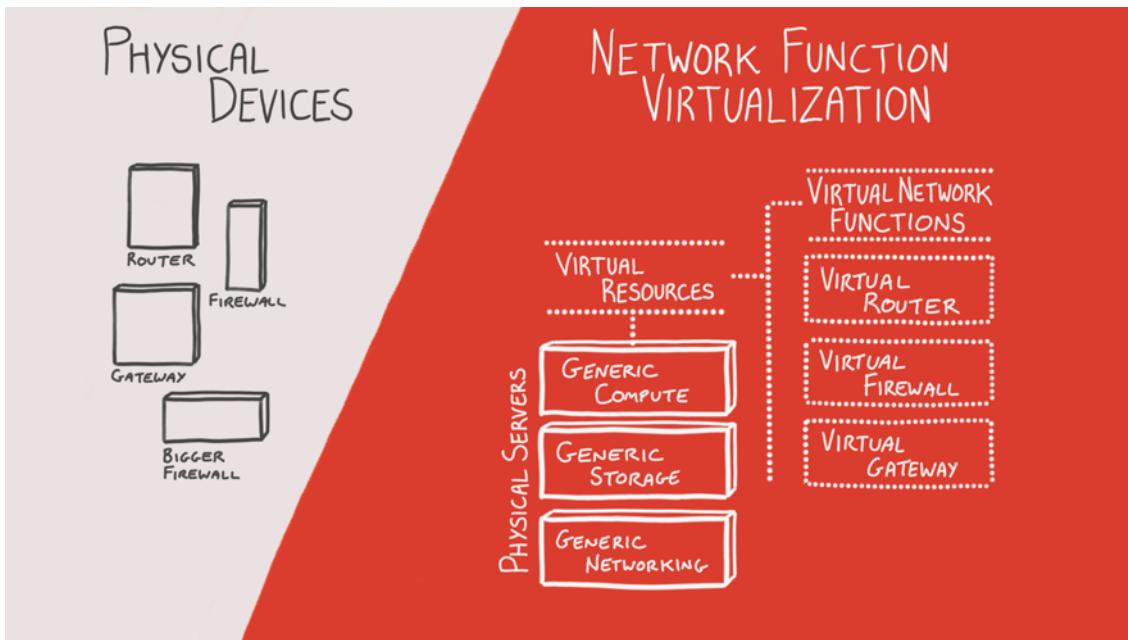


Figura 5: Approccio tradizionale vs approccio NFV

Invece di utilizzare un dispositivo hardware per ogni elemento della rete, la funzionalità del dispositivo viene virtualizzata su un server standard ad alta capacità oppure anche su un'infrastruttura cloud. Un vantaggio derivante dall'approccio NFV è sicuramente una riduzione del costo dell'hardware specifico. Inoltre il guadagno è soprattutto in termini di flessibilità: trattandosi di puro software funzionante su una piattaforma hardware generica, è possibile modificare i componenti su domanda e secondo le necessità. Per di più, questa operazione è realizzabile da remoto, senza intervento di tecnici specialisti sul campo. In Figura 5 un'illustrazione esemplificativa del cambiamento dall'approccio di rete tradizionale all'approccio di rete NFV.

1.3 Entità principali

1.3.1 MEC host

Il **MEC host** è un'entità che contiene una MEC platform ed una infrastruttura virtualizzata che fornisce computazione, storage e risorse di network per ospitare MEC applications. L'infrastruttura virtualizzata include un data plane che esegue le regole di traffico ricevute dalla MEC platform ed instrada il traffico tra applicazioni, servizi, etc.

1.3.2 MEC platform

La **MEC platform** è responsabile di fornire un ambiente in cui le MEC applications possono offrire servizi agli utenti finali. La MEC platform riceve istruzioni dal MEC platform manager e può offrire lei stessa dei servizi MEC. La MEC platform è inoltre responsabile di auto-monitorare la proprie risorse hardware e comunicarle periodicamente al MEC orchestrator.

1.3.3 MEC applications

Le **MEC applications** vengono eseguite come virtual machines all'interno di un ambiente virtuale fornito dal MEC host. Interagiscono con la MEC platform per fornire e/o ricevere servizi. Le applicazioni fanno parte di un servizio che viene richiesto dall'utente al sistema e possono avere un certo numero di regole e requisiti associati ad esse, ad esempio parametri come la larghezza di banda oppure il tempo di processamento. Questi requisiti verranno validati dal MEC orchestrator e potranno essere assegnati di default nel caso in cui non siano specificati.

1.3.4 MEC orchestrator

Il **MEC orchestrator** è la funzionalità principale del MEC system level. È responsabile di mantenere una visione globale del sistema, in particolare dello stato dei vari MEC host e dei servizi MEC disponibili. Inoltre deve convalidare le regole ed i requisiti per le applicazioni, adeguando gli stessi per conformarsi alle politiche dell'operatore. L'orchestratore deve selezionare i MEC host più appropriati per l'istanziazione delle MEC applications, in base a vincoli quali latenza, risorse e servizi disponibili. Infine gestisce l'attivazione e la chiusura delle applicazioni e, se necessario, la migrazione di applicazioni, se la funzionalità è supportata.

1.3.5 Altre entità

- L'**Operations support system (OSS)** riceve dal CFS portal e dalle applicazioni dei device richieste riguardanti l'istanziazione e la terminazione di servizi MEC. Dialoga con il MEC orchestrator che procede ad attivare o terminare le applicazioni.
- Lo **User application lifecycle management proxy** è un'entità del sistema che si occupa di gestire le richieste provenienti dagli utenti. Dialoga con l'OSS ed il MEC orchestrator per il processamento di queste richieste.
- Il **MEC platform manager** è responsabile del ciclo di vita delle MEC applications che vengono eseguite all'interno del MEC platform. Il MEC platform manager dialoga con il MEC orchestrator per istruire la MEC platform sulle funzioni da svolgere, per esempio l'istanziazione di una applicazione con determinati requisiti convalidati dall'orchestratore.
- La **Virtualization infrastructure manager (VIM)** è responsabile di gestire le risorse virtuali (compute, storage, networking) dell'infrastruttura di virtualizzazione. La VIM deve preparare e configurare l'infrastruttura di virtualizzazione ad eseguire un immagine software. VIM inoltre monitora costantemente le risorse virtuali, collezionando dati di performance e di fault.
- Le **Device applications**, come introdotto precedentemente, sono delle applicazioni che risiedono nei device degli utenti e che possono interagire con il sistema MEC attraverso uno user application lifecycle management proxy.
- Il **Customer facing service portal** si occupa di gestire le richieste di clienti di terze parti, ad esempio imprese commerciali. Queste possono selezionare e richiedere un set di MEC applications che soddisfano le loro esigenze e di ricevere informazioni sul livello di servizio dalle applicazioni fornite.

Altre entità specifiche nell'architettura MEC in NFV

- **NFVO**: responsabile dell'istanziazione dei servizi
- **VNFM (MEC Platform LCM)**: responsabile del ciclo di vita della MEC platform
- **VNFM (MEC App LCM)**: responsabile del ciclo di vita delle MEC apps

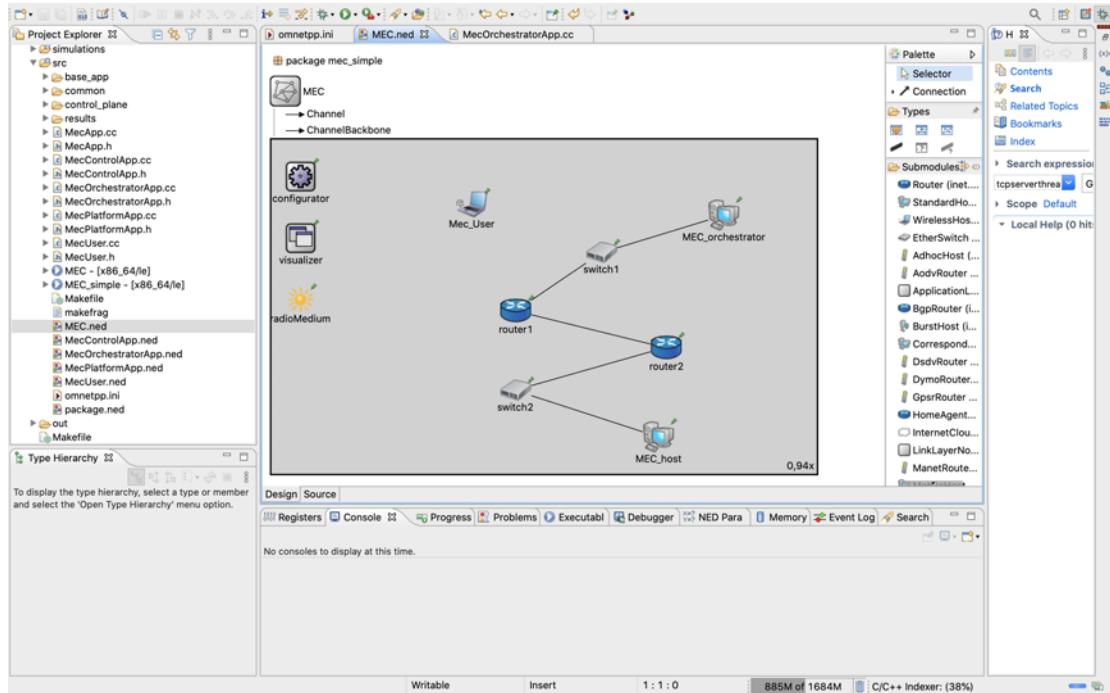
Capitolo 2

OMNeT ++

OMNeT ++ è una libreria e framework di simulazione C++ estensibile, modulare e basata su componenti, principalmente utilizzato per la creazione di simulatori di rete [3]. Il termine *rete* è inteso in un senso più ampio rispetto a reti di comunicazione cablate e wireless. OMNeT ++ offre un IDE basato su Eclipse, un ambiente di runtime grafico ed una serie di altri strumenti. Sono disponibili estensioni per la simulazione in tempo reale, l'emulazione di rete, l'integrazione del database e molte altre funzioni. Sebbene OMNeT ++ non sia un simulatore di rete in sé, ha guadagnato una popolarità diffusa come piattaforma di simulazione di rete nella comunità scientifica e in contesti industriali, creando una vasta comunità di utenti. I componenti base di OMNeT ++ sono i moduli e sono programmati in C++, quindi assemblati in componenti e modelli più grandi utilizzando un linguaggio di alto livello (NED).

Componenti principali di OMNeT ++

- Libreria del kernel di simulazione (C++)
- Il linguaggio di descrizione della topologia NED
- IDE di simulazione basato sulla piattaforma Eclipse
- GUI di runtime di simulazione interattiva (Qtenv)
- Interfaccia della riga di comando per l'esecuzione della simulazione (Cmdenv)
- Utilities (makefile creation tool, etc.)
- Documentazione, simulazioni di esempio, ecc.

**Figura 6:** OMNeT ++ IDE

2.1 OMNeT ++ IDE

L’ambiente di sviluppo OMNeT ++ è basato sulla piattaforma Eclipse, che estende con nuovi editor, viste, wizards e funzionalità aggiuntive. OMNeT ++ aggiunge funzionalità per creare e configurare modelli, tramite i file NED e ini, eseguire esecuzioni batch ed analizzare i risultati della simulazione. Eclipse fornisce, tramite vari plug-in open-source e commerciali, editing C ++, integrazione GIT ed altre funzionalità opzionali, come la modellazione UML, l’ integrazione bugtracker oppure l’accesso al database. In Figura 6 è mostrato l’ambiente di lavoro di OMNeT ++.

2.2 Il linguaggio NED

Il linguaggio NED è ciò che permette all’utente di definire la struttura del modello di simulazione. NED è l’acronimo di *Network Description*. Tramite il linguaggio NED l’utente può definire *simple modules* e connetterli tra di loro, oppure assemlarli per formare *compound modules*. Il linguaggio permette inoltre di definire *channels* e *networks*; queste ultime sono una particolare tipologia di moduli composti. I files NED sono salvati con l’estensione `.ned`.

2.2.1 Caratteristiche del linguaggio

- **Hierarchical:** qualsiasi modulo che sarebbe troppo complesso come una singola entità può essere suddiviso in moduli più piccoli e utilizzato come modulo composto.
- **Component-Based:** moduli semplici e moduli composti sono intrinsecamente riutilizzabili, il che non solo riduce la copia del codice, ma, cosa più importante, consente l’esistenza di librerie di componenti, come INET Framework, MiXiM, Castalia, ecc.
- **Interfaces:** è possibile definire delle interfacce sia per i moduli sia per i canali. Le interfacce possono essere utilizzate come placeholder dove normalmente verrebbe utilizzato un modulo o un tipo di canale. I moduli concreti devono implementare l’interfaccia che sostituiscono. Ad esempio, dato un tipo di modulo composto denominato `MobileHost`, che contiene un sottomodulo di mobilità di tipo `IMobility` (dove `IMobility` è un’interfaccia), il tipo effettivo di mobilità può essere scelto dai moduli che implementano `IMobility` (`RandomWalkMobility`, `TurtleMobility`, ecc.).
- **Inheritance:** moduli e channels possono essere estendibili via subclassing. I moduli e i canali derivati possono aggiungere nuovi parametri, gates e, nel caso di moduli composti, nuovi sottomoduli e connessioni. È possibile impostare i parametri esistenti su un valore specifico e anche impostare la dimensione di un gate vector. Per esempio un modulo composto `WebClientHost` può derivare da un modulo composto `BaseHost`, aggiungendo un sottomodulo `WebClientApp` e collegandolo al sottomodulo `TCP` ereditato.
- **Packages:** il linguaggio NED presenta una struttura dei packages simile a Java, per ridurre il rischio di conflitti di nome tra modelli diversi. È stato introdotto anche `NEDPATH` (simile a `CLASSPATH` di Java) per semplificare la specifica delle dipendenze tra i modelli di simulazione.

```

package mec_simple;

simple MecPlatformApp like IMecControlApp
{
    parameters:

        @class(MecPlatformApp);

        int listeningPort = default(1000);

        string orchestratorAddress = default("MEC_orchestrator");
        int orchestratorPort = default(1000);
        double startTime @unit(s) = default(0s);

    gates:
        input socketIn @labels(Server/up);
        output socketOut @labels(Server/down);
}

```

Figura 7: Metadata annotations

- **Inner Types:** i tipi di channels e i tipi di moduli utilizzati localmente da un modulo composto possono essere definiti all'interno del modulo composto, al fine di ridurre il namespace pollution.
- **Metadata Annotations:** è possibile annotare moduli, canali, parametri, gates e sottomoduli aggiungendo proprietà, utilizzando il simbolo `@`. I metadati/proprietà non vengono utilizzati direttamente dal kernel di simulazione, ma possono trasportare informazioni aggiuntive per vari strumenti, l'ambiente di runtime o anche per altri moduli nel modello. Ad esempio la rappresentazione grafica di un modulo (icona, ecc.), la stringa di richiesta oppure l'unità di misura di un parametro (milliwatt, ecc.) sono specificate come annotazioni di metadati. In Figura 7 alcuni esempi di utilizzo delle proprietà.

2.2.2 The Network

```
network Tictoc1
{
    @display("bgb=254,200");
    submodules:
        tic: Txc1 {
            @display("p=58,133");
        }
        toc: Txc1 {
            @display("p=191,34");
        }
    connections:
        tic.out --> { delay = 100ms; } --> toc.in;
        tic.in <-- { delay = 100ms; } <-- toc.out;
}
```

Figura 8: Esempio di network

Una rete viene definita con il prefisso **network** all'interno di un file NED. Una rete è un modulo composto che contiene lo scenario di simulazione desiderato. In Figura 8 viene definita una rete di nome **Tictoc1** che contiene due moduli di tipo **Txc1**, chiamati rispettivamente **tic** e **toc**. I due moduli sono collegati da due canali con **delay** di 100ms ognuno e la rete rappresenta la comunicazione tra due entità. È possibile definire più di una rete all'interno del medesimo file NED, così come è possibile servirsi di diversi file NED. In fase di configurazione è necessario specificare quale rete si vuole simulare, solitamente tramite l'opzione **network** all'interno del file di configurazione (di default il file **omnetpp.ini**). La configurazione della rete, così come il concetto di modulo, connections e messaggi saranno approfonditi nelle prossime sezioni.

2.2.3 Simple Modules

```
simple Queue
{
    parameters:
        int capacity;
        @display("i=block/queue");
    gates:
        input in;
        output out;
}
```

Figura 9: Simple Module

I **simple modules** sono la parte attiva della simulazione. Vengono definiti tramite la keyword **simple**. Sia i parametri che i gates sono opzionali, quindi possono non essere specificati nel caso in cui il modulo non contenga parametri o gates. Inoltre anche la keyword **parameters** è opzionale, sempre le nel caso che il modulo non abbia dei parametri. Da notare che la definizione NED del modulo non contiene alcun codice per descrivere il comportamento che questo deve avere. Quella parte è espressa in C++. Di default OMNeT ++ cerca la classe C++ con il medesimo nome del simple module. In alternativa è possibile specificare il nome della classe C++ attraverso l'opzione **@class**, potendo anche specificare il namespace relativo, come mostrato in Figura 9. Se molti moduli hanno le rispettive classi C++ all'interno dello stesso namespace allora potrebbe risultare comodo utilizzare l'opzione **@namespace**. In Figura 10 le classi C++ saranno **mylib::App** e **mylib::Router**.

```
@namespace(mylib);

simple App {
    ...
}

simple Router {
    ...
}

simple Queue {
    ...
}
```

Figura 10: Utilizzo dell'opzione **@namespace**

I simple modules sono estendibili tramite subclassing. Le motivazioni per estendere un simple module possono essere molteplici: settare o aggiungere parametri e gates, oppure usare una classe C++ differente (Figura 11). Di default una sottoclasse di un simple module utilizzerà la classe C++ del padre, quindi nel caso in cui si voglia utilizzare un'altra classe bisogna utilizzare l'opzione `@class`.

```
simple Queue
{
    int capacity;
    ...
}

simple BoundedQueue extends Queue
{
    capacity = 10;
}
```

Figura 11: Subclassing

2.2.4 Compound Modules

Un **compound module** raggruppa altri moduli in un'unità più grande. I moduli composti, così come i moduli semplici, possono avere parametri e gates, ma a loro non è associato nessun comportamento attivo. Un compound module viene definito attraverso la keyword **module**. Un modulo composto può contenere diverse sezioni, tutte opzionali.

- Nella sezione **types** si possono definire moduli e/o canali usati localmente al modulo composto come inner types.
- Nella sezione **submodules** sono definiti i moduli contenuti all'interno del compound module.
- Nella sezione **connections** vengono definite le connessioni tra i vari submodules, se necessario si possono utilizzare costrutti di programmazione(loop, conditional) per specificare le connessioni.
- Le sezioni **parameters** e **gates** svolgono le stesse funzionalità delle sezioni dei simple modules.

Anche i moduli composti possono essere estesi tramite subclassing. L'ereditarietà può essere sfruttata per aggiungere, oltre a parametri e gates, anche nuovi submodules, connections e types. In Figura 12 viene esteso il modulo `WirelessHostBase` creando la sottoclass `WirelessHost`, in cui viene aggiunto un sottomodulo `webAgent`, di tipo `WebAgent`, che viene collegato al sottomodulo `tcp`.

```

module WirelessHostBase
{
    gates:
        input radioIn;
    submodules:
        tcp: TCP;
        ip: IP;
        wlan: IEEE80211;
    connections:
        tcp.ipOut --> ip.tcpIn;
        tcp.ipIn <-- ip.tcpOut;
        ip.nicOut++ --> wlan.ipIn;
        ip.nicIn++ <-- wlan.ipOut;
        wlan.radioIn <-- radioIn;
}

module WirelessHost extends WirelessHostBase
{
    submodules:
        webAgent: WebAgent;
    connections:
        webAgent.tcpOut --> tcp.appIn++;
        webAgent.tcpIn <-- tcp.appOut++;
}

```

Figura 12: Subclassing - compound modules

2.2.5 Channels

I **channels** contengono i parametri ed il comportamento associato alle connessioni tra vari moduli. I channels sono dei simple modules e quindi possiedono delle classi C++ che ne determinano il comportamento. La differenza principale con i simple modules sta nel fatto che difficilmente si scriveranno classi C++ personalizzate per i propri canali, ma invece, estendendo una delle tre classi principali di channels, si andranno ad ereditare le classi C++ relative. Di seguito una descrizione dei tre tipi principali di channels, dove **ned** indica il nome del package:

- **ned.IdealChannel**: non possiede parametri e rappresenta quindi un canale con delay nullo e senza nessun altro effetto collaterale.
- **ned.DelayChannel**: possiede due parametri che sono **delay** e **disabled**. Delay indica il ritardo di propagazione del messaggio nel canale, mentre se disabled è settato a **true** tutti i messaggi su quel canale verranno eliminati.
- **ned.DatarateChannel**: possiede alcuni parametri in più rispetto a Delay-Channel, come **datarate**, **ber** e **per**. Datarate rappresenta il datarate del canale, mentre ber e per indicano rispettivamente il Bit Error Rate ed il Packet Error Rate e sono indicati da un double nel range [0,1].

2.2.6 Parametri

```
module Host
{
    submodules:
        ping : PingApp {
            packetLength = 128B;
        }
        ...
}
```

Figura 13: Assegnamento parametro tramite **submodules**

I parametri sono delle variabili che derivano dai moduli. Possono essere usati all'interno dei files NED oppure anche come input ai file C++. I parametri possono essere di tipo **double**, **int**, **bool**, **string** e **xml**; possono anche essere specificati come **volatile**. È possibile specificare un valore di default e, per i parametri numerici, anche un'unità di misura tramite l'opzione **@unit**.

Assegnare un valore I valori possono essere assegnati ai parametri tramite il codice NED, tramite il file di configurazione oppure anche in maniera interattiva prima dell'inizio della simulazione. Il linguaggio NED permette di assegnare dei valori ai parametri in diversi punti del codice: per esempio è possibile assegnare dei valori tramite l'ereditarietà, all'interno del blocco **submodules** (Figura 13) oppure del blocco **parameters** (Figura 14) di un compound module. Da notare in Figura 14 come l'asterisco viene usato per fare il match con qualsiasi stringa non contenente un punto, mentre la dicitura [0..49] oppure [50..] stia ad indicare un range numerico.

```

network Network
{
    parameters:
        host[*].ping.timeToLive = default(3);
        host[0..49].ping.destAddress = default(50);
        host[50..].ping.destAddress = default(0);

    submodules:
        host[100]: Host;
        ...
}

```

Figura 14: Assegnamento parametro tramite **parameters**

volatile La keyword **volatile** comporta la valutazione del parametro ogni volta che questo viene letto. Questo assume importanza se il parametro non è un valore costante, ma per esempio include un'espressione con un generatore di numeri casuali. Un altro caso in cui risulta efficiente utilizzare **volatile** è per la valutazione di un parametro con un operatore ternario all'interno del file di configurazione:

$$\text{serviceTime} = \text{simTime}() < 1000s ? 1s : 2s$$

Units È possibile dichiarare l'unità di misura di un parametro numerico utilizzando l'opzione **@unit**. Da quel momento i valori assegnati al parametro devono avere lo stessa unità di misura, oppure una compatibile. L'opzione **@unit** di un parametro non può essere aggiunta o modificata (override) via subclassing o tramite dichiarazioni di sottomoduli.

XML Parameters Molto spesso i moduli richiedono delle strutture dati complesse come input e l'utilizzo dei parametri diventa difficile per risolvere questo problema. La soluzione è inserire i dati in un file dedicato e passare il nome del file all'interno di una stringa al modulo. OMNeT ++ è predisposto per la lettura e la validazione di file XML. Sono messi a disposizione il tipo di parametro `xml` e la funzione `xml/doc()`, come mostrato in Figura 15

```

simple TrafGen {
    parameters:
        xml profile;
    gates:
        output out;
}

module Node {
    submodules:
        trafGen1 : TrafGen {
            profile = xml/doc("data.xml");
        }
        ...
}

```

Figura 15: Parametri XML

2.2.7 Gates

I gates sono i punti di connessione dei moduli. OMNeT ++ offre tre tipologie di gates: `input`, `output` e `inout`, dove l'ultimo è essenzialmente un gate di input ed uno di output uniti insieme. È possibile creare gates singoli o vettori di gates. La dimensione del vettore può essere specificata all'interno delle parentesi quadre nella dichiarazione, oppure può non essere specificata, scrivendo una coppia di parentesi vuote []. Se si sceglie di lasciare la dimensione non specificata è possibile creare i gates con l'operatore `gate++`, che espande dinamicamente il vettore. La dimensione di un vettore di gates può essere ricavata tramite l'operatore `sizeof()`. NED richiede che tutti i gates siano collegati, ma tramite l'opzione `@loose` è possibile disattivare il controllo di connessione per il relativo gate. È inoltre possibile disattivare il controllo di connessione per tutti i gates all'interno di un modulo composto, tramite la keyword `allowunconnected`.

2.2.8 Submodules

```
network Net6
{
    parameters:
        string.nodeType;
    submodules:
        node[6]: <nodeType> like INode {
            address = index;
        }
    connections:
        ...
}
```

Figura 16: Parametric Submodules Types

I submodules sono i moduli, semplici o composti, che sono contenuti all'interno di un modulo composto. I sottomoduli possono essere assegnati staticamente, ma è anche possibile specificarne la tipologia tramite una string expression; quest'ultima feature prende il nome di *Parametric Submodules Types*.

Parametric Submodules Types

L'esempio in Figura 16 crea un vettore di submodules chiamato `node` ed indica la tipologia dei moduli attraverso il parametro `string.nodeType`. La keyword `like` indica che il modulo corrispondente al valore che prenderà il parametro `nodeType` dovrà essere un modulo che implementa l'interfaccia `INode`.

Esiste inoltre un'altra sintassi, più efficace se per esempio si volessero assegnare diverse tipologie di moduli in diversi indici di un vettore di submodules. L'espressione all'interno delle parentesi angolari (`<>`) può non essere specificata:

`nic : <> like INic`

Per assegnare un tipo di modulo al sottomodulo `nic` si utilizzerà l'assegnazione tramite la keyword `typename`, che può avvenire all'interno del codice NED nella sezione `parameters` oppure all'interno del file di configurazione.

`node[*].nic.typename = "Ieee80211g"`

2.2.9 Connections

```

module Chain
  parameters:
    int count;
  submodules:
    node[count] : Node {
      gates:
        port[2];
    }
  connections allowunconnected:
    for i = 0..count-2 {
      node[i].port[1] <--> node[i+1].port[0];
    }
}

```

Figura 17: Multiple connections

Le connections permettono di far comunicare i moduli contenuti all'interno di un compound module e sono definite nella sezione **connections**. Possono unire due submodules gates oppure un submodule gate con un gate del modulo padre, quindi il compound module. I gates di input ed output sono collegati attraverso una freccia \rightarrow , mentre i gates inout sono collegati con una doppia freccia \leftrightarrow . I gates di inout possono essere collegati individualmente utilizzando la notazione **gate*i*** e **gate*o***. Se si vuole utilizzare un canale built-in (2.2.5) è possibile ometterne il nome, che sarà dedotto dal numero di parametri.

a.g++ <--> {delay = 10ms;} <--> b.g++;

Reconnect

Normalmente è un errore cercare di connettere un gate che è già connesso, ma tramite l'opzione **@reconnect** è possibile oltrepassare questo controllo. Se infatti l'opzione viene inclusa nella connessione, il network builder per prima cosa controllerà se il gate specifico è connesso. Se lo è allora lo disconnetterà per poi creare una nuova connessione. In realtà l'utilizzo dell'opzione **@reconnect** non va molto d'accordo con il significato di ereditarietà, infatti sarebbe possibile cambiare lo schema di collegamenti ereditato via subclassing.

Connessioni multiple

Quando abbiamo a che fare con vettori di submodules può risultare comodo utilizzare dei costrutti di programmazione (loop, conditional) per creare le connessioni desiderate (Figura 17).

Parametric Connection Types Così come visto con la *Parametric Submodules Types*, è possibile è possibile utilizzare un parametro come placeholder per una connection, sempre utilizzando la keyword `like`. Anche in questo caso è possibile utilizzare due sintassi: la prima che prevede l'utilizzo di una stringa come placeholder,

```
a.g ++ <--> <channelType> like IMyChannel <--> b.g ++;
```

mentre la seconda che omette la stringa ed il tipo di canale verrà assegnamento tramite `typename`:

```
a.g ++ <--> <> like IMyChannel <--> b.g ++;
```

2.2.10 Metadata Annotations

```
@namespace(foo); // file property

module Example
{
    parameters:
        @node; // module property
        @display("i=device/pc"); // module property
        int a @unit(s) = default(1); // parameter property
    gates:
        output out @loose @labels(pk); // gate properties
```

Figura 18: Metadata Annotations

Le NED properties sono informazioni aggiuntive che possono essere assegnate a moduli, parametri, gates, connections, NED files, packages e praticamente a qualsiasi cosa in NED. Le proprietà aggiungono informazioni extra agli elementi NED; alcuni esempi di annotazioni sono: `@display`, `@class`, `@namespace`, `@unit`, `@prompt`, `@loose`, `@directIn`. In Figura 18 un frammento di codice di compound module che contiene alcune proprietà. In alcune situazioni può risultare utile avere più proprietà con lo stesso nome, per esempio nel caso di più statistiche prodotte dallo stesso simple module. È possibile utilizzare gli indici delle proprietà per questo scopo, utilizzando

un numero oppure una chiave all'interno di parentesi quadre dopo il nome della proprietà. Le proprietà possono contenere o meno dei dati, che possono essere dei dati singoli, delle liste, una coppia chiave-valore o una lista di coppie chiave-valore. Infine le proprietà possono essere aggiunte e/o modificate via subclassing. Quando una proprietà viene modificata, la nuova proprietà viene unita a quella vecchia, seguendo le regole di merging di NED.

2.2.11 Packages

Tenere tutti i NED files all'interno di un'unica directory può andare bene per progetti di piccola dimensione. Quando un progetto cresce diventa necessario introdurre una struttura di directory e di conseguenza organizzare i NED files in **packages**. I packages sono inoltre utili per ridurre il conflitto di nomi dal momento che saranno qualificati con il nome del package. Se i NED files si trovano in `root/a/b/c` allora all'inizio dei files andrà inserita la dichiarazione:

```
package a.b.c;
```

La directory `root` è la così detta *NED source folder* e può essere cambiata modificando la variabile di sistema *NEDPATH*.

Se si vuole utilizzare un modulo contenuto in un file che si trova in `root/a/b/c` è possibile riferirsi ad esso in diversi modi:

- fully qualified name: si utilizza la dicitura `a.b.c.moduleName`
- simple name: se si importa il modulo allora si può utilizzare direttamente il suo nome
- se ci troviamo all'interno dello stesso package allora si può riferirsi direttamente ad esso con il simple module, senza importazione.

Un modulo viene importato tramite la keyword **import**, specificando il fully qualified name.

2.3 Simple Modules

I moduli semplici sono la parte attiva nella simulazione della rete. Sono programmati in C++, utilizzando la libreria OMNeT++. Il modello di simulazione di OMNeT++ è composto da moduli e connessioni; i moduli possono essere semplici oppure composti e, come specificato, i moduli semplici sono la parte attiva della simulazione ed il loro comportamento è definito da classi C++. I moduli e i canali sono rappresentati rispettivamente dalle classi **cModule** e **cChannel**, che derivano entrambe dalla classe **cComponent**. I moduli semplici in realtà sono rappresentati dalla classe **cSimpleModule**, che l'utente può estendere e personalizzare. In Figura 19 viene mostrata la gerarchia tra le classi base che OMNeT++ mette a disposizione.

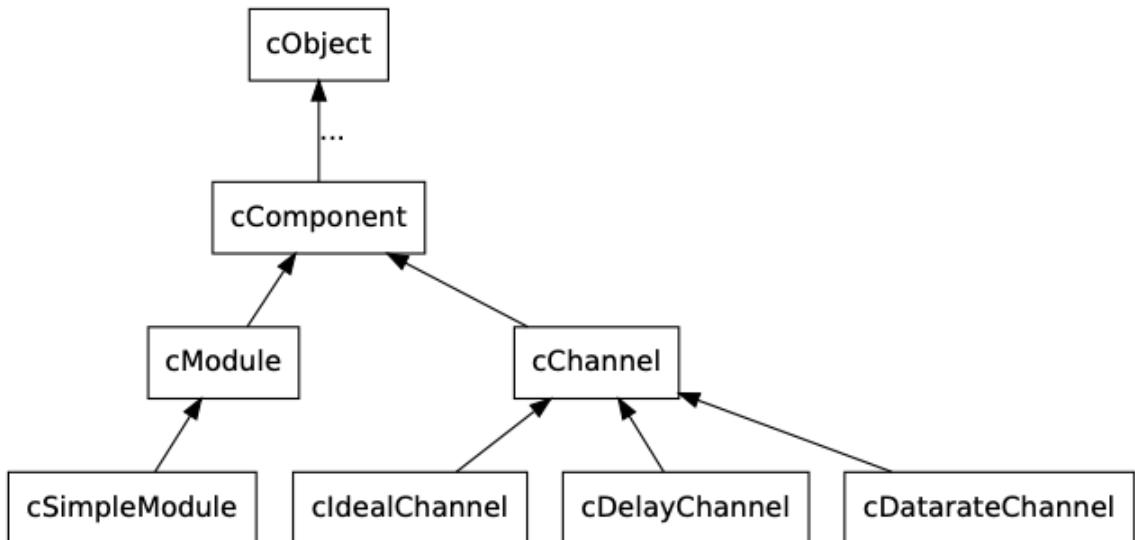


Figura 19: Ereditarietà delle classi

In OMNeT++ gli eventi si verificano all'interno dei simple modules. I simple modules encapsulano codice C++ che genera eventi e risponde ad eventi, implementando il comportamento del modulo. Le due funzioni principali da implementare sono **initialize()** e **handleMessage(cMessage *msg)**.

2.3.1 Overview

```
// file: HelloModule.cc
#include <omnetpp.h>
using namespace omnetpp;

class HelloModule : public cSimpleModule
{
protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

// register module class with OMNeT++
Define_Module(HelloModule);

void HelloModule::initialize()
{
    EV << "Hello World!\n";
}

void HelloModule::handleMessage(cMessage *msg)
{
    delete msg; // just discard everything we receive
}
```

Figura 20: Simple Module

I simple modules non sono altro che classi C++ che estendono la classe di OMNeT ++ `cSimpleModule`, ridefinendo alcune componenti per fornire un determinato comportamento al modulo. La classe C++ deve essere registrata con OMNeT ++ utilizzando la macro `Define_Module()`, che deve essere inclusa nei files .cc o .cpp, ma non negli header files .h. In Figura 20 viene mostrato un esempio di classe C++ che rappresenta un modulo semplice. La classe si chiama `HelloModule`, estende la classe `cSimpleModule` e si registra ad OMNeT ++ tramite la macro `Define_Module(HelloModule)`, che richiede ovviamente l'esistenza di un simple module di nome `HelloModule`. I moduli semplici non sono mai istanziati dall'utente direttamente, ma di questo si occupa il kernel di simulazione. Questo implica che non è possibile scrivere costruttori arbitrari, ma è permesso solamente il costruttore con nessun argomento.

2.3.2 Ciclo di vita

Come accennato nella Sezione 2.3.1 è il kernel di simulazione che si occupa dell’istanziazione dei moduli semplici e più in generale del ciclo di vita dell’intera simulazione. Due momenti particolari da considerare sono l’inizio e la fine della simulazione, quando il kernel invoca rispettivamente le funzioni **initialize()** e **finish()** di ogni simple module. Le due funzioni sono dichiarate nella classe **cComponent** e forniscono appunto all’utente la possibilità di eseguire codice all’inizio e alla fine della simulazione.

initialize()

La ragione dell’esistenza di *initialize()* è dovuta al fatto che la simulazione della rete è gestita appunto dal kernel di simulazione. Nessuno garantisce in che ordine e/o quali oggetti siano disponibili durante il momento in cui un modulo viene creato dal kernel, quindi potrebbe risultare fallimentare inserire del codice all’interno del costruttore. Le funzioni di inizializzazione, invece, vengono chiamate appena prima la simulazione cominci, quando tutto i componenti della rete sono disponibili.

finish()

La funzione di terminazione è sfruttata soprattutto per raccogliere statistiche sulla simulazione. Viene chiamata solo nel caso in cui la simulazione termina in modo normale e non nel caso in cui si verifichi un errore.

Ordine di invocazione

Le funzioni **initialize()** dei moduli vengono chiamate prima che il primo evento della simulazione venga processato. Le funzioni **finish()** vengono chiamate dopo che gli eventi della simulazione hanno terminato e solo nel caso in cui non ci siano stati errori durante la simulazione. Come detto nella Sezione 2.2.4 un compound module non possiede un comportamento attivo a differenza dei simple modules. È però possibile associare ad un compound module le funzioni di **initialize()** e **finish()** estendendo la classe **cModule** ed inserendo l’opzione **@class(<classname>)** all’interno del file NED.

Inizializzazione multi-stage

Se l’inizializzazione ad uno stage non è sufficiente è possibile definire un inizializzazione a più stages. L’utente dovrà ridefinire le funzioni **virtual void initialize(int stage)** e **virtual int numInitStages() const**. La funzione **numInitStages()** deve ritornare il numero di stage desiderati e l’utente deve poi ridefinire **initialize(int stage)** in funzione dello stage.

2.3.3 Inviare e ricevere messaggi

Una simulazione OMNeT ++ non è altro che un insieme di moduli semplici che comunicano tra di loro attraverso lo scambio di messaggi. L'essenza dei moduli semplici è creare, inviare, ricevere, salvare, modificare, schedulare e distruggere messaggi. In OMNeT ++ i messaggi sono istanze della classe `cMessage` o di classi che la estendono. I pacchetti di rete sono rappresentati dalla classe `cPacket`, che estende `cMessage`. I messaggi sono creati attraverso l'operatore `new` e distrutti tramite la keyword `delete`, quando questi non sono più necessari. I messaggi e i pacchetti sono descritti più in dettaglio nella Sezione 2.4.

Invio

I messaggi sono creati attraverso l'operatore `new` e vengono inviati attraverso le seguenti funzioni:

- `send(cMessage *msg, const char *gateName, int index=0);`
- `send(cMessage *msg, int gateId);`
- `send(cMessage *msg, cGate *gate);`

Il primo argomento di tutte le funzioni è il puntatore `cMessage*` al messaggio. La prima funzione prende come ulteriori argomenti il nome del gate su cui si vuole inviare e, nel caso questo sia un vettore, l'indice del gate. La seconda funzione prende come ulteriore argomento il gate id, mentre la terza il puntatore `cGate*` al gate. Queste ultime due sono generalmente più performanti della prima funzione, in quanto non devono eseguire una ricerca del gate tramite stringa. Nella Sezione 2.3.5 viene descritto in maniera più approfondita l'accesso ai gates.

Self messages

In simulazione è spesso necessario schedulare eventi nel futuro, per implementare timers, timeouts oppure delays. Alcuni esempi possono essere: un modulo che periodicamente crea ed invia dei messaggi, un server che processa jobs da una coda o ancora, l'implementazione dei timer di un protocollo di rete. In OMNeT ++ la schedulazione di eventi viene implementata facendo inviare ad un modulo semplice un messaggio a se stesso. Questo tipo di messaggi prendono il nome di **self-messages**. Il modulo può auto-inviarsi un messaggio utilizzando la funzione `scheduleAt` che prende due argomenti: il tempo, che solitamente viene indicato come `simTime() + delta`, e il messaggio. Un self-message può essere riconosciuto come tale utilizzando la funzione `isSelfMessage()`.

Ricezione

Ogni volta che un messaggio arriva ad un modulo viene chiamata la funzione di nome `handleMessage(cMessage *msg)`, ereditata da `cSimpleModule`. La funzione non esegue nulla di default e l'utente deve ridefinirla per processare il messaggio. In Figura 21 un esempio di codice di partenza per rappresentare un semplice protocollo di trasporto di rete. La funzione `handleMessage()` distingue appunto i casi in cui il messaggio sia un self-message, oppure arrivi dai livelli superiori o inferiori.

```

class FooProtocol : public cSimpleModule
{
    protected:
        // state variables
        // ...

    virtual void processMsgFromHigherLayer(cMessage *packet);
    virtual void processMsgFromLowerLayer(FooPacket *packet);
    virtual void processTimer(cMessage *timer);

    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

// ...

void FooProtocol::handleMessage(cMessage *msg)
{
    if (msg->isSelfMessage())
        processTimer(msg);
    else if (msg->arrivedOn("fromNetw"))
        processMsgFromLowerLayer(check_and_cast<FooPacket *>(msg));
    else
        processMsgFromHigherLayer(msg);
}

```

Figura 21: Handle Message

2.3.4 Parametri dei moduli

I parametri dei moduli dichiarati nei file NED sono rappresentati dalla classe **cPar** e sono accessibili tramite il metodo **par()** della classe **cComponent**.

```
int& listeningPort = par("listeningPort");
```

2.3.5 Gates

```
cGate *gIn=gate("g$i");
cGate *gOut=gate("g$o");

cGate *gInHalf=gateHalf("g", cGate::INPUT);
cGate *gOutHalf=gateHalf("g", cGate::OUTPUT);
```

Figura 22: Funzioni **gate()** e **gateHalf()**

I gates dei moduli sono rappresentati dalla classe **cGate**. La classe **cModule** fornisce una serie di funzioni per lavorare con i gates. Per accedere ad un gate si può utilizzare il metodo **gate()** che prende come argomento il nome del gate.

```
cGate *outGate = gate("out");
```

Quando dobbiamo leggere un gate di tipo **inout** la funzione **gate()** restituirebbe un **gate not found error**. Per indicare al metodo se ci si vuole riferire al gate di input piuttosto che a quello di output bisogna concatenare al nome del gate la stringa **\$i** oppure **\$o**. Lo stesso lavoro può essere svolto dalla funzione **gateHalf()**. Altre funzioni utili possono essere **hasGate()** e **getId()**. In Figura 22 degli esempi delle funzioni **gate()** e **gateHalf()**. Nel caso in cui si volesse accedere ad un vettore di gates la funzione **gate()** funziona anche passando come secondo argomento l'indice del gate desiderato all'interno del vettore. La lunghezza del vettore può essere calcolata tramite la funzione **gateSize()**. È possibile accedere ad un gate anche tramite il suo ID sempre tramite la funzione **gate()**. La particolarità di questa procedura è che gli IDs sono contigui all'interno di un vettore di gates. Quindi l'ID di **gate[k]** può essere calcolato come: (ID del **gate[0]**)+k.

2.3.6 Gerarchia del modulo

Ogni componente della rete (moduli e canali) possiede un ID che può essere ottenuto tramite la funzione `getId()`. L'ID identifica un modulo o un canale univocamente per l'intera durata della simulazione. Per svolgere il processo opposto, ossia ottenere un componente della rete tramite ID, è possibile utilizzare la funzione `getComponent()` oppure una delle sue due varianti `getModule()` e `getChannel()`. Tutte e tre le funzioni ovviamente prendono come argomento un intero che rappresenta un ID. Una feature interessante che OMNeT ++ mette a disposizione è la possibilità di navigare all'interno della gerarchia del modulo. In particolare è possibile accedere al modulo padre tramite la funzione `getParentModule()`.

2.3.7 Creazione dinamica di moduli

```
cModuleType *moduleType = cModuleType::get("Node");
cModule *module = moduleType->createScheduleInit("new_module", this);

module->deleteModule();
```

Figura 23: Creazione ed eliminazione dinamica

Alcune simulazioni potrebbero richiedere la possibilità di creare ed eliminare dinamicamente dei moduli. Un esempio pratico è la simulazione dell'arrivo di un nuovo utente all'interno di una rete cellulare; questo scenario potrebbe essere implementato creando un modulo a runtime. OMNeT ++ permette la creazione runtime sia di moduli semplici che composti. Per creare un modulo dinamicamente prima di tutto bisogna trovare la factory corrispondente a quel modulo tramite la funzione `get()` della classe `cModuleType`. A questo punto basta chiamare sull'oggetto la funzione `createScheduleInit(const char *name, cModule *parentmod)`. Per eliminare un modulo è necessario chiamare la funzione `deleteModule()`. Se il modulo in questione è un compound module il metodo implica l'eliminazione ricorsiva di tutti i suoi sottomoduli. In Figura 23 un esempio in cui viene creato dinamicamente un nuovo modulo di nome `new_module` di tipo `Node` che viene successivamente eliminato.

2.4 Messaggi e pacchetti

I messaggi sono un concetto fondamentale in OMNeT++. Nella simulazione un messaggio può rappresentare un evento, un pacchetto, un comando, un job oppure altre entità, che dipendono dal dominio della simulazione stessa. I messaggi sono rappresentati dalla classe **cMessage** e dalla sua sottoclasse **cPacket**, che solitamente viene utilizzata per i pacchetti di rete (frames, packets, datagrams). Gli utenti possono estendere queste classi per creare nuove tipologie di messaggi e pacchetti.

2.4.1 cMessage

La classe **cMessage** viene messa a disposizione da OMNeT++ per rappresentare un messaggio. Il costruttore prende come parametri una stringa ed un intero, entrambi opzionali, che rappresentano rispettivamente il nome ed il tipo di messaggio. Il nome viene mostrato in diverse posizioni nell’interfaccia grafica della simulazione, quindi è consigliabile utilizzare un nome descrittivo. L’intero, detto anche *message kind*, è utilizzabile liberamente dal programmatore per descrivere le categorie dei propri messaggi (i valori negativi sono riservati al kernel di simulazione). Il costruttore può essere invocato anche senza nessun parametro. Di seguito alcuni esempi di utilizzo del costruttore della classe **cMessage**:

```
cMessage *msg1 = new cMessage();
cMessage *msg2 = new cMessage("timeout");
cMessage *msg3 = new cMessage("timeout", KIND_TIMEOUT);
```

Figura 24: Costruttore della classe *cMessage*

La classe **cMessage** offre una serie di metodi **set** e **get** per modificare e ricevere i dati dell’istanza del messaggio, per esempio: **setName(const char *name)**, **setKind(short k)**, **setTimestamp()**, **getName()** ed altri ancora.

2.4.2 Self-Messages

In OMNeT++ i messaggi sono spesso utilizzati per rappresentare eventi interni ad un modulo, come anticipato nella Sezione 2.3.3. Un messaggio viene definito **self-message** quando viene utilizzato internamente per descrivere un evento. Un self-message è comunque un oggetto di tipo **cMessage** o una sua estensione. OMNeT++ mette a disposizione i metodi **isSelfMessage()** e **isScheduled()** che rispettivamente dicono se un messaggio è un self-message e se un messaggio è attualmente schedulato.

2.4.3 cPacket

La classe **cPacket** estende la classe **cMessage** per rappresentare i pacchetti di rete. Il costruttore è simile, ma accetta un parametro ulteriore: un intero che esprime la lunghezza del pacchetto in bit. Così come per **cMessage**, anche la classe **cPacket** offre una serie di metodi **set**" e **get** ed inoltre esistono i metodi **setBitError(bool e)** e **hasBitError()** che permettono di settare o ricevere l'informazione relativa ad un pacchetto corrotto.

2.4.4 Definire messaggi e pacchetti

Per aggiungere informazioni extra a un messaggio o pacchetto il programmatore deve estendere la classe **cMessage** oppure **cPacket**; questa operazione può risultare monotona e sprecare del tempo. Per risolvere questo problema OMNeT ++ mette a disposizione una soluzione più conveniente chiamata **message definitions**. Message definitions offre una sintassi compatta per descrivere il contenuto di un tipo di messaggio e il corrispondente codice C++ viene automaticamente generato sulla base della definizione. La definizione dei messaggi deve avvenire all'interno di un file con estensione **.msg**.

```
class MecUpdateMessage extends MecControlMessage{
    chunkLength = inet::B(400);
    ctlMsgType = MEC_STATUS_UPDATE;
    double memoryCapacityUsed;
    double computationalCapacityUsed;
    double storageCapacityUsed;
    double networkCapacityUsed;
};
```

Figura 25: Message definitions

In Figura 25 un esempio di definizione di un messaggio: la classe si chiama **MecUpdateMessage**, che estende **MecControlMessage**. Il codice C++, automaticamente generato nel momento di compilazione, offre inoltre i metodi **set** e **get** per ogni parametro definito.

2.5 Configurazione della simulazione

Una volta descritta la topologia della rete l'utente può ovviamente procedere con la simulazione, dopo aver definito la configurazione della simulazione. I dati di input e di configurazione per la simulazione vengono definiti in un file chiamato solitamente `omnetpp.ini`.

```
[General]
network = FifoNet
sim-time-limit = 100h
cpu-time-limit = 300s
#debug-on-errors = true
#record-eventlog = true

[Config Fifo1]
description = "low job arrival rate"
**.gen.sendIntTime = exponential(0.2s)
**.gen.msgLength = 100b
**.fifo.bitsPerSec = 1000bps

[Config Fifo2]
description = "high job arrival rate"
**.gen.sendIntTime = exponential(0.01s)
**.gen.msgLength = 10b
**.fifo.bitsPerSec = 1000bps
```

Figura 26: File `omnetpp.ini`

Il file di configurazione è un file di testo e line oriented. Può contenere *section heading lines*, *key-value lines* e *directive lines*. Le prime definiscono le sezioni del file, le seconde assegnano valori ai parametri della rete, mentre le ultime rappresentano direttive di importazione. In Figura 26 un esempio di file di configurazione: il file è diviso nelle sezioni `[General]`, `[Config Fifo1]` e `[Config Fifo2]`, ognuna delle quali contiene diverse entries. Ogni sezione rappresenta una configurazione della rete che si vuole simulare; la sezione `[General]` è quella considerata di default da OMNeT++. Ogni sezione contiene delle opzioni globali di configurazione (es. `network`) ed altre che riguardano determinati moduli della rete. Gli asterischi `**` permettono di fare pattern matching con qualsiasi componente che contiene dei punti nel suo namespace. I commenti sono preceduti da `#`.

2.6 Esecuzione della simulazione

Le simulazioni OMNeT ++ possono essere eseguite utilizzando diverse user interfaces, ossia diversi runtime environments. Le user interfaces supportate sono:

- Qtenv: Qt-based user interface grafica
- Tkenv: Tcl/tk-based user interface grafica
- Cmdenv: command-line user interface

L’interfaccia può essere specificata nella Sezione *Run/Run Configurations/User interface*. A questo punto la simulazione può essere avviata.

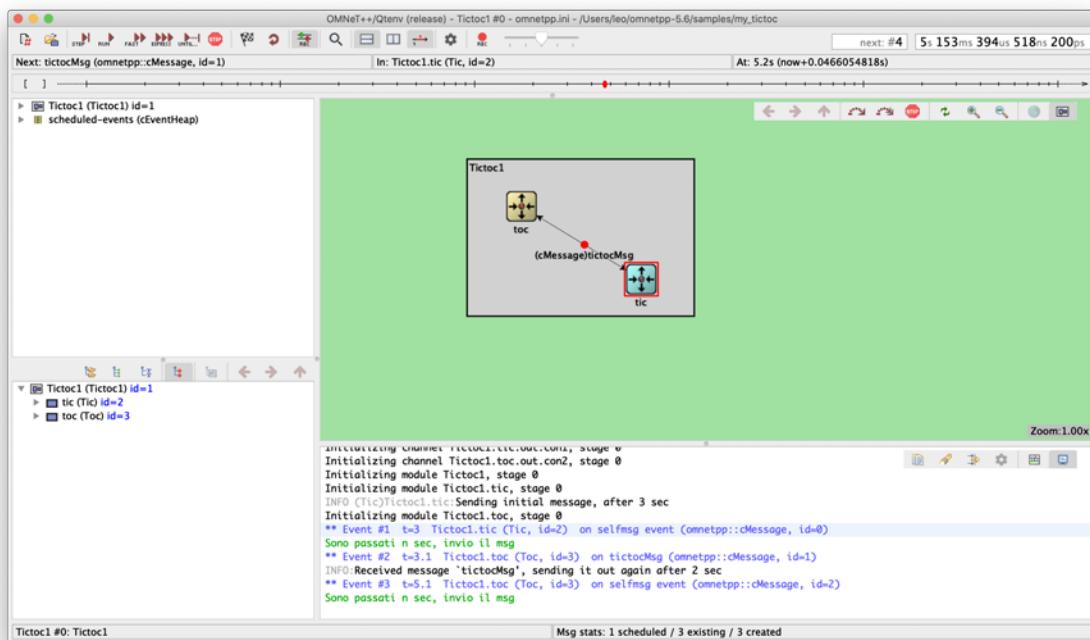


Figura 27: Esecuzione della simulazione

In Figura 27 un esempio di esecuzione di una semplice simulazione utilizzando l’interfaccia Qtenv. La simulazione può essere avviata, stoppata, eseguita a diverse velocità oppure re-inizializzata. Sulla parte inferiore si trova la console tramite la quale è possibile seguire i vari passi della simulazione.

2.7 Analisi dei risultati

OMNeT ++ mette a disposizioni delle funzionalità built-in per l'analisi dei risultati della simulazione. I risultati sono salvati all'interno della directory *results*. Esistono due modalità di registrazione dei dati statistici:

1. direttamente dal codice C++, utilizzando la libreria di simulazione
2. tramite l'utilizzo dei signal

2.7.1 Libreria di simulazione

Con questo approccio, i risultati scalari e statistici vengono raccolti in variabili dichiarate all'interno dei moduli e registrati nella fase di terminazione tramite la funzione `recordScalar()`. I vettori vengono registrati utilizzando la classe `cOutVector`. Per registrare le distribuzioni, in ottica di calcolare successivamente statistiche di riepilogo come media, deviazione standard, minimo o massimo, vengono utilizzate classi come `cHistogram` o `cPSquare`. Le statistiche possono essere aggiornate per i vettori e istogrammi rispettivamente tramite i metodi `record()` e `collect()`. La registrazione di singoli vettori, scalari e statistiche può essere abilitata o disabilitata tramite il file di configurazione.

2.7.2 Signals

Questo approccio sfrutta il meccanismo dei signals messo a disposizione da OMNeT++. Un signal deve essere dichiarato nel codice C++ del modulo che lo intende utilizzare e successivamente deve essere registrato; solitamente questa operazione viene fatto nel metodo `initialize()`. A questo punto il signal può essere emesso, tramite la funzione `emit()`, che prende come argomenti il signal e il dato che si vuole registrare. L'ultimo step è definire il signal anche nel codice NED del modulo. A questo punto nel file di configurazione è possibile scegliere il metodo di visualizzazione (istogramma/vettore) per ogni signal dichiarato, senza modificare il codice C++ come nella soluzione precedente.

Capitolo 3

INET

I componenti della rete del progetto sono stati realizzati servendosi del framework INET. INET è una libreria open source per il simulatore OMNeT ++, che fornisce protocolli, componenti ed altre strutture per la simulazione delle reti di comunicazione, rendendolo molto utilizzato per la creazione e validazione di nuovi protocolli e scenari di rete [4]. INET fornisce l'intero stack TCP/IP, implementando protocolli come TCP, UDP, IPv4, IPv6, OSPF, BGP, ma anche protocolli di livello di collegamento sia cablati che wireless, come Ethernet, PPP, IEEE 802.11, ecc. INET offre supporto per la mobilità, protocolli MANET, DiffServ e MPLS. Infine è possibile trovare nel framework diversi modelli di applicazioni e molti altri protocolli e componenti. Molti altri framework di simulazione prendono INET come base e lo estendono in direzioni specifiche, come reti veicolari, reti overlay/peer-to-peer o LTE. Alcuni esempi di estensioni sono *OverSim*, *Veins* o *SimuLTE*.

Features di INET

- Implementazione dei livelli dello stack TCP/IP (physical, link-layer, network, transport, application)
- Livello di rete (IPv4 e IPv6)
- Livello di trasporto (TCP, UDP, SCTP)
- Protocolli di routing
- Interfacce di rete cablate e wireless (Ethernet, PPP, IEEE 802.11, etc.)
- Livello fisico
- Diversi modelli di applicazioni

INET è costruito attorno al concetto di moduli che comunicano tramite lo scambio di messaggi. Agenti e protocolli di rete sono rappresentati da componenti, che possono essere combinati liberamente per formare host, router, switch ed altri dispositivi di rete. Nuovi componenti possono essere programmati dall'utente e i componenti esistenti sono stati scritti in modo che siano facili da capire e modificare. INET beneficia dell'infrastruttura fornita da OMNeT ++: oltre ad utilizzare i servizi forniti dal kernel e dalla libreria di simulazione (modellazione dei componenti, parametrizzazione, registrazione dei risultati, ecc.). Questo significa anche che i modelli possono essere sviluppati, assemblati, parametrizzati, eseguiti ed i loro risultati valutati comodamente dall'IDE di simulazione o dalla riga di comando.

Directory INET I moduli di INET sono organizzati in una struttura di directory che segue i protocolli di rete:

- `src/inet/applications/` – traffic generators and application models
- `src/inet/transportlayer/` – transport layer protocols
- `src/inet/networklayer/` – network layer protocols and accessories
- `src/inet/linklayer/` – link layer protocols and accessories
- `src/inet/physicallayer/` – physical layer models
- `src/inet/routing/` – routing protocols (internet and ad hoc)
- `src/inet/mobility/` – mobility models
- `src/inet/power/` – energy consumption modeling
- `src/inet/environment/` – model of the physical environment
- `src/inet/node/` – preassembled network node models
- `src/inet/visualizer/` – visualization components (2D and 3D)
- `src/inet/common/` – miscellaneous utility components

I packages corrispondono alle directories che si trovano all'interno di `src/`, quindi per esempio la directory `src/inet/transportlayer/tcp` corrisponde al package `inet.transportlayer.tcp`.

3.1 Componenti della rete

INET fornisce diversi nodi di rete preassemblati con componenti accuratamente selezionati. Supportano la personalizzazione tramite parametri e parametric submodule types, ma non sono pensati per essere universali. A volte può essere necessario creare modelli di nodi di rete per particolari scenari di simulazione. In ogni caso, il seguente elenco fornisce un assaggio dei nodi di rete incorporati.

- **StandardHost**: contiene i protocolli Internet più comuni: UDP, TCP, IPv4, IPv6, Ethernet, IEEE 802.11. Supporta anche un modello di mobilità opzionale, modelli energetici opzionali ed un numero qualsiasi di applicazioni completamente configurabili dal file ini.
- **EtherSwitch**: modella uno switch Ethernet contenente un'unità MAC per porta.
- **Router**: fornisce i protocolli di instradamento più comuni: OSPF, BGP, RIP, PIM (Figura 28).
- **AccessPoint**: modella un punto di accesso Wifi con più interfacce di rete IEEE 802.11 e più porte Ethernet.
- **WirelessHost**: fornisce un nodo di rete con un'interfaccia di rete IEEE 802.11 (predefinita), adatta per l'utilizzo con un AccessPoint.

I nodi comunicano a livello di rete scambiando messaggi OMNeT ++ che sono rappresentazioni astratte di segnali fisici sul mezzo di trasmissione. I segnali encapsulano pacchetti specifici di INET che rappresentano i dati digitali trasmessi. I pacchetti sono ulteriormente suddivisi in blocchi che forniscono rappresentazioni per porzioni di dati più piccole (ad es. intestazioni di protocollo, dati dell'applicazione).

Inoltre, ci sono dei componenti che non sono modelli di nodi di rete fisici, ma sono necessari per modellare altri aspetti, per esempio:

- un modulo di tipo `radio medium` (es. `Ieee80211RadioMedium`) sono un componente richiesto delle reti wireless.
- `PhysicalEnvironment` modella l'effetto dell'ambiente fisico, ovvero gli ostacoli, sulla propagazione del segnale radio. È un componente opzionale.
- moduli di tipo `configurators` (es. `Ipv4NetworkConfigurator`) configurano vari aspetti della rete. Ad esempio, il configuratore IPv4 assegna indirizzi IP a

host e router e imposta il routing statico. Viene utilizzato quando si modella l'assegnazione dell'indirizzo IP dinamico (ad es. tramite DHCP) o il routing dinamico non è importante.

```

module Router extends ApplicationLayerNodeBase
{
    parameters:
        @display("i=abstract/router");
        @figure[submodules];
        forwarding = true;
        bool hasOspf = default(false);
        bool hasRip = default(false);
        bool hasBgp = default(false);
        bool hasPim = default(false);
        bool hasDhcp = default(false);
        hasUdp = default(hasRip || hasDhcp);
        hasTcp = default(hasBgp);
        *.routingTableModule = default("^.ipv4.routingTable");

    submodules:
        ospf: <default("Ospfv2")> like IOspf if hasOspf {
            parameters:
                @display("p=975,226");
        }
        bgp: <"Bgp"> like IBgp if hasBgp {
            parameters:
                ospfRoutingModule = default(hasOspf ? ".ospf" : "");
                @display("p=600,100");
        }
        rip: <"Rip"> like IApp if hasRip {
            parameters:
                @display("p=975,76");
        }
        pim: <"Pim"> like IPim if hasPim {
            parameters:
                @display("p=825,226");
        }
        dhcp: <"DhcpServer"> like IApp if hasDhcp {
            parameters:
                @display("p=1125,76");
        }

    connections allowunconnected:
        ospf.ipOut --> tn.in++ if hasOspf;
        ospf.ipIn <-- tn.out++ if hasOspf;

        bgp.socketOut --> at.in++ if hasBgp;
        bgp.socketIn <-- at.out++ if hasBgp;

        rip.socketOut --> at.in++ if hasRip;
        rip.socketIn <-- at.out++ if hasRip;

        pim.networkLayerOut --> tn.in++ if hasPim;
        pim.networkLayerIn <-- tn.out++ if hasPim;

        dhcp.socketOut --> at.in++ if hasDhcp;
        dhcp.socketIn <-- at.out++ if hasDhcp;
}

```

Figura 28: NED File src/inet/node/inet/Router.ned

3.2 Livello fisico

In una rete di comunicazione la trasmissione può avvenire in due modalità: cablata o senza cavi; modalità meglio conosciute rispettivamente con i nomi **wired** e **wireless**.

3.2.1 Wired Network

```
network WiredNetworkExample
{
    parameters:
        int numClients; // number of clients in the network
    submodules:
        configurator: Ipv4NetworkConfigurator; // network autoconfiguration
        server: StandardHost; // predefined standard host
        router: Router; // predefined router
        switch: EtherSwitch; // predefined ethernet switch
        client[numClients]: StandardHost;
    connections: // network level connections
        router.pppg++ <--> { datarate = 1GBps; } <--> server.pppg++; // PPP
        switch.ethg++ <--> Eth1G <--> router.ethg++; // bidirectional ethernet
        for i=0..numClients-1 {
            client[i].ethg++ <--> Eth1G <--> switch.ethg++; // ethernet
        }
}
```

Figura 29: Wired Network

Le connessioni di rete cablate, ad esempio i cavi Ethernet, sono rappresentate con connessioni OMNeT ++ standard utilizzando il tipo NED **DatarateChannel**. I parametri **datarate** e **delay** del canale devono essere forniti per tutte le connessioni cablate. Il numero di interfacce cablate in un host di solito non necessita di essere configurato manualmente, perché può essere dedotto automaticamente dal numero effettivo di collegamenti ai nodi vicini. La Figura 29 mostra quanto sia semplice creare un modello per una rete cablata. Questa rete contiene un server connesso ad un router tramite PPP, che a sua volta è connesso ad uno switch tramite Ethernet. La rete contiene anche un numero parametrizzabile di client, tutti collegati allo switch, che formano una topologia a stella. I nodi di rete utilizzati sono tutti moduli predefiniti in INET. Per evitare la configurazione manuale degli indirizzi IP e delle tabelle di instradamento, è incluso anche un configuratore di rete automatico.

Per eseguire una simulazione utilizzando la rete in Figura 29, è necessario creare un file di configurazione. Il file seleziona la rete, imposta il parametro relativo al numero di client e configura una semplice applicazione TCP per ogni client. Il server è configurato per avere un'applicazione TCP che rispecchia tutti i dati ricevuti dai client individualmente. Quando viene eseguita la simulazione, ogni applicazione client si connette al server utilizzando un socket TCP. Quindi ognuno di loro invia 1 MB di dati, che a sua volta viene ripreso dal server e la simulazione si conclude. Le statistiche predefinite vengono scritte nella cartella dei risultati della simulazione per un'analisi successiva.

3.2.2 Wireless Network

```
network WirelessNetworkExample
  submodules:
    configurator: Ipv4NetworkConfigurator;
    radioMedium: IEEE80211ScalarRadioMedium;
    host1: WirelessHost { @display("p=200,100"); }
    host2: WirelessHost { @display("p=500,100"); }
    accessPoint: AccessPoint { @display("p=374,200"); }
}
```

Figura 30: Wireless Network

Le connessioni di rete wireless non sono modellate con le connessioni OMNeT ++ a causa della natura della connettività che cambia dinamicamente. Per le reti wireless, è necessario un modulo aggiuntivo, che rappresenta il mezzo di trasmissione, per mantenere le informazioni sulla connettività. Nella rete in Figura 30, le posizioni nelle “display strings” forniscono le posizioni per il mezzo di trasmissione durante il calcolo della propagazione del segnale e della perdita di percorso.

3.3 Livello data-link

Così come il mezzo fisico di propagazione viene distinto in cablato e senza fili, i nodi che comunicano nella rete a loro volta devono disporre di interfacce di rete wired e wireless.

3.3.1 Interfacce Wired

Le interfacce cablate hanno una coppia di porte OMNeT ++ che rappresentano la capacità di avere una connessione fisica esterna verso un altro nodo di rete (ad es. Porta Ethernet). Per far funzionare la comunicazione cablata, queste porte devono essere collegate con connessioni che rappresentano il cavo fisico tra le porte fisiche, ossia con connessioni di tipo `DatarateChannel`. Le interfacce di rete cablate sono moduli composti che implementano l'interfaccia `IWiredInterface`. INET dispone delle seguenti interfacce di rete cablate:

- PPP
- Ethernet

3.3.2 Interfacce Wireless

Le interfacce wireless utilizzano l'invio diretto per la comunicazione invece dei collegamenti. I loro moduli composti non hanno porte di uscita a livello fisico, ma solo una porta di ingresso dedicata alla ricezione. Un'altra differenza, rispetto alle interfacce cablate, è che le interfacce wireless richiedono e collaborano con un *transmission medium* a livello di rete. Il transmission medium rappresenta il mezzo di trasmissione condiviso (campo elettromagnetico o mezzo acustico). È responsabile della modellazione di effetti fisici come l'attenuazione del segnale e mantiene le informazioni di connettività. Le interfacce di rete wireless sono moduli composti che implementano l'interfaccia `IWirelessInterface`.

3.4 Livello di rete

Il protocollo IP è uno dei principali della suite di protocolli TCP/IP. Tutti i pacchetti UDP, TCP, ICMP vengono incapsulati in pacchetti IP e trasportati dal livello IP. Mentre i protocolli di livello superiore trasferiscono i dati tra due endpoint di comunicazione, il livello IP fornisce un servizio di consegna hop-by-hop, inaffidabile e connectionless. I nodi che sono connessi a Internet possono essere un host o un router. Gli host possono inviare e ricevere pacchetti IP ed il loro sistema operativo implementa l'intero stack TCP/IP incluso il livello di trasporto. D'altra parte, i router hanno più di una scheda di interfaccia ed eseguono l'instradamento dei pacchetti tra le reti connesse. La divisione tra router e host non è rigida, perché se un host ha più interfacce, di solito queste possono essere configurate per funzionare anche come router. Ogni nodo su Internet ha un indirizzo IP univoco ed i pacchetti IP contengono l'indirizzo IP della destinazione. Il compito dei router è scoprire l'indirizzo IP del prossimo hop sulla rete locale ed inoltrare il pacchetto ad esso. Il framework INET contiene diversi moduli per costruire il livello di rete IPv4 di host e router:

- **Ipv4:** è il modulo principale che implementa RFC791. Questo modulo esegue l'incapsulamento/decapsulamento IP, la frammentazione e l'assemblaggio e l'instradamento di datagrammi IP.
- **Ipv4RoutingTable:** è un modulo di supporto che gestisce la tabella di instradamento. Viene richiesto dal modulo Ipv4 per i percorsi migliori e aggiornato dai demoni di routing che implementano i protocolli RIP, OSPF, Manet, ecc.
- **Icmp:** può essere utilizzato per generare pacchetti di errore ICMP. Supporta anche le applicazioni ICMP echo.
- **Arp:** esegue la traduzione dinamica degli indirizzi IP in indirizzi MAC.

Questi moduli sono assemblati in un modulo composto a livello di rete chiamato `Ipv4NetworkLayer`. Il modulo `Ipv4NetworkLayer` è presente per esempio nei moduli `StandardHost` e `Router`.

3.5 Livello di trasporto

I protocolli del livello di trasporto forniscono servizi di comunicazione host-to-host per le applicazioni. Forniscono servizi come comunicazione orientata alla connessione, affidabilità, controllo del flusso e multiplexing. INET attualmente fornisce il supporto per i protocolli del livello di trasporto **TCP**, **UDP**, **SCTP** e **RTP**. I nodi INET come **StandardHost** contengono istanze opzionali e sostituibili di questi protocolli, come mostrato in Figura 31:

```
tcp: <default("Tcp")> like ITcp if hasTcp;
udp: <default("Udp")> like IUdp if hasUdp;
sctp: <default("Sctp")> like ISctp if hasSctp;
```

Figura 31: Protocolli di livello di trasporto in *StandardHost*

Poiché RTP è più specializzato degli altri (streaming multimediale), INET fornisce un tipo di nodo separato, **RtpHost**, per la modellazione del traffico RTP.

3.5.1 TCP

Il protocollo TCP è il protocollo più utilizzato in Internet. Fornisce una consegna affidabile e ordinata del flusso di byte da un'applicazione su un computer ad un'altra applicazione su un altro computer. Il protocollo TCP di base è descritto in RFC793, ma altre decine di RFC contengono modifiche ed estensioni al TCP. Di conseguenza, il TCP è un protocollo complesso e talvolta è difficile vedere come i diversi requisiti interagiscono tra loro. INET contiene tre implementazioni del protocollo TCP:

- **Tcp**: è l'implementazione principale, progettata per leggibilità, estensibilità e sperimentazione.
- **TcpLwip**: è un wrapper attorno alla libreria *lwIP* (Lightweight IP), uno stack TCP/IP open source ampiamente utilizzato nei sistemi embedded.
- **TcpNsc**: include *Network Simulation Cradle* (NSC), una libreria che consente di utilizzare stack di rete TCP/IP del mondo reale all'interno di un simulatore di rete.

Tutti e tre i tipi di modulo implementano l'interfaccia **ITcp**.

3.5.2 UDP

Il protocollo UDP è un protocollo di trasporto molto semplice, che fondamentalmente rende i servizi del livello di rete disponibili alle applicazioni. Esegue solo il multiplexing ed il demultiplexing dei pacchetti alle porte e solo un rilevamento di errori di base. Il modulo semplice **Udp** implementa il protocollo UDP. Esiste un’interfaccia del modulo (**IUdp**) che definisce i gate del componente **Udp**. Nel nodo **StandardHost**, il componente UDP può essere qualsiasi modulo che implementa tale interfaccia.

3.5.3 SCTP

Il modulo **Sctp** implementa il protocollo SCTP (Stream Control Transmission Protocol). Come TCP, SCTP fornisce una consegna affidabile dei dati ordinati su una rete affidabile. La caratteristica più importante di SCTP è la capacità di trasmettere più flussi di dati contemporaneamente tra due endpoint che hanno stabilito una connessione.

3.5.4 RTP

Il protocollo *RTP* (*Real-time Transport Protocol*) è un protocollo del livello di trasporto per la distribuzione di audio e video su reti IP. RTP è ampiamente utilizzato nei sistemi di comunicazione ed intrattenimento che coinvolgono lo streaming multimediale, come telefonia, applicazioni di teleconferenza video, servizi televisivi e funzionalità push-to-talk basate sul web. *RTP Control Protocol* (RTCP) è un protocollo gemello del Real-time Transport Protocol (RTP). RTCP fornisce statistiche fuori banda e informazioni di controllo per una sessione RTP. INET fornisce i seguenti moduli:

- **Rtp**: implementa il protocollo RTP
- **Rtcp**: implementa il protocollo RTCP

3.6 Livello di applicazione

Tutte le applicazioni implementano l’interfaccia del modulo `IApp`. Le applicazioni INET si dividono in due categorie. Nella prima categoria, le applicazioni implementano comportamenti molto specifici e generano modelli di traffico corrispondenti in base ai loro parametri specifici. Nella seconda categoria, le applicazioni sono più generiche e separano la generazione di traffico dall’utilizzo del protocollo, ad esempio `Udp` o `Tcp`.

3.6.1 Applicazioni TCP

INET mette a disposizione una serie di applicazioni preassemblate basate su TCP, di seguito un elenco dei principali moduli.

- **TcpBasicClientApp**: client per un protocollo generico in stile richiesta-risposta su TCP.
- **TcpSinkApp**: accetta qualsiasi numero di connessioni TCP in entrata e scarta tutto ciò che arriva su di esse.
- **TcpGenericServerApp**: applicazione server generica per la modellazione di protocolli o applicazioni in stile richiesta-risposta basati su TCP.
- **TcpEchoApp**: accetta un numero qualsiasi di connessioni TCP in entrata e restituisce i dati che arrivano su di esse.
- **TcpSessionApp**: apre una connessione, invia il numero di byte specificato e si chiude.

3.6.2 Applicazioni UDP

INET fornisce le seguenti applicazioni basate su UDP:

- **UdpBasicApp**: invia pacchetti UDP ad un determinato indirizzo IP in un determinato intervallo.
- **UdpBasicBurst**: invia pacchetti UDP agli indirizzi IP forniti a raffica o funge da packet sink.
- **UdpEchoApp**: è simile a `UdpBasicApp`, ma restituisce il pacchetto dopo la ricezione.
- **UdpSink**: consuma e stampa i pacchetti ricevuti dal modulo `Udp`.
- **UdpVideoStreamClient**: simula lo streaming video su UDP.

Capitolo 4

Progetto

In questo Capitolo viene illustrato il progetto centrale della tesi, ossia la progettazione e lo sviluppo dell’architettura ETSI MEC nel simulatore OMNeT ++. In particolare l’attenzione è posta sul componente *MEC orchestrator*, sui messaggi che scambia con le altre entità e sulle sue strutture dati di controllo. Sulla base del framework ETSI MEC è stata ricreata l’architettura del sistema, adottando alcune semplificazioni, come spiegato nella Sezione 4.1. Il progetto prevede la realizzazione delle entità MEC orchestrator, MEC host, MEC platform e delle user device applications. L’obbiettivo è quello di ricreare, all’interno di un ambiente di simulazione, lo scenario di lavoro del sistema ETSI MEC, dalla richiesta da parte dell’utente di servizi alla ricezione da parte dell’orchestratore, con relativo look up sul sistema globale per la gestione del processo di inizializzazione e successivo monitoraggio dell’architettura MEC.

Le entità di rete che compongono il sistema sono state realizzate servendosi del framework INET. INET è una libreria open source per il simulatore OMNeT ++, che fornisce protocolli, componenti ed altre strutture per la simulazione delle reti di comunicazione. Il framework INET viene illustrato nel Capitolo 3.

Nella Sezione 4.1 vengono descritti i principali moduli che compongono il progetto e come questi sono collegati per dialogare assieme, mentre nelle sezioni successive vengono spiegate le scelte progettuali che sono state adottate, quindi le classi e le interfacce che si è deciso di implementare ed utilizzare. Inoltre verranno illustrati i messaggi di rete che sono stati creati per la comunicazione tra le entità del progetto e verrà spiegata la configurazione dei vari componenti del sistema per la simulazione.

4.1 Componenti MEC

Sulla base del framework ETSI MEC, descritto nella Sezione 1.1, è stata ricreata l’architettura MEC, adottando alcune semplificazioni. Lo studio e lo sviluppo del sistema si focalizzano sulla componente di gestione dell’architettura e di monitoraggio delle risorse del sistema stesso. Il **MEC orchestrator** è quindi il soggetto principale del progetto, a discapito di altre entità altrettanto importanti, ad esempio l’infrastruttura di virtualizzazione o le MEC applications. Sono stati realizzati i componenti MEC orchestrator, MEC host, MEC platform e le user device applications. Il **MEC orchestrator** ingloba tutte le funzioni e le entità che nel framework svolgono attività di controllo nel system level. Il **MEC host** ospita la **MEC platform**, che a sua volta ingloba le sue funzionalità di controllo e di effettivo lavoro. Il **MEC user** rappresenta tutte le richieste che possono arrivare esternamente al sistema, come per esempio la richiesta di utilizzo di servizi MEC.

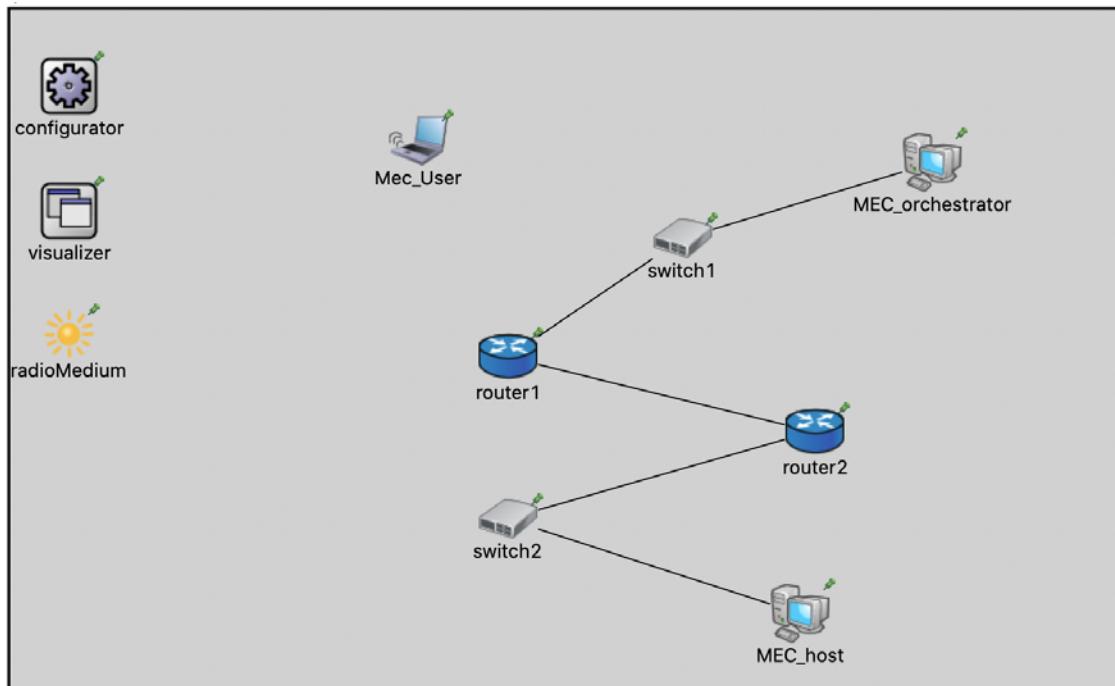


Figura 32: File MEC.ned

In Figura 32 la rappresentazione grafica del file MEC.ned, che sarà spiegato nel dettaglio nella Sezione 4.2. Il file illustra la struttura base dell’architettura che è stata creata sulla base del framework ETSI MEC.

4.1.1 MEC Orchestrator

Il MEC orchestrator è il soggetto principale del progetto. Mantiene una visione globale sull'intera architettura e dialoga sia con il MEC host che con il MEC user. L'orchestratore ingloba tutte le funzionalità dei componenti che compongono il system level del framework ETSI MEC, come l'Operations support system (OSS), lo User application lifecycle management proxy e NFVO. Tutti questi componenti sono spiegati in dettaglio nel Capitolo 1. Il lavoro svolto dal MEC orchestrator nel progetto rispecchia una semplificazione delle funzionalità del system level del framework MEC. Di seguito una breve spiegazione delle principali funzioni implementate, che saranno illustrate più nel dettaglio nelle prossime sezioni.

- Il MEC orchestrator riceve le richieste di utilizzo di servizi MEC da parte del MEC user. Un servizio può essere composto dalla richiesta di istanziazione di più MEC applications, ognuna delle quali potenzialmente richiesta con particolari parametri di utilizzo. Il MEC orchestrator convalida o meno le richieste, poi procede alla comunicazione con uno o più MEC host per gestire il MEC service.
- Il MEC orchestrator registra tutti i MEC host che si dichiarano a lui e li memorizza in apposite strutture dati.
- Il MEC orchestrator, una volta rilevato un MEC host, continua a riceverne aggiornamenti riguardanti lo stato.
- Il MEC orchestrator comunica con il MEC host per avviare un'applicazione e riceve una sua risposta di avvenuta istanziazione.

4.1.2 MEC Host e MEC Platform

Il componente MEC host contiene una MEC platform. Nel progetto queste due entità svolgono principalmente un ruolo di gestione, ricoprendo ciò che nel framework MEC viene svolto dal MEC platform manager. Infatti il MEC host contiene una MEC platform app che, in fase di inizializzazione della simulazione, si dichiara al MEC orchestrator e gli fornisce le proprie caratteristiche hardware. Da quel momento in poi la MEC platform aggiorna il MEC orchestrator periodicamente sullo stato delle sue risorse. Inoltre la MEC platform è predisposta per ricevere la richiesta di avvio di un'applicazione e comunicarne la risposta al MEC orchestrator. Uno scenario più reale dell'architettura MEC presenta molti MEC host, posizionati ai margini della rete, vicino agli utilizzatori.

4.1.3 MEC User

Il MEC user è il componente che nel progetto rappresenta tutte le richieste dall'esterno verso l'architettura MEC. Tramite il MEC user viene riproposto, in maniera semplificata, il comportamento delle *device applications* e del *customer facing service portal* del framework ETSI MEC. Il MEC user, in fase di inizializzazione, schedula l'invio di un messaggio all'orchestratore, contenente la richiesta di attivazione di un servizio MEC. Una volta che l'orchestratore gestisce l'avvio del servizio notifica il MEC user con un messaggio. L'intero processo che il MEC orchestrator compie per attivare il servizio e successivamente per gestirlo resta trasparente al MEC user. In una situazione più reale gli utenti utilizzatori del sistema sono ovviamente più di uno, alcuni dei quali potrebbero anche essere in movimento.

4.1.4 Sequence Diagram della simulazione

In Figura 33 viene proposto un sequence diagram che rappresenta la comunicazione che avviene durante la simulazione del progetto. In una prima parte avviene la dichiarazione da parte del MEC host verso il MEC orchestrator. Successivamente il MEC user richiede un servizio al MEC orchestrator; per semplicità il MEC user invia la richiesta dopo un lasso di tempo sufficiente per compiere la registrazione del MEC host da parte del MEC orchestrator. Una volta che il MEC orchestrator ha ricevuto la richiesta di attivazione di un servizio contatta il MEC host per istruirlo ad eseguire l'applicazione che compone il servizio. Anche qui si è scelto di utilizzare un servizio composto da un'unica applicazione MEC ed inoltre, vista la presenza di un solo MEC host nel progetto, il MEC orchestrator non compie nessuna scelta per istanziare il servizio, ma semplicemente lo inoltra al MEC host. Dopo questo il MEC host risponde al MEC orchestrator che a sua volta risponde al MEC user. Infine, parallelamente a questo processo, il MEC host dopo essersi dichiarato continua ad inviare periodicamente messaggi di update al MEC orchestrator, contenenti informazioni sul suo stato. La descrizione delle tipologie di messaggi che vengono scambiati durante la simulazione viene illustrata in dettaglio nella Sezione 4.8.

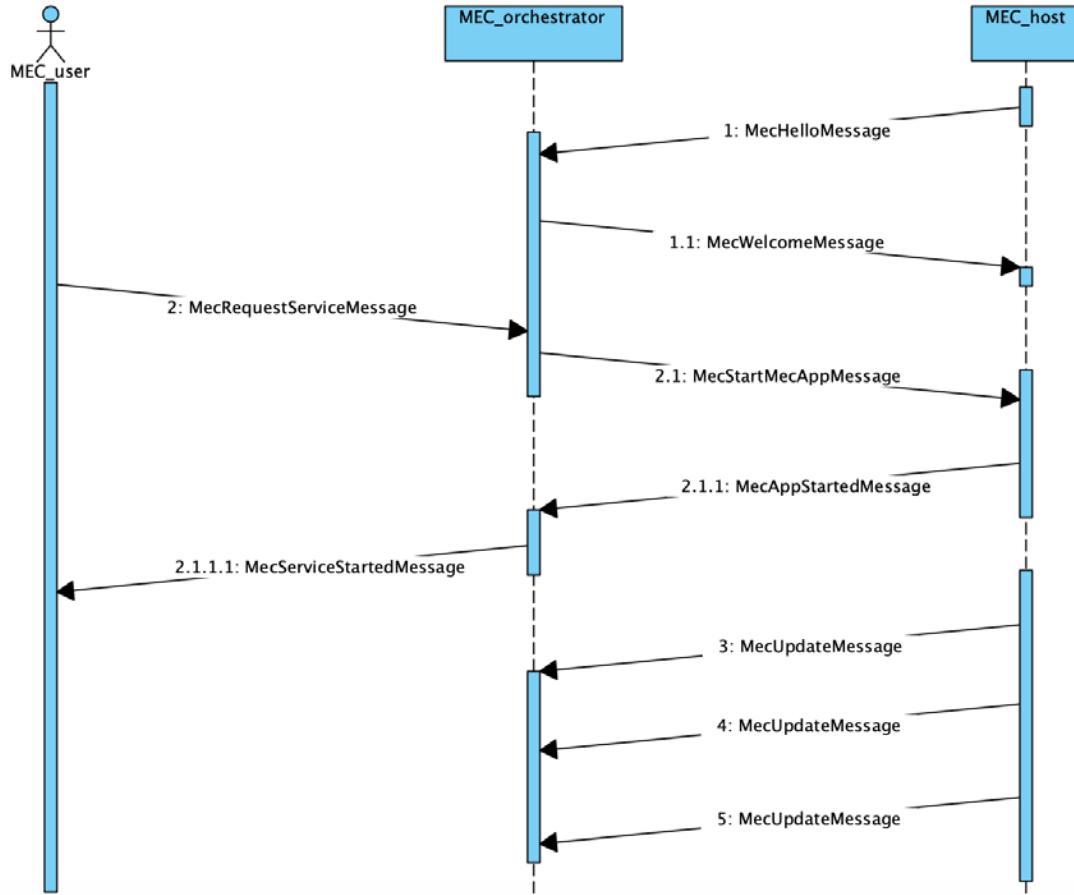


Figura 33: Sequence Diagram della simulazione

4.1.5 Collegamenti

I tre componenti MEC orchestrator, MEC host e MEC user sono collegati nel progetto attraverso una semplice rete, composta da due switch e due router, come mostrato in Figura 32. Il MEC user utilizza una comunicazione wireless per agganciarsi alla rete. Gli indirizzi IP vengono assegnati durante la fase di inizializzazione della simulazione da un componente chiamato *configurator*, che è illustrato nella Sezione 3.1. Ovviamente la topologia delle reti potrebbe essere diversa e più complicata. L'obiettivo principale del progetto è quello di ricreare, all'interno di un ambiente di simulazione, uno scenario di utilizzo del sistema MEC, ponendo il focus sulla componente di gestione delle varie entità, soprattutto sul MEC orchestrator. Per questo motivo si è preferito tenere semplice la topologia delle reti. In uno scenario più concreto dobbiamo immaginarcici i vari MEC host fisicamente vicini ai MEC user, mentre il MEC orchestrator che potenzialmente potrebbe trovarsi più distante dal resto. Non

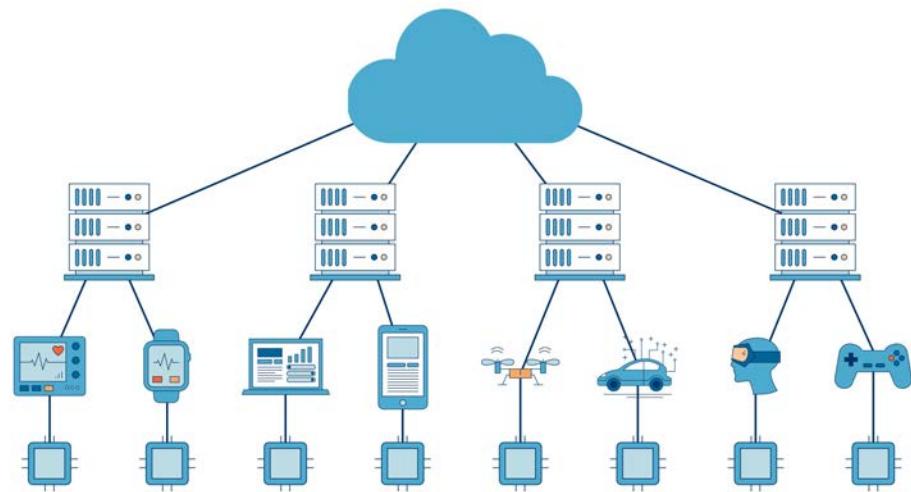


Figura 34: Edge Computing

scordiamoci infatti che l’architettura ETSI MEC è un’architettura di tipo *Edge Computing*, come mostrato in Figura 34, immagine che ricorda la struttura “piramidale” di questa architettura di rete. In basso troviamo tutto ciò che si posiziona ai margini della rete, ossia gli utilizzatori con le loro device applications, seguiti dai nodi edge che offrono i servizi (MEC user e MEC host). In alto, a livello cloud, troviamo la parte di gestione dell’intero sistema, che è composta da molte meno entità rispetto a quelle al margine della rete (MEC orchestrator).

4.2 La rete

Nelle prossime sezioni verranno illustrate le principali scelte progettuali che sono state adottate per l'implementazione dei componenti del progetto. Questo processo è stato sviluppato sulla base delle funzionalità di ogni singolo componente spiegate nel framework ETSI MEC (Sezione 1.1), tenendo conto delle semplificazioni adottate ed illustrate nella Sezione 4.1. Per prima cosa viene data una visione globale della rete del progetto, entrando successivamente nello specifico di ogni modulo, spiegandone le caratteristiche e le funzionalità. Verranno illustrate le interfacce e classi le C++ utilizzate ed implementate, così come per i file NED. Infine una spiegazione sulle varie tipologie di messaggi che sono stati creati per la comunicazione ed una overview sul file di configurazione, per predisporre ogni componente della rete all'esecuzione della simulazione.

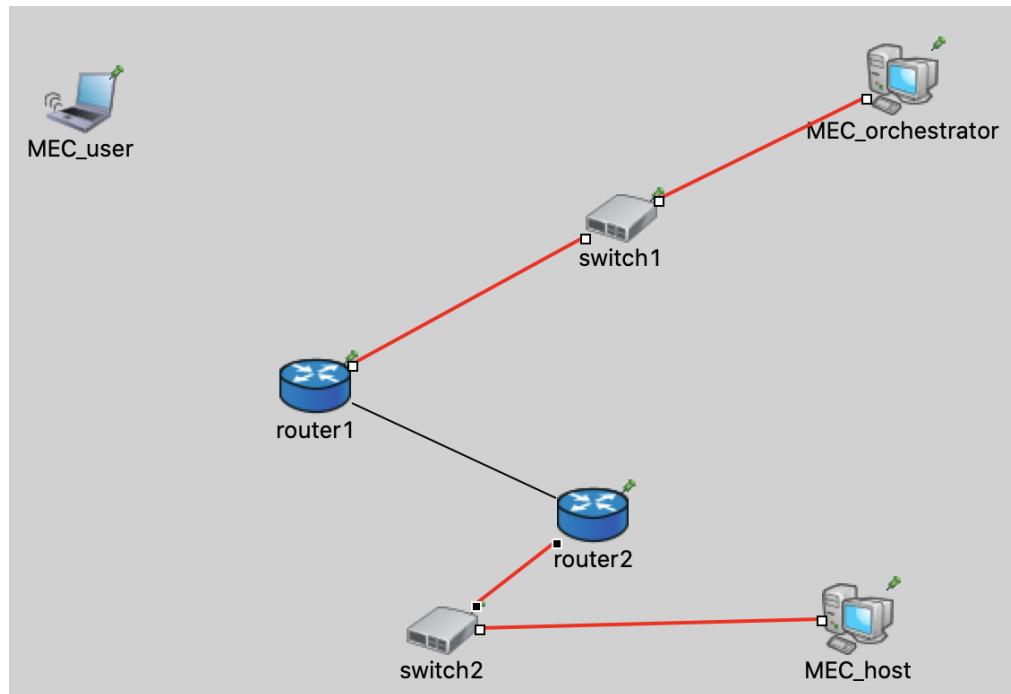


Figura 35: Channel

Come spiegato in precedenza, la struttura base del progetto è composta da un MEC orchestrator, un MEC host ed un MEC user collegati tra di loro tramite la rete. In Figura 35 si può notare la struttura della rete che è stata creata con i tre componenti. Più in particolare la figura è la rappresentazione grafica che OMNeT ++ mette a disposizione per ogni file NED.

4.2.1 Types

Nella rete vengono dichiarati due tipi di canali:

- **Channel**: è il canale utilizzato maggiormente come collegamento tra i nodi. Estende `DatarateChannel` e setta i parametri `delay` e `datarate` rispettivamente a 1ms e 10Mbps (Figura 35).
- **ChannelBackbone**: viene utilizzato tra i due router della rete, che sono collegate tramite PPP. Estende `DatarateChannel1` e setta i parametri `delay` e `datarate` rispettivamente a 10ms e 100Mbps (Figura 36).

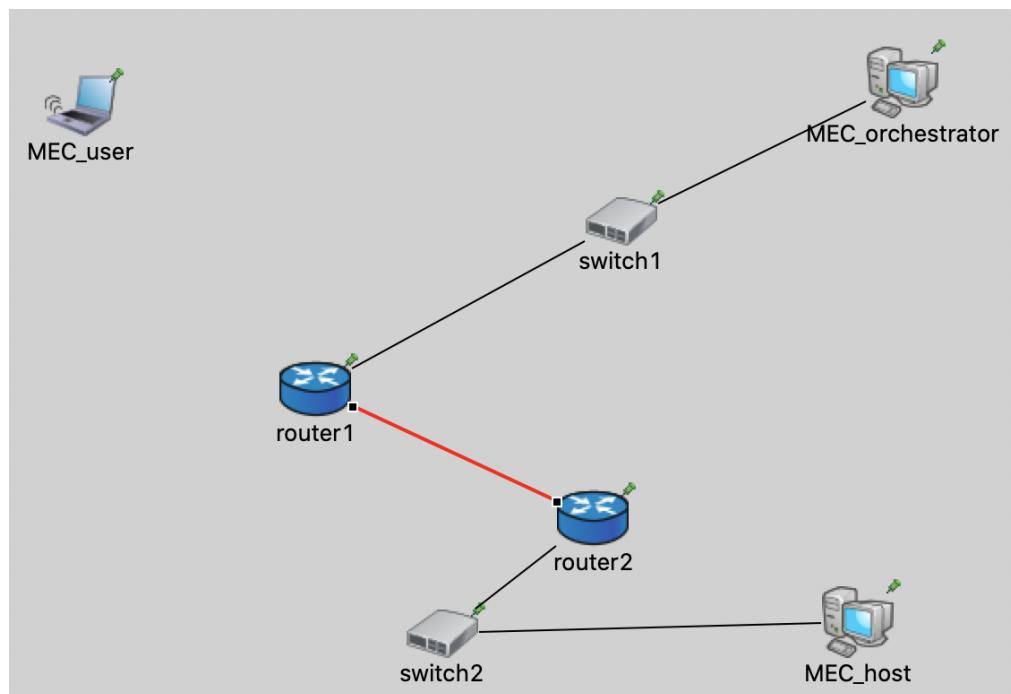


Figura 36: Channel Backbone

4.2.2 Submodules

submodules:

```

MEC_orchestrator: StandardHost {
    @display("p=628.02124,104.4925");
}
MEC_host: StandardHost {
    @display("p=538.45624,411.5725");
}
Mec_User: WirelessHost {
    @display("p=278.29126,89.565");
}

configurator: Ipv4NetworkConfigurator {
    @display("p=40.60875,52.0625");
}
visualizer: <default("IntegratedVisualizer")> like IIIntegratedVisualizer {

    @display("p=40.60875,137.445");
}
radioMedium: <default("UnitDiskRadioMedium")> like IRadioMedium {
    @display("p=40.60875,220.74501");
}

switch1: EtherSwitch {
    @display("p=458.48752,155.6725");
}
switch2: EtherSwitch {
    @display("p=339.0675,346.53125");
}
router1: Router {
    @display("p=339.0675,236.7075");
}
router2: Router {
    @display("p=548.0525,287.8875");
}

```

Figura 37: MEC.ned - submodules

In Figura 37 i sottomoduli che compongono la rete, che comprendono le entità MEC, i router, gli switch ed altri moduli ausiliari che INET mette a disposizione. Vengono utilizzati i moduli StandardHost, WirelessHost, Ipv4NetworkConfigurator, IntegratedVisualizer, UnitDiskRadioMedium, EtherSwitch e Router.

4.3 StandardHost

I componenti di nome `MEC_orchestrator` e `MEC_host` sono delle istanze di **StandardHost**, modulo che INET mette a disposizione e che rappresenta un host generico capace di interfacciarsi su una rete cablata e di processare pacchetti attraverso lo stack TCP/IP. StandardHost da l'inizio ad una serie di subclassing che principalmente ricoprono l'intero stack di protocolli TCP/IP. Per fare un esempio, il modulo che rappresenta il livello 4 (trasporto) estende il modulo che rappresenta il livello 3 (rete) e così via. In Figura 38 viene mostrato l'intero albero di subclassing partendo da StandardHost.

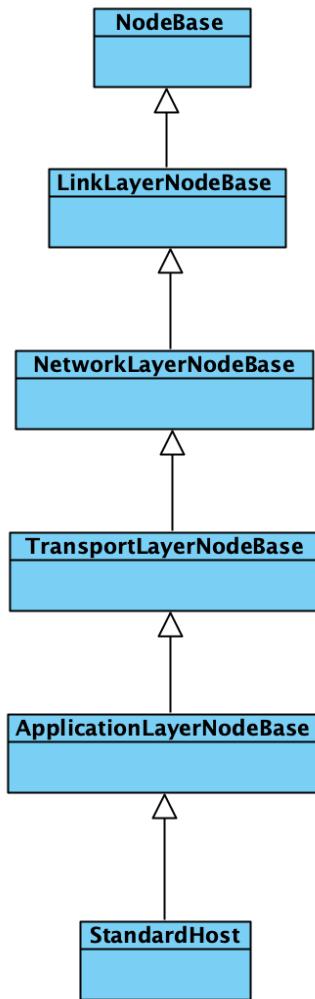


Figura 38: Subclassing di StandardHost

```

module ApplicationLayerNodeBase extends TransportLayerNodeBase
{
    parameters:
        int numApps = default(0);
        @figure[applicationLayer](type=rectangle; pos=250,6; size=1000,130;
        @figure[applicationLayer.title](type=text; pos=1245,11; anchor=ne;

    submodules:
        app[numApps]: <> like IApp {
            parameters:
                @display("p=375,76,row,150");
        }
        at: MessageDispatcher {
            parameters:
                @display("p=750,146;b=1000,5,,,1");
        }

    connections allowunconnected:
        for i=0..numApps-1 {
            app[i].socketOut --> at.in++;
            app[i].socketIn <-- at.out++;
        }

        at.out++ --> udp.appIn if hasUdp;
        at.in++ <-- udp.appOut if hasUdp;

        at.out++ --> tcp.appIn if hasTcp;
        at.in++ <-- tcp.appOut if hasTcp;

        at.out++ --> sctp.appIn if hasSctp;
        at.in++ <-- sctp.appOut if hasSctp;

        at.out++ --> tn.int++;
        at.in++ <-- tn.out++;
    }
}

```

Figura 39: ApplicationLayerNodeBase

È interessante analizzare il modulo **ApplicationLayerNodeBase**, in Figura 39. Possiede un vettore di submodules denominato **app** e la sua lunghezza è data dal parametro **numApps**. Le istanze del vettore devono implementare l'interfaccia **IApp** e queste saranno le applicazioni che effettivamente risiederanno nello StandardHost. Questi parametri saranno configurati nel file di configurazione della simulazione. La stessa caratteristica è posseduta dal componente **MEC_user**, che è un'istanza del modulo **WirelessHost**. Quest'ultimo non è altro che un'estensione di StandardHost. I componenti del progetto necessitano quindi di applicazioni e quasi tutto il resto del lavoro di rete è garantito da INET. Le funzionalità di gestione del framework ETSI MEC sono implementate come applicazioni che vengono eseguite (o meglio simulate) all'interno di StandardHost e WirelessHost.

4.4 MecControlApp

Il componente **MecControlApp** è l'implementazione della struttura base di un componente di controllo dell'architettura MEC. Esistono tre file all'interno del progetto che la rappresentano, ossia un file NED e due file C++ (.h e .cc). In realtà il comportamento del componente NED non è legato direttamente ai file C++ perché MecControlApp è stato appunto ideato come un entità generale, da estendere per implementare oggetti specifici. Esistono tre file nel progetto che descrivono MecControlApp:

- IMecControlApp.ned
- MecControlApp.h
- MecControlApp.cc

4.4.1 Modello

In Figura 40 è illustrata l'interfaccia **IMecControlApp**. L'interfaccia estende **IApp** e da contratto dichiara il parametro intero **listeningPort**, che verrà specificato nel file di configurazione.

```
import inet.applications.contract.IApp;

moduleinterface IMecControlApp extends IApp{

    parameters:
        int listeningPort;
}
```

Figura 40: IMecControlApp.ned

4.4.2 Implementazione

Come spiegato in precedenza, non è collegato un comportamento attivo al modulo MecControlApp, difatti la sua implementazione in NED è un'interfaccia. Sono stati comunque creati due file C++ che rappresentano una sorta di classe astratta per MecControlApp, che verrà estesa per realizzare i componenti *MEC orchestrator* e *MEC host*. I due file, **MecControlApp.h** e **MecControlApp.cc** sono rispettivamente un file di intestazione (.h) ed un file di implementazione (.cc) e contengono un set di metodi implementati.

```

class MecControlApp : public cSimpleModule, public inet::TcpSocket::ReceiveQueueBasedCallback{

private:

protected:
    inet::TcpSocket serverSocket;
    inet::SocketMap socketMap; // < id, socket > used for incoming messages
    map<string, inet::TcpSocket *> clientSocketMap; // < hostName, socket > used for outgoing messages
    const char *hostName;

    virtual int numInitStages() const override { return inet::NUM_INIT_STAGES; }
    virtual void initialize(int numstage) override;

    virtual void socketDataArrived(inet::TcpSocket *socket) override;
    virtual void socketAvailable(inet::TcpSocket *socket, inet::TcpAvailableInfo *availableInfo) override;
    virtual void socketEstablished(inet::TcpSocket *socket) override {};
    virtual void socketPeerClosed(inet::TcpSocket *socket) override {};
    virtual void socketClosed(inet::TcpSocket *socket) override {};
    virtual void socketFailure(inet::TcpSocket *socket, int code) override {};
    virtual void socketStatusArrived(inet::TcpSocket *socket, inet::TcpStatusInfo *status) override {};
    virtual void socketDeleted(inet::TcpSocket *socket) override {};

    virtual void handleMessage(cMessage *msg) override;

    virtual void addClientSocketToMap (inet::TcpSocket *newSocket, const char *dest);
    virtual void connectClientSocket(const char *address, int port);
    virtual void sendMecControlMessage(const char *dest, inet::Ptr<const MecControlMessage> message);

    template <typename T> inet::Ptr<T> createMecControlMessage(){
        static_assert(std::is_base_of<MecControlMessage, T>::value, "Not derived from MecControlMessage");
        auto message = inet::makeShared<T>();
        message->setSourceHost(hostName);
        message->setTimestamp(simTime().dbl());

        return message;
    }
}

```

Figura 41: *MecControlApp.h - parte 1*

La Figura 41 mostra una porzione di codice del file *MecControlApp.h*. La classe estende *cSimpleModule* e *TcpSocket::ReceiveQueueBasedCallback*, quest'ultima è una classe messa a disposizione da INET per simulare la ricezione dei messaggi di livello applicativo tramite le socket di sistema. Di seguito una descrizione dei parametri e dei metodi della classe:

- *TcpSocket serverSocket*: socket lato server, sulla quale si riceveranno richieste di connessione.
- *SocketMap socketMap*: una mappa messa a disposizione da INET; le coppie che contiene sono del tipo <id, socket>. Viene utilizzata in fase di ricezione, per trovare una determinata socket a partire dal messaggio ricevuto.
- *map<string, TcpSocket *> clientSocketMap*: è una mappa che contiene coppie del tipo <hostName, socket> e viene utilizzata in fase di invio. A partire dal nome del host che vogliamo contattare, ricaviamo la socket su cui inviare il messaggio.
- *const char * hostName*: nome del host; viene autoassegnato in fase di inizializzazione.

numInitStages()

Restituisce il numero di stage previsto da INET, cioè 13.

initialize (int numstage)

Questo metodo è ereditato da `cSimpleModule` e gestisce l'inizializzazione a livello applicazione. Per prima cosa assegna al parametro `hostName` il nome del modulo padre di `MecControlApp`, ossia il nome dello `StandardHost` che contiene l'applicazione. Successivamente avviene il setup per la `serverSocket`. È molto importante la riga di codice in cui viene settata la “Callback”: per come è implementata la socket, quando arrivano dei dati dal livello inferiore, viene chiamato il metodo `socketDataArrived` su una callback. In questo caso la callback è `MecControlApp` stessa, quindi i flussi in entrata vengono gestiti dal metodo di questa classe.

socketDataArrived (TcpSocket *socket)

Questo metodo è ereditato da `TcpSocket` e viene invocato quando arrivano dei pacchetti da una socket e li processa fino a che la coda dei messaggi in arrivo non è vuota. A questo punto, tramite uno switch sul tipo di messaggio viene invocato un determinato metodo di processamento.

socketAvailable (TcpSocket *socket, TcpAvailableInfo *availableInfo)

Metodo ereditato da `TcpSocket` che accetta connessioni TCP lato server e inserisce, tramite il metodo `addClientSocketToMap`, la nuova socket alle strutture dati della classe.

handleMessage (cMessage *msg)

Il metodo è ereditato da `cSimpleModule` e viene invocato quando arriva un messaggio al modulo. Il messaggio viene processato in maniera differente a seconda che sia un `selfMessage` oppure no. Nel secondo caso viene controllato se il messaggio proviene da una connessione TCP già aperta (identificando la socket desiderata tramite la `socketMap`) oppure se il messaggio è indirizzato alla `serverSocket`. Il metodo `handleMessage()` non è da confondere con `socketDataArrived()`: il primo può scaturire l'invocazione del secondo tramite la socket.

addClientSocketToMap (TcpSocket *newSocket, const char *dest)

Questo metodo prende come parametri una socket ed una stringa che identifica un host e popola le strutture dati di `MecControlApp`.

connectClientSocket(const char *address, int port)

Il metodo prende come parametri un nome di un host (`address`) ed un numero di porta (`port`). Viene utilizzato per creare una connessione verso un host. Per prima cosa crea una nuova socket e viene instaurata una connessione proprio con l'host di nome `address` alla porta `port`. Successivamente aggiunge la nuova socket alle strutture dati.

sendMecControlMessage(const char *dest, Ptr<const MecControlMessage> message)

Il metodo incapsula un messaggio di tipo `MecControlMessage` in un pacchetto di rete, successivamente lo inoltra su una determinata socket che identifica all'interno della `clientSocketMap` grazie al nome del destinatario, ossia il parametro `dest`.

createMecControlMessage()

Questo template crea e restituisce un messaggio di tipo `MecControlMessage`. Più precisamente il parametro che si aspetta è una classe derivate da `MecControlMessage`. Il template inserisce nel corpo del messaggio il source `hostName` ed il `timestamp`.

```
virtual void processMecUpdateMessage(inet::Ptr<const MecUpdateMessage> message, inet::TcpSocket *socket){
    throw invalid_argument( "MecControlApp can't process Mec Host Update Messages" );
}
virtual void processMecHelloMessage(inet::Ptr<const MecHelloMessage> message, inet::TcpSocket *socket){
    throw invalid_argument( "MecControlApp can't process Mec Host Hello Messages" );
}
virtual void processMecWelcomeMessage(inet::Ptr<const MecWelcomeMessage> message){
    throw invalid_argument( "MecControlApp can't process Mec Orchestrator Welcome Messages" );
}
virtual void processMecRequestServiceMessage(inet::Ptr<const MecRequestServiceMessage> message){
    throw invalid_argument( "MecControlApp can't process Mec Request Service Messages" );
}
virtual void processMecStartMecAppMessage(inet::Ptr<const MecStartMecAppMessage> message){
    throw invalid_argument( "MecControlApp can't process Mec Start MecApp Messages" );
}
virtual void processMecAppStartedMessage(inet::Ptr<const MecAppStartedMessage> message){
    throw invalid_argument( "MecControlApp can't process Mec App Started Messages" );
}
virtual void processSelfMessage(cMessage *msg){
    throw invalid_argument( "MecControlApp can't process a self message" );
}
```

Figura 42: `MecControlApp.h` - parte 2

La Figura 42 mostra la seconda porzione di codice del file `MecControlApp.h`. Questa parte di file è quella che rende la classe `MecControlApp` una sorta di classe astratta. La classe infatti è stata ideata per non implementare questi metodi, ma lasciare l'implementazione alle classi che la estendono. Questi metodi sono utilizzati ed implementati nelle classi `MecOrchestratorApp` e `MecPlatformApp`.

4.5 MecOrchestratorApp

Il componente **MecOrchestratorApp** rappresenta appunto l'orchestratore dell'architettura ETSI MEC. Come illustrato in Figura 43 il componente MecOrchestratorApp è pensato come estensione di **MecControlApp**, sia nel NED che nel codice C++. Esistono tre file che lo rappresentano:

- **MecOrchestratorApp.ned**
- **MecOrchestratorApp.h**
- **MecOrchestratorApp.cc**

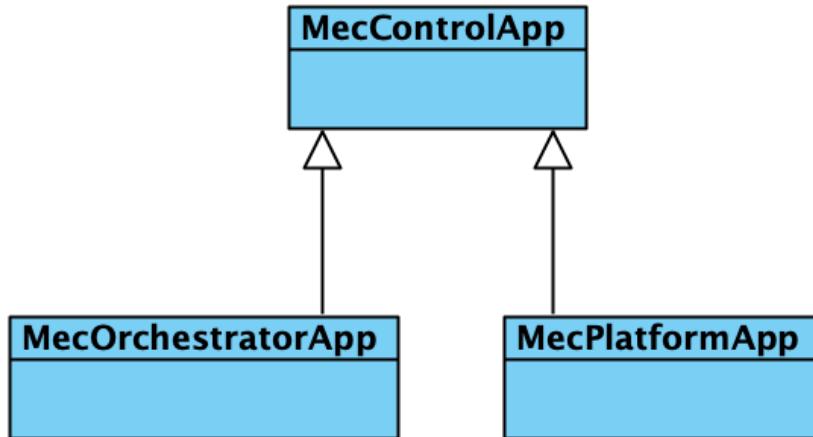


Figura 43: Subclassing di MecControlApp

4.5.1 Modello

La Figura 44 mostra il codice NED per il modulo MecOrchestratorApp, contenuto nel file **MecOrchestratorApp.ned**. Il modulo estende l'interfaccia **IMecControlApp**, descritta nella Sezione 4.4.1. Al parametro **listeningPort** viene assegnato come valore di default 1000. Vengono dichiarati altri parametri ai quali sono assegnati altri valori di default.

```

simple MecOrchestratorApp like IMecControlApp
{
    parameters:

        @class(MecOrchestratorApp);

        int listeningPort = default(1000);

        string connectAddress = default(""); // server address (may be symbolic)
        int connectPort = default(1000); // port number to connect to
        double updateRate @unit(s) = default(0s);

    gates:
        input socketIn @labels(Server/up);
        output socketOut @labels(Server/down);
}

```

Figura 44: MecOrchestratorApp.ned

4.5.2 Implementazione

La classe MecOrchestratorApp estende la classe MecControlApp e quindi eredita i suoi attributi e metodi. In Figura 46 il file MecOrchestratorApp.h. I metodi sono dichiarati qui ed implementati nel file cc. La classe presenta un attributo privato:

- `map<const char *, Host *> MECHosts`: una mappa in cui il MEC orchestrator registra eventuali MEC host quando questi si dichiarano. Il tipo Host è definito nel file MecControlApp.h e mostrato in Figura 45.

```

class Host {

    protected:
        const char *mechHostId;
        Resources cap;

    public:

        Host(const char *mechHostId, Resources cap){
            this->mechHostId = mechHostId;
            this->cap = cap;
        }

        friend ostream & operator << (ostream &out, const Host &obj) {
            return out << "ID: " << obj.mechHostId << "\tCPU: " << obj.cap.getCpu() << endl;
        }
};

```

Figura 45: Classe Host

```

#include "MecControlApp.h"

using namespace omnetpp;
using namespace std;

class MecOrchestratorApp: public MecControlApp{
private:
    map<const char *, Host*> MECHosts;

protected:
    virtual void processMecUpdateMessage/inet::Ptr<const MecUpdateMessage> message, inet::TcpSocket *socket) override;
    virtual void processMecHelloMessage/inet::Ptr<const MecHelloMessage> message, inet::TcpSocket *socket) override;
    virtual void processMecRequestServiceMessage/inet::Ptr<const MecRequestServiceMessage> message) override;
    virtual void processMecAppStartedMessage/inet::Ptr<const MecAppStartedMessage> message) override;
    virtual void processSelfMessage(cMessage *msg) override;
    virtual void handleApplication(MecAppDescription application, const char *serviceName);

public:
};

```

Figura 46: MecOrchestratorApp.h

processMecHelloMessage(Ptr<const MecHelloMessage> message, TcpSocket *socket)

Questo metodo viene invocato quando un MEC Host si dichiara al MEC orchestrator, che rileva le sue capabilities salva le informazioni nella mappa MECHosts. Successivamente gli risponde, comunicandogli il parametro updateRate.

processMecRequestServiceMessage(Ptr<const MecRequestServiceMessage> message)

Metodo che viene invocato quando il MEC user invia una richiesta di utilizzo di un servizio. Il MEC orchestrator prende visione delle applicazioni che fanno parte di quel servizio e le gestisce una dopo l'altra.

handleApplication(MecAppDescription application, const char *serviceName)

Il metodo viene chiamato per gestire l'attivazione di un'applicazione su un MEC host, creando un **MecStartMecAppMessage** ed inoltrando la richiesta al componente **MEC_host**.

processMecAppStartedMessage(Ptr<const MecAppStartedMessage> message)

Metodo che viene invocato quando un MEC host comunica all'orchestratore la conferma di istanziazione di un'applicazione. Successivamente il MEC orchestrator inoltra la conferma all'utente.

4.6 MecPlatformApp

Il componente **MecPlatformApp** rappresenta l’entità MEC platform, che risiede all’interno di un MEC host. La Figura 43, come già anticipato, mostra che il componente MecPlatformApp è un’estensione di **MecControlApp**, sia nel NED che nel codice C++. I file che descrivono il componente sono:

- `MecPlatformApp.ned`
- `MecPlatformApp.h`
- `MecPlatformApp.cc`

4.6.1 Modello

```
simple MecPlatformApp like IMecControlApp
{
    parameters:
        @class(MecPlatformApp);
        int listeningPort = default(1000);
        string orchestratorAddress = default("MEC_orchestrator");
        int orchestratorPort = default(1000);
        double startTime @unit(s) = default(0s);
    gates:
        input socketIn @labels(Server/up);
        output socketOut @labels(Server/down);
}
```

Figura 47: MecPlatformApp.ned

In Figura 47 il codice NED contenuto nel file `MecPlatformApp.ned`. Così come per `MecOrchestratorApp`, vengono definiti alcuni parametri di default. Una differenza è che in questo caso vengono specificati l’indirizzo simbolico e la porta dell’orchestratore, in quanto la MEC platform vorrà connettersi con quell’entità.

4.6.2 Implementazione

Anche in questa situazione la classe estende `MecControlApp`, ereditando metodi ed attributi. La Figura 48 mostra il codice contenuto nel file `MecPlatformApp.h`. I metodi sono implementati nel file cc, mentre qui vengono dichiarati tre attributi private:

- `cMessage *sendUpdate`
- `cMessage *sendResources`
- `int updateRate`

```
#include "MecControlApp.h"

using namespace omnetpp;
using namespace std;

class MecPlatformApp: public MecControlApp{

private:
    cMessage *sendUpdate;
    cMessage *sendResources;
    int updateRate;

protected:
    virtual void initialize(int numstage) override;
    virtual void processSelfMessage(cMessage *msg) override;
    virtual void processMecWelcomeMessage(inet::Ptr<const MecWelcomeMessage> message) override;
    virtual void processMecStartMecAppMessage(inet::Ptr<const MecStartMecAppMessage> message) override;

public:
};
```

Figura 48: MecPlatformApp.h"

`processMecWelcomeMessage(Ptr<const MecWelcomeMessage> message)`

Questo metodo viene invocato quando `MecOrchestratorApp` conferma la ricezione della dichiarazione del MEC host. In particolare l'orchestratore comunica alla `MecPlatformApp` l'`updateRate` con cui vuole essere aggiornato.

`processMecStartMecAppMessage(Ptr<const MecStartMecAppMessage> message)`

Il metodo viene invocato quando viene ricevuto un messaggio di `MecStartMecAppMessage` da parte del MEC orchestrator. Successivamente `MecPlatformApp` risponde confermando l'avvenuta istanziazione dell'applicazione.

initialize (int numstage)

MecPlatformApp esegue l'override del metodo `initialize`, richiamando però anche il metodo del padre. La classe inizializza i due `cMessage`, successivamente crea una connessione con il MEC orchestrator ed infine schedula il self message `sendResources`.

processSelfMessage(cMessage *msg)

Il metodo gestisce i vari self messages che sono stati auto-inviati. In ogni caso MecPlatformApp invia informazioni al MEC orchestrator. Nel primo caso si dichiara comunicando le sue risorse, mentre nel secondo invia degli aggiornamenti sul suo stato.

4.7 MecUser

Il componente **MecUser** rappresenta le *Device applications* e il *Customer facing service portal* del framework ETSI MEC. Nel progetto è rappresentato da tre file:

- `MecUser.ned`
- `MecUser.h`
- `MecUser.cc`

4.7.1 Modello

Il modulo MecUser definisce due parametri che serviranno per creare una connessione con **MEC_orchestrator**, ossia l'indirizzo simbolico (`orchestratorAddress`) e il numero della porta (`orchestratorPort`).

4.7.2 Implementazione

La classe MecUser dichiara i seguenti attributi e metodi:

- `TcpSocket orchestratorSocket`: socket lato client, tramite la quale si aprirà una connessione con il MEC Orchestrator.
- `const char * hostName`: nome del host; viene autoassegnato in fase di inizializzazione.
- `cMessage * requestService`

`initialize(int numstage)`

Durante la fase di inizializzazione viene assegnato il nome al parametro `hostName` e viene creato ed assegnato un messaggio al parametro `requestService`. Successivamente viene creata una connessione TCP con **MEC_orchestrator** ed infine viene schedulato il self message.

`socketDataArrived (TcpSocket *socket)`

Metodo ereditato da `TcpSocket` che viene invocato quando arrivano dei pacchetti da una socket, che processa fino a quando la coda dei messaggi in arrivo non è vuota. A questo punto, tramite uno switch sul tipo di messaggio viene invocato un determinato metodo di processamento.

handleMessage (cMessage *msg)

Il metodo è ereditato da `cSimpleModule` e viene invocato quando arriva un messaggio al modulo. Il messaggio viene processato in maniera differente a seconda che sia un `selfMessage` oppure sia un messaggio proveniente dalla `orchestratorSocket`.

sendMecControlMessage(Ptr<const MecControlMessage> message)

Il metodo incapsula un messaggio di tipo `MecControlMessage` in un pacchetto di rete e successivamente lo inoltra sulla `orchestratorSocket`.

createMecControlMessage()

Questo template crea e ritorna un messaggio di tipo `MecControlMessage`. Più precisamente il parametro che si aspetta è una classe che estende `MecControlMessage`. Il template inserisce nel corpo del messaggio il source `hostName` ed il `timestamp`.

processSelfMessage(cMessage *msg)

Questo metodo crea un messaggio di tipo `MecRequestServiceMessage` e lo invia al MEC orchestrator. Il messaggio rappresenta la richiesta di avvio di un servizio e contiene la descrizione di un applicazione e del servizio che la contiene.

processMecServiceStartedMessage (Ptr<const MecServiceStartedMessage> message)

Questo metodo viene invocato quando il MEC orchestrator notifica al MEC user l'avvenuta istanziazione di un servizio.

4.8 Messaggi

Le entità che compongono il progetto comunicano attraverso dei pacchetti di rete, incapsulati secondo lo stack TCP/IP grazie al framework INET. I messaggi a livello applicazione sono definiti nella cartella del progetto `control_plane`, più precisamente nel file `MecControlMessages.msg`.

```

enum ControlMessageType{
    MEC_HELLO_MESSAGE = 0;                                // platform -> orchestrator
    MEC_STATUS_UPDATE = 1;                                // platform -> orchestrator
    MEC_WELCOME_MESSAGE = 2;                             // orchestrator -> platform
    MEC_REQUEST_SERVICE_MESSAGE = 3;                      // user -> orchestrator
    MEC_START_MEC_APP_MESSAGE = 4;                        // orchestrator -> platform
    MEC_APP_STARTED_MESSAGE = 5;                          // platform -> orchestrator
    MEC_SERVICE_STARTED_MESSAGE = 6;                     // orchestrator -> user
};

class Resources {
    int cpu;
    int ram;
    int disk;
    int network;
}

class QoSRequirements {
    double expectedDelay;
    double bandwidth;
    double processingTime;
}

class MecAppDescription {
    string appName;
    Resources requiredResources;
    QoSRequirements qosRequirements;
}

class MecControlMessage extends inet::FieldsChunk{
    long messageId;
    string sourceHost;
    ControlMessageType ctlMsgType;
    double timestamp;
}

```

Figura 49: MecControlMessages.msg - parte 1

In Figura 49 viene mostrato il contenuto in `MecControlMessages.msg`. Per prima cosa viene dichiarato un `enum` grazie al quale sarà possibile identificare le varie tipologie di messaggi. Successivamente vengono dichiarate tre classi (`Resources`, `QoSRequirements` e `MecAppDescription`) che serviranno per descrivere le applicazioni da eseguire. Infine viene dichiarata la classe principale dei messaggi, ossia `MecControlMessage`, che estende `FieldsChunk` di INET.

```

// platform -> orchestrator
class MecUpdateMessage extends MecControlMessage{
    chunkLength = inet::B(400);
    ctlMsgType = MEC_STATUS_UPDATE;
    double memoryCapacityUsed;
    double computationalCapacityUsed;
    double storageCapacityUsed;
    double networkCapacityUsed;
};

// platform -> orchestrator
class MecHelloMessage extends MecControlMessage{
    chunkLength = inet::B(200);
    ctlMsgType = MEC_HELLO_MESSAGE;
    Resources capabilities;
}

// orchestrator -> platform
class MecWelcomeMessage extends MecControlMessage{
    chunkLength = inet::B(200);
    ctlMsgType = MEC_WELCOME_MESSAGE;
    double updateRate;
}

// user -> orchestrator
class MecRequestServiceMessage extends MecControlMessage{
    chunkLength = inet::B(200);
    ctlMsgType = MEC_REQUEST_SERVICE_MESSAGE;
    string serviceName;
    MecAppDescription mecApplications[];
}

// orchestrator -> platform
class MecStartMecAppMessage extends MecControlMessage{
    chunkLength = inet::B(200);
    ctlMsgType = MEC_START_MEC_APP_MESSAGE;
    string serviceName;
    MecAppDescription mecApplication;
}

// platform -> orchestrator
class MecAppStartedMessage extends MecControlMessage{
    chunkLength = inet::B(200);
    ctlMsgType = MEC_APP_STARTED_MESSAGE;
    string serviceName;
    string appName;
}

// orchestrator -> user
class MecServiceStartedMessage extends MecControlMessage{
    chunkLength = inet::B(200);
    ctlMsgType = MEC_SERVICE_STARTED_MESSAGE;
    string serviceName;
}

```

Figura 50: *MecControlMessages.msg* - parte 2

In Figura 50 vengono illustrate le classi che descrivono i vari messaggi, contenute nel file *MecControlMessages.msg*. Questi messaggi, di livello applicazione, vengono utilizzati per la comunicazione tra i componenti del progetto. Ogni classe è un'estensione di *MecControlMessage*.

4.9 Configurazione e Simulazione

```
[Config MEC]

description = "ETSI MEC - Orchestrator & MEC Host"
network = MEC
sim-time-limit = 600s
#####
# MEC ORCHESTRATOR
*.MEC_orchestrator.numApps = 1
*.MEC_orchestrator.app[0].typename = "MecOrchestratorApp"
*.MEC_orchestrator.app[0].listeningPort = 50001
*.MEC_orchestrator.app[0].updateRate = 2s

# MEC PLATFORM
*.MEC_host.numApps = 1
*.MEC_host.app[0].typename = "MecPlatformApp"
*.MEC_host.app[0].listeningPort = 50002
*.MEC_host.app[0].orchestratorPort = 50001

# MEC USER
*.MEC_user.numApps = 1
*.MEC_user.app[0].typename = "MecUser"
*.MEC_user.app[0].orchestratorPort = 50001
*.MEC_user.app[0].startTime = 1s

*.MEC_user.wlan[0].typename = "AckingWirelessInterface"
*.MEC_user.wlan[0].mac.useAck = false
*.MEC_user.wlan[0].mac.fullDuplex = false
*.MEC_user.wlan[0].radio.transmitter.communicationRange = 500m
*.MEC_user.wlan[0].radio.receiver.ignoreInterference = true
*.MEC_user.wlan[0].mac.headerLength = 23B
*.MEC_user.**.bitrate = 1Mbps
#####

# ROUTER 1
*.router1.numWlanInterfaces = 1
*.router1.wlan[0].typename = "AckingWirelessInterface"
*.router1.wlan[0].mac.useAck = false
*.router1.wlan[0].mac.fullDuplex = false
*.router1.wlan[0].radio.transmitter.communicationRange = 500m
*.router1.wlan[0].radio.receiver.ignoreInterference = true
*.router1.wlan[0].mac.headerLength = 23B
*.router1.**.bitrate = 1Mbps

# VISUALIZER
*.visualizer.*.interfaceTableVisualizer.displayInterfaceTables = true
*.visualizer.*.interfaceTableVisualizer.interfaceFilter = "not(%o*)"
*.visualizer.*.interfaceTableVisualizer.nodeFilter = "not(%oswitch*)"
*.visualizer.*.interfaceTableVisualizer.displayWiredInterfacesAtConnections = false
*.visualizer.*.interfaceTableVisualizer.format = "%N %n"
```

Figura 51: omnetpp.ini

La configurazione della simulazione avviene grazie al file `omnetpp.ini`, mostrato nella Figura 51. Le prime tre righe specificano una descrizione, la rete da simulare ed il tempo massimo di simulazione.

Successivamente vengono configurati i vari componenti della rete:

- **MEC_ordinator**: lo StandardHost che implementa l'orchestratore contiene un'applicazione, ossia `MecOrchestratorApp`. La porta di ascolto è la 50001 e l'update rate è settato a 2 secondi.
- **MEC_host**: lo StandardHost che implementa il MEC host contiene un'applicazione, ossia `MecPlatformApp`. La porta di ascolto è la 50002, mentre la porta sulla quale stabilire una connessione con il MEC orchestrator è la 50001.
- **MEC_user**: il WirelessHost che implementa il MEC user contiene un'applicazione, ossia `MecUser`. La porta sulla quale stabilire una connessione con il MEC orchestrator è la 50001 e lo start time è settato a 1 secondo. Le successive righe del MEC user riguardano configurazioni del modulo per il corretto funzionamento dell'interfaccia wireless.
- **router1**: configurazioni del router per il corretto funzionamento dell'interfaccia wireless per dialogare con il MEC user.
- **visualizer**: configurazioni per la visualizzazione di informazioni durante la simulazione, come gli indirizzi IP dei nodi.

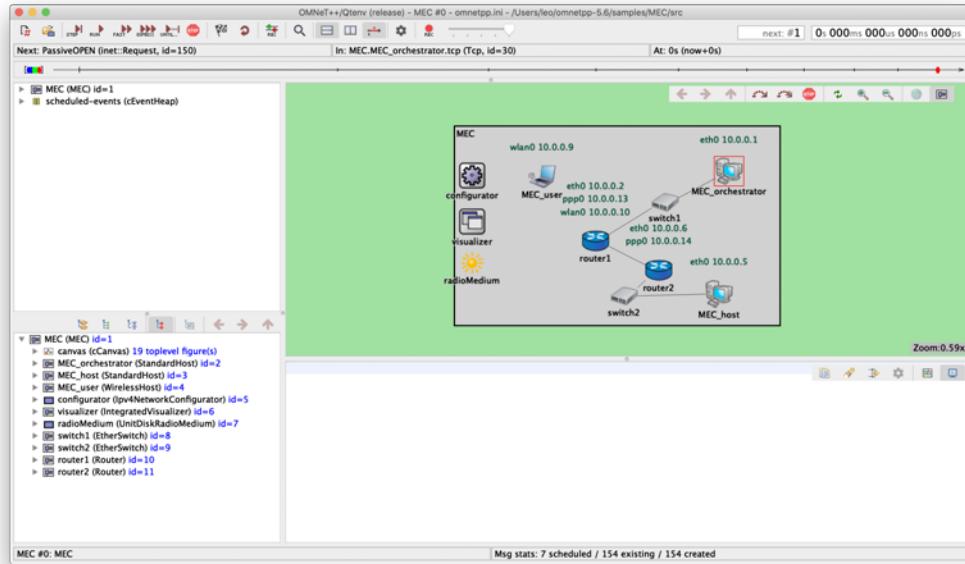


Figura 52: Rete della simulazione

```

** Event #5 t=0 MEC.MEC_host.app[0] (MecPlatformApp, id=68) on selfmsg Send resources (omnetpp::cMessage)
INFO:MEC PLATFORM send resources
** Event #142 t=0.010735613636 MEC.MEC_user.app[0] (MecUser, id=104) on ESTABLISHED (inet::Indication, id=298)
** Event #167 t=0.013451517045 MEC.MEC_orchestrator.app[0] (MecOrchestratorApp, id=32) on AVAILABLE (inet::Indication, id=318)
** Event #169 t=0.013451517045 MEC.MEC_orchestrator.app[0] (MecOrchestratorApp, id=32) on ESTABLISHED (inet::Indication, id=32)
** Event #224 t=0.03269936 MEC.MEC_host.app[0] (MecPlatformApp, id=68) on ESTABLISHED (inet::Indication, id=352)
** Event #258 t=0.04757032 MEC.MEC_orchestrator.app[0] (MecOrchestratorApp, id=32) on AVAILABLE (inet::Indication, id=366)
** Event #262 t=0.04757032 MEC.MEC_orchestrator.app[0] (MecOrchestratorApp, id=32) on ESTABLISHED (inet::Indication, id=369)
** Event #263 t=0.04757032 MEC.MEC_orchestrator.app[0] (MecOrchestratorApp, id=32) on data (inet::Packet, id=370)
INFO:Rilevato MEC HOST: MEC_host CPU:10 RAM:20 Disk:30 Network:40
INFO:MEC ORCHESTRATOR send welcome message
** Event #331 t=0.06250848 MEC.MEC_host.app[0] (MecPlatformApp, id=68) on data (inet::Packet, id=399)
INFO:MEC PLATFORM: received welcome message from orchestrator
INFO:MEC PLATFORM: received updateRate of: 2s
** Event #366 t=1 MEC.MEC_user.app[0] (MecUser, id=104) on selfmsg Request Service (omnetpp::cMessage, id=58)
INFO:USER requires service
** Event #391 t=1.004626303409 MEC.MEC_orchestrator.app[0] (MecOrchestratorApp, id=32) on data (inet::Packet, id=433)
INFO:Service Required by User: Service
INFO:Application Name: Application
INFO:Required Resources: CPU:10 RAM:20 Disk:30 Network:40
INFO:Required QoS: Expected Delay: 0.1 Bandwidth: 10 Processing Time: 0.2
INFO:MEC ORCHESTRATOR has sent a mec start meccapp message to mec host
** Event #452 t=1.019564463409 MEC.MEC_host.app[0] (MecPlatformApp, id=68) on data (inet::Packet, id=469)
INFO:MEC PLATFORM has received the mec start meccapp message
** Event #522 t=1.034502623409 MEC.MEC_orchestrator.app[0] (MecOrchestratorApp, id=32) on data (inet::Packet, id=499)
INFO:MEC ORCHESTRATOR got that the MecApp has been started
** Event #564 t=1.039196126818 MEC.MEC_user.app[0] (MecUser, id=104) on data (inet::Packet, id=527)
INFO:USER got that the service with name Service has been started
** Event #607 t=2.06250848 MEC.MEC_host.app[0] (MecPlatformApp, id=68) on selfmsg Send event (omnetpp::cMessage, id=38)
INFO:MEC PLATFORM send update message
** Event #643 t=2.07803544 MEC.MEC_orchestrator.app[0] (MecOrchestratorApp, id=32) on data (inet::Packet, id=569)
INFO:MEC ORCHESTRATOR: received update

```

Figura 53: Console della simulazione

In Figura 52 e Figura 53 vengono mostrate rispettivamente la topologia di rete della simulazione prima dell'avvio e la console di log dopo circa 3 secondi dall'inizio. Ai componenti della rete è stato assegnato un indirizzo IP dal configuratore. Nella console possiamo vedere il flusso di messaggi che rappresenta la comunicazione tra le entità. Per prima cosa la MEC platform si dichiara al MEC orchestrator, che registra le sue informazioni e comunica nella risposta l'update rate. Successivamente il MEC user richiede un servizio composto da un'applicazione. Il MEC orchestrator istruisce la MEC platform per gestire l'avvio dell'applicazione e, quando questa è stata istanziata, comunica la conferma di avvio al MEC user. I dati ed i valori assegnati ai parametri del servizio e dell'applicazione sono simbolici.

Conclusioni

Durante questo lavoro di tesi è stata ricreata l'infrastruttura del framework ETSI MEC all'interno di un ambiente di simulazione, concentrandosi sul piano di controllo. Una prima parte del lavoro di tesi è stata dedicata allo studio dell'Edge Computing e del framework ETSI MEC, per poi passare alla comprensione del simulatore OMNeT ++ e di INET. Successivamente l'obbiettivo è stato quello di creare delle semplici topologie di rete, come una struttura client-server, utilizzando opportunamente i componenti che INET mette a disposizione ed estendendoli correttamente in maniera da ospitare delle specifiche applicazioni. Una volta appresa la capacità di modellare delle topologie di rete a piacere, il lavoro si è focalizzato sulla creazione dell'architettura ETSI MEC all'interno dell'ambiente di simulazione, cercando di perfezionare sempre di più il sistema in ottica di ricreare una reale situazione di funzionamento dell'infrastruttura MEC.

Una parentesi sui possibili sviluppi futuri del progetto in relazione al punto di partenza iniziale, agli obbietti predisposti ed in relazione al punto di arrivo pervenuto. Il progetto propone una struttura base di partenza per la simulazione dell'architettura ETSI MEC nel simulatore OMNeT ++, con particolare enfasi sui processi di gestione del sistema e sul componente *MEC orchestrator*. Il progetto può essere complicato e perfezionato, rendendo l'infrastruttura sempre più simile all'intera architettura ETSI MEC che lavora in un ambiente reale. Per prima cosa la rete può essere arricchita aggiungendo ulteriori MEC user e MEC host. Inoltre, se il numero di nodi cresce, allora è legittimo pensare che il MEC orchestrator debba adottare qualche metodo di ragionamento per svolgere determinate funzioni, come l'assegnamento dell'esecuzione di un'applicazione ad un determinato MEC host. In generale l'orchestratore potrebbe essere arricchito con altre strutture dati di controllo per avere una visione sempre più dettagliata dell'infrastruttura. Per esempio, un ulteriore sviluppo è legato alla gestione dello stato degli altri componenti del sistema, in particolare dello stato delle applicazioni e dei servizi richiesti dagli utenti. Parlando di servizi ed applicazioni, anche la richiesta di un servizio da parte di un MEC user può essere sviluppata: il

messaggio di richiesta può essere composto da più di un'applicazione ed è possibile specificare un'applicazione entry point del servizio. Infine l'orchestratore deve essere in grado di effettuare delle migrazioni di applicazioni da una MEC platform ad un'altra, sulla base dello stato del sistema e sulla base di vincoli fisici, geografici e qualitativi del servizio che si vuole offrire all'utente.

Bibliografia

- [1] “Multi-access Edge Computing (MEC).” [Online]. Available: <https://www.etsi.org/technologies/multi-access-edge-computing>
- [2] ETSI GS MEC 003, “Multi-access Edge Computing (MEC); Framework and Reference Architecture,” ETSI, Tech. Rep., January 2019.
- [3] “OMNeT ++ Discrete Event Simulator.” [Online]. Available: <https://omnetpp.orgc>
- [4] “INET Framework.” [Online]. Available: <https://inet.omnetpp.org>