Università degli studi di Milano

Facoltà di Scienze e Tecnologie

Corso di Laurea in Informatica

Valutazione delle prestazioni di un servizio microcloud di task offloading in vehicular edge computing

Relatore: Christian Quadri

Tesi di:

Luca Parenti

Matricola: **909049**

Anno Accademico 2022-2023

Per Mamma, Papà ed Andre.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Con la rapida evoluzione degli *smart vehicles* ed i recenti progressi nelle tecnologie di comunicazione è stato reso possibile un nuovo approccio alle reti di comunicazione di traffico veicolare, in cui i veicoli stessi rappresentano dei veri e propri nodi di elaborazione. Questi ultimi possono essere sfruttati per la computazione di task, più o meno difficili, tenendo sempre in considerazione le loro risorse limitate. Ciò ha il vantaggio di portare la computazione molto vicina all'edge della rete, con una conseguente diminuzione dei tempi di latenza, consentendo lo svolgimento di compiti che richiedono una risposta quasi real-time. Tutto ciò ha dato vita ad un nuovo modello di computazione: il Vehicular Edge Computing (VEC). Questo modello ha l'effetto di portare le risorse di computazione molto più in prossimità al luogo di generazione del dato rispetto ad altri paradigmi di computazione, come ad esempio il cloud computing. Tuttavia, va considerato che nelle reti di traffico veicolare, la mobilità rappresenta un fattore determinante per la stabilità delle connessioni inter-veicolari, oltre che la richiesta di protocolli specifici, data la presenza di modalità di accesso al canale distribuite e non coordinate.

Il *task offloading* è un servizio piuttosto diffuso ed utilizzato nel VEC, che ha lo scopo di distribuire il carico di lavoro e delegare la computazione dei dati da chi genera il task ai nodi adiacenti presenti in rete (in questo caso i veicoli). Tuttavia

la mobilità e le ridotte capacità computazionali dei veicoli rappresentano una grande sfida per la realizzazione di un servizio *efficiente* e *robusto*.

In questa tesi è stato implementato un servizio di task offloading ad-hoc per il paradigma di computazione di VEC che sfrutta la comunicazione di tipo Vehicle-to-Vehicle (V2V), con l'utilizzo dello stack di protocolli `WAVE` e degli standard `802.11p` e `1609.4`. Durante la progettazione del protocollo, è stato tenuto conto delle diverse situazioni in un ambiente dinamico ed in costante evoluzione come quello di VEC. L'obiettivo di questo lavoro di tesi è quello di fornire un'analisi preliminare sull'efficacia e la robustezza del protocollo di task offloading ideato.

Al fine di valutare le prestazioni del servizio offerto sono stati utilizzati software di simulazione quali OMNeT++, SUMO ed il framework Veins_INET. Il primo offre un ambiente di simulazione di eventi discreti ad eventi utilizzato per modellare reti di comunicazione di svariato genere. Il secondo, SUMO, è un simulatore di traffico e mobilità urbana, mentre Veins_INET permette l'integrazione dei due software precedentemente elencati, fornendo un ambiente di simulazione completo di tutto lo stack di protocolli di rete che sfrutta il paradigma di comunicazione V2V.

Nelle diverse simulazioni eseguite sono stati configurati 4 principali scenari che variano la dimensione di task, la capacità di calcolo del veicolo ed il tempo di generazione del task. Dai risultati ottenuti, è stato evidenziato che all'aumentare della dimensione del task, aumenta anche il tempo richiesto per la computazione ad un tasso molto più alto di quanto ci si aspetti. Questo accade perché l'accesso al canale di comunicazione avviene tramite un algoritmo di contesa non coordinato, utilizzato dallo standard `802.11p`. Ciò garantisce fairness per l'accesso al canale radio, ma causa un tempo variabile e non predicibile del tempo di attesa per l'accesso a quest'ultimo. Questo fenomeno aumenta all'aumentare della dimensione del task, provocando così anche un tempo totale di trasmissione più alto.

La seguente tesi è così strutturata:

- nel Capitolo 1 verranno introdotti i diversi paradigmi di computazione, gli standard ed i protocolli che permettono l'implementazione del servizio di task offloading

- nel Capitolo 2 verrà presentato l'intero protocollo di task offloading

- nel Capitolo 3 verranno presentati i diversi strumenti software e framework utilizzati per lo sviluppo del protocollo e l'ambiente di simulazione

- nel Capitolo 4 verranno presentati ed analizzati i risultati ottenuti nelle diverse simulazioni

Capitolo 1

Letteratura

L'Internet of Vehicles (IoV) è una tecnologia derivata dall'Internet of Things (IoT) e rappresenta un modo efficace di realizzare un sistema di trasporto intelligente (ITS). Ciò avviene connettendo veicoli oppure altre strutture alla rete, collezionando e processando informazioni derivanti dalle unità presenti in strada, per avere dei feedback in real-time e servizi di supporto.

Con la rapida evoluzione delle reti wireless e dell'intelligenza artificiale, i veicoli hanno iniziato ad essere sempre più interconnessi ed "intelligenti" nell'era dell'IoT. Il trend delle reti veicolari è in rapida evoluzione e ciò ha portato ad avere sempre più applicazioni delay-sensitive, cioè molto sensibili alla latenza ed ai ritardi dovuti alla grande quantità di dati da processare. Queste applicazioni possono essere di svariato genere, come ad esempio applicazioni di controllo intelligente del traffico, piuttosto che guida autonoma, fino ad arrivare ad applicazioni di navigazione real-time assistita.

In IoV ci sono due principali schemi di comunicazione wireless sotto il controllo del sistema di rete cellulare: Vehicle to Infrastructure (V2I) e Vehicle to Vehilce (V2V). Il Vehicular Edge Computing (VEC) è una promettente soluzione al problema di sensibilità che propone di scaricare tutto (o parzialmente) il carico computazionale dai veicoli ai server presenti sulle Road Side Unit (RSU) presenti a bordo strada (V2I) oppure ad altri veicoli che hanno disponibilità di risorse computazionali extra (V2V).

La tecnica appena descritta di delegare altri veicoli o alle infrastrutture presenti a bordo strada alla computazione di un dato è chiamato *task offloading*.

Sfruttando quindi la tecnica di task offloading, un generico veicolo può: sia svolgere altri compiti (o task) occupando solo parzialmente le sue capacità computazionali, sia permettersi di elaborare dati molto più grandi rispetto alle sue scarse risorse di computazione delegando ad altri parte della computazione.

1.1 Edge Computing e Vehicular Edge Computing

In questa sezione verranno introdotti i paradigmi di computazione di edge computing e vehicular edge computing, analizzando quali punti di forza e svantaggi essi portano all'interno del panorama computazionale moderno.

1.1.1 Edge Computing

Prima di poter approfondire l'argomento dell'edge computing è opportuno trattare l'IoT e fare una piccola introduzione al cloud computing.

Il cloud computing è un paradigma che offre servizi on-demand agli utenti finali tramite un pool di risorse di computazione che includono, ad esempio, servizi di storage oppure di processamento di dati. Va tenuto conto però del fatto che la disponibilità e la qualità di servizi cloud all'interno della rete internet dipende molto dalla distanza di instradamento multi-hop tra l'utente finale ed il server cloud, ciò significa che più è grande la distanza tra il dispositivo che richiede i dati ed il server, più saranno alti il tempo di computazione e la latenza.

L'IoT rappresenta una rete di dispositivi fisici connessi tra di loro che scambiano dati ed informazioni attraverso la rete internet. Questi dispositivi possono essere di svariato genere e legati a svariati ambiti, passiamo ad esempio da comuni oggetti

presenti nella vita quotidiana delle persone (smartphone, TV, PC, ...) fino ad arrivare a sofisticati strumenti industriali. Inoltre porta con se alcuni vantaggi, tra i quali:

- una maggiore efficienza, grazie alla possibilità di raccogliere ed analizzare dati in tempo reale
- una considerevole riduzione dei costi, dato che possono essere ridotti notevolmente i consumi di energia
- un livello di sicurezza più alto

Attualmente quasi tutte le applicazioni IoT sviluppate per un utilizzo in cloud come ad esempio quelle di controllo del traffico real-time oppure di realtà virtuale richiedono tempi di processamento e di risposta molto rapidi. Queste applicazioni di norma vengono eseguite sui dispositivi dell'utente finale utilizzando le loro risorse limitate, mentre l'erogazione del servizio ed il processamento delle informazioni e dei dati vengono eseguiti sui server in cloud.

L'edge computing porta con se servizi ed utilità comuni al cloud computing e li rende più accessibili e vicini all'utente finale. Le sue principali caratteristiche sono: una computazione molto veloce e di tempi di risposta delle applicazioni molto ridotti rispetto, ad esempio, alla computazione in cloud. Inoltre, a differenza del cloud computing, direziona i dati da computare, le applicazioni ed i servizi erogati dai server cloud ai nodi presenti alle estremità della rete (da qui edge computing). I fornitori di contenuti e gli sviluppatori di applicazioni possono quindi utilizzare i sistemi di computazione situati ai bordi della rete per portare più vicino possibile gli stessi servizi offerti dal cloud computing agli utenti finali [1].

Integrando quindi i concetti di IoT e, sapendo inoltre, che gli stessi servizi offerti dal cloud computing possono essere portati molto vicini a dove avviene la generazione dei dati da computare, grazie all'edge computing si ottengono tempi di computazione

quasi istantanei e latenze molto basse, sviluppando un'architettura di rete visualizzabile in Figura 1. Ciò ci permette di ottenere soluzioni innovative con latenze e tempi di computazione molto bassi per diversi ambiti, tra cui quello che interessa maggiormente questo lavoro di tesi: le reti di traffico veicolare.

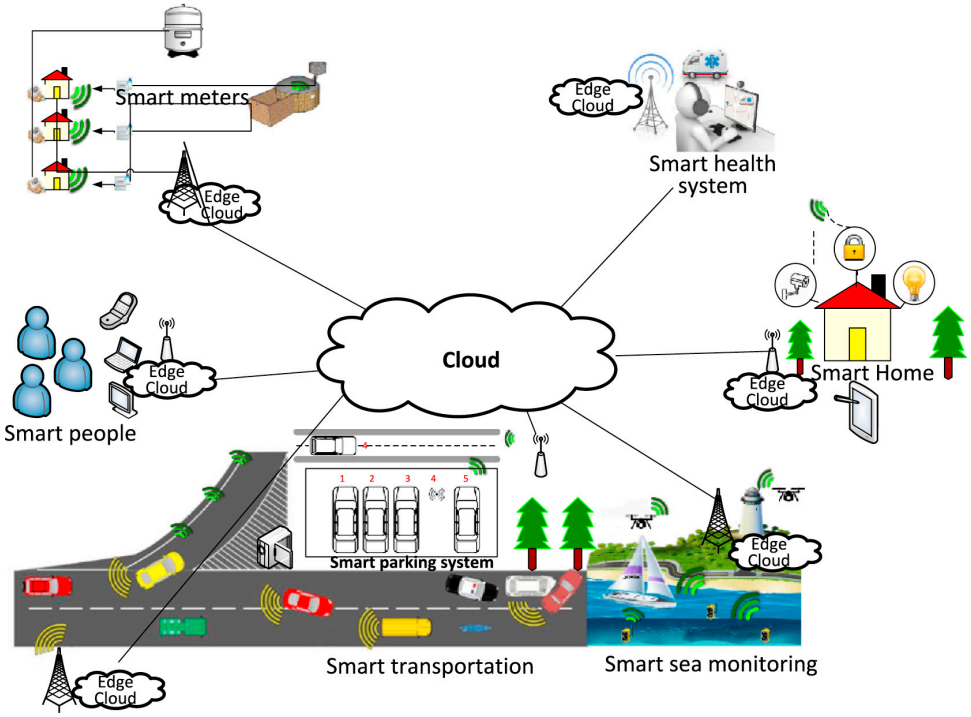

Figura 1: *Architettura edge computing*

Oltre agli effetti positivi elencati sopra che questo paradigma possiede, è doveroso evidenziare il fatto che le risorse di computazione dei nodi di rete connessi nelle vicinanze verrebbero sfruttate appieno, invece di essere sfruttate parzialmente o solamente come nodi di interconnessione per il passaggio di informazioni.

1.1.2 Vehicular Edge Computing

L'obiettivo delle reti veicolari è quello di costruire una rete di comunicazione inter-veicolo che sia aperta, poco costosa e semplice da distribuire. Al giorno d'oggi, le

reti veicolari costituiscono una parte importante dei diversi sistemi di trasporto, rendendoli più intelligenti, sicuri e convenienti. Però, con l'avvento di nuove applicazioni computazionalmente sempre più complicate e sensibili alla latenza, viene posta un'importante sfida alle reti veicolari già esistenti.

Le tecnologie emergenti di edge computing, come il mobile edge computing (MEC), possono rappresentare una soluzione a questo problema. I principali punti di forza del MEC sono la prossimità, la bassa latenza e la consapevolezza della posizione. Ciò si ottiene posizionando server MEC alle estremità della rete, in modo tale che la computazione stessa di un generico task può essere processata molto vicino a dove il dato da computare viene generato. Questo permette anche una generale ottimizzazione dell'esperienza utente, facendo si che quest'ultimo ottenga risultati quasi istantanei per i task richiesti [2].

Le performance della rete in vehicular edge computing (VEC), grazie all'introduzione dei server MEC, sono migliorate significativamente, ma si è dovuto tenere conto di una gestione complessa e gerarchica dell'utilizzo combinato di RSU, veicoli e MEC servers. Tuttavia sono state considerate le seguenti linee guida per lo sviluppo del VEC:

- scalabilità

- ottimizzazione delle performance energetiche

- supporto alla mobilità

Con scalabilità si intende la capacità della rete di mantenere delle prestazioni adeguate, anche in casi in cui il numero di veicoli dovesse aumentare di molto o subire delle variazioni importanti in brevi lassi di tempo. Come conseguenza il sistema di rete deve adattare ed adeguare la rapidità di calcolo in base a come evolve l'ambiente ed anche gestire la presenza eventuale di diverse computazioni che i veicoli stessi devono svolgere.

Con ottimizzazione delle performance energetiche si intende il mantenimento di un basso consumo di energia ed una bassa latenza. Questo fattore è importante per quanto riguarda le emissioni di carbonio dei veicoli. Consumo di energia all'interno di un MEC server di un veicolo significa, generalmente, consumo generale di gasolio o benzina e di conseguenza, mantenere un basso consumo di energia significa mantenere un basso valore di emissioni di carbonio.

Infine il supporto alla mobilità è una importante caratteristica che un sistema di rete veicolare deve possedere per il Quality of Service (QoS). Negli scenari di VEC i veicoli si muovono rapidamente e la topologia di rete deve cambiare rapidamente per adattarsi ai loro spostamenti. Di conseguenza la connessione tra le RSU ed i veicoli, oppure tra veicoli ed altri veicoli, non è stabile. Per questo motivo è richiesta una maggiore sensibilità al supporto alla mobilità dei veicoli.

Basandosi su queste linee guida l'architettura del VEC, osservabile in Figura 2, contiene diversi MEC servers, RSU e veicoli. Essendo le risorse dei veicoli abbastanza limitate, essi possono fare *offloading*, ossia delegare la computazione ad altri veicoli oppure alle RSU presenti a bordo strada.

Cooperative computing

Basandosi su edge computing e vehicular edge computing è stato introdotto un altro tipo di paradigma di computazione chiamato *cooperative computing*. Essa sfrutta i concetti principali precedentemente elencati di entrambe le modalità di computazione, aggiungendo *resilienza* e *flessibilità* distribuendo i task in modo dinamico, per far si che vengano raggiunti requisiti imposti di latenza e responsività all'interno della computazione. Questi requisiti possono essere diversi, un esempio potrebbe essere il completamento di un task entro un certo tempo definito a priori, perché ad esempio si vuole avere una latenza molto bassa nella comunicazione. Inoltre, all'interno del cooperative computing, viene definito un concetto di *Virtual Edge Computing* (V-Edge)

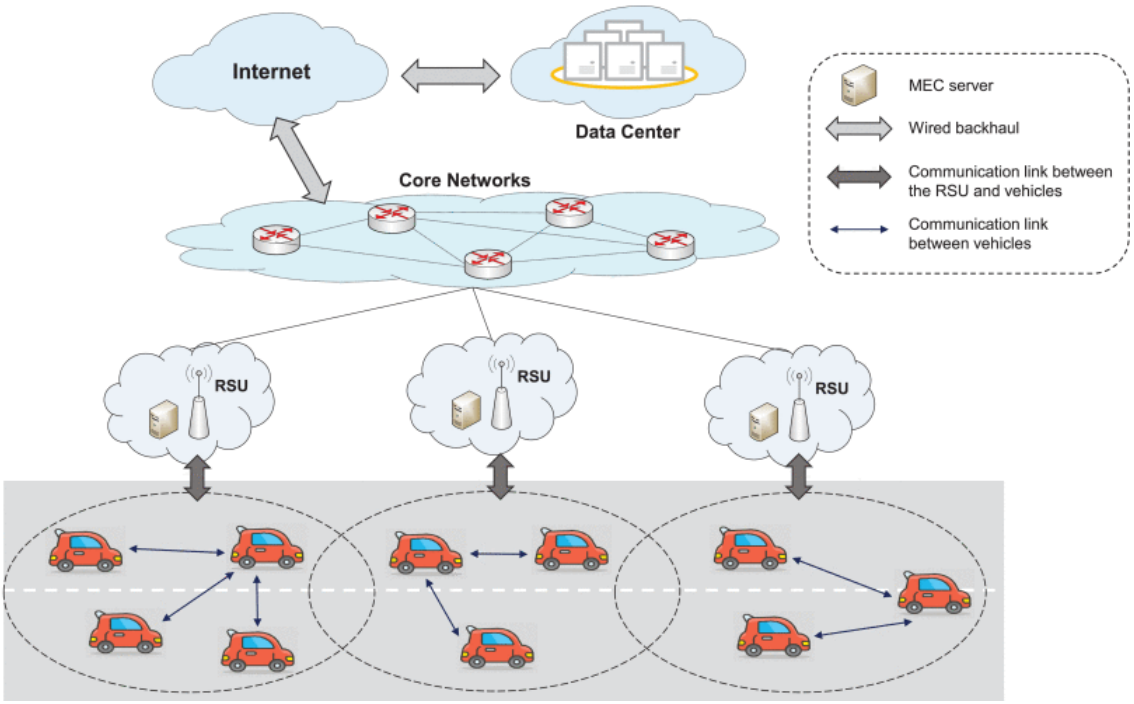

Figura 2: *Architettura del Vehicular Edge Computing*

che sfrutta la flessibilità della softwarizzazione della rete per integrare, in maniera dinamica, l'insieme di risorse disponibili localmente presenti agli edge della rete [3].

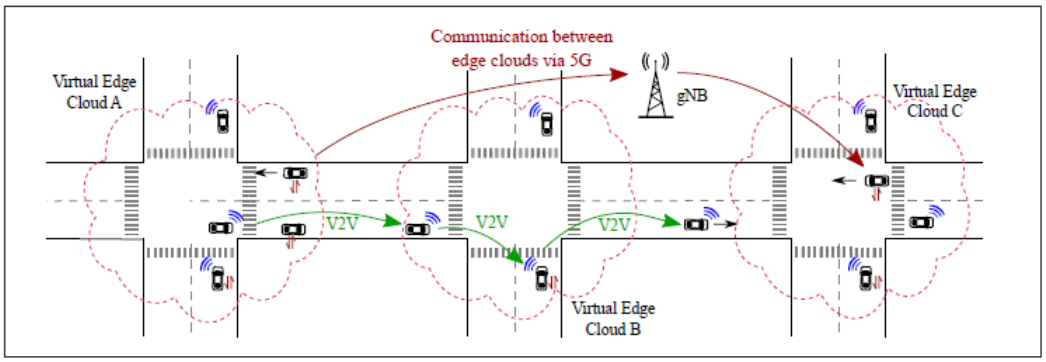

Figura 3: *Architettura del Cooperative Computing*

In Figura 3 si può osservare un esempio di funzionamento di cooperative computing. I veicoli presenti all'interno dello stesso V-Edge e quelli adiacenti possono

comunicare tra di essi sfruttando la comunicazione V2V. Quelli più distanti invece comunicano tramite la rete 5G. La copertura del V-Edge permette di collocare e distribuire funzioni di computazione all'interno della rete veicolare, assumendo che il contesto di quest'ultima sia *distribuito* e *coordinato*. Ciò permette ai veicoli di utilizzare queste funzioni come supporto per poter portare a termine task che essi devono computare.

1.2 Task Offloading nel VEC

Il task offloading nell'ambito di vehicular edge computing è considerato una soluzione promettente ai problemi di latenza e difficoltà computazionale presentati nelle sezioni precedenti, in quanto consente di alleggerire il carico computazionale dei veicoli che devono svolgere *task* computazionalmente pesanti come per esempio la guida autonoma [4].

Il *Vehicular Task Offloading* (VTO) è una particolare tipologia di task offloading. In esso si utilizzano due principali tipologie di comunicazione: *V2V* e *V2I*. Nel V2V la comunicazione è inter-veicolare, ossia i veicoli comunicano tra di essi, scambiandosi dati ed informazioni, sfruttando il protocollo `802.11p` e la tecnologia `WAVE`. Nel V2I la comunicazione avviene scambiando dati con gli edge-servers presenti all'interno delle RSU posizionate a bordo strada.

La principale differenza tra queste due tipologie di comunicazione è che, di norma, le RSU a bordo strada hanno risorse computazionali molto più grandi in termini di disponibilità rispetto a quelle, molto più limitate, dei veicoli.

Nel VTO i task possono essere generati da un singolo nodo che farà *offloading* della computazione, ossia manderà i dati da computare ad altri nodi (veicoli o RSU) presenti all'interno della rete veicolare ed attenderà la risposta computata da essi. Uno

schema di questa comunicazione all'interno della rete veicolare può essere visualizzato in Figura 4.

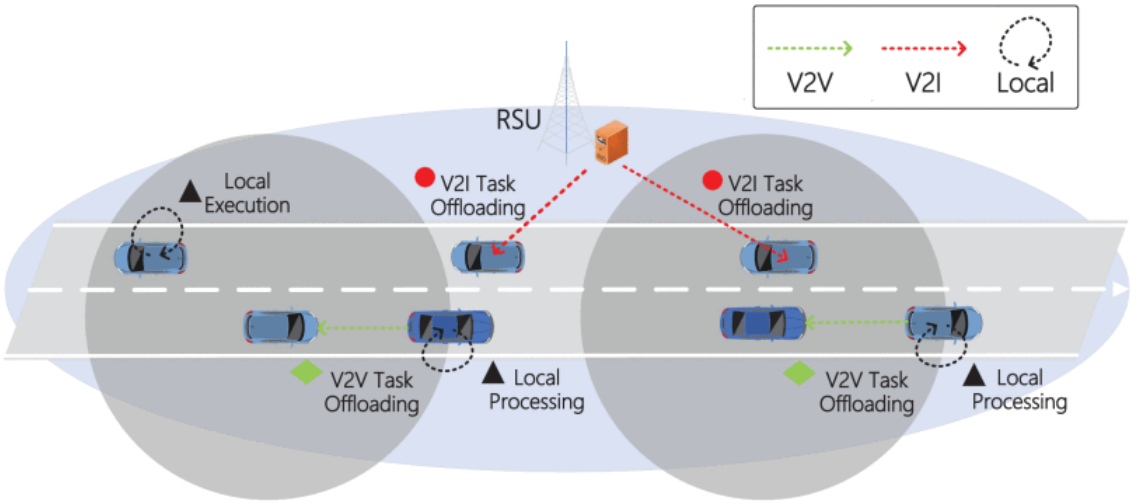

Figura 4: *Funzionamento task offloading in VTO*

I protagonisti principali della comunicazione quando si utilizza la tecnica di task offloading sono:

- I veicoli (o le infrastrutture) che generano il task

- I veicoli (o le infrastrutture) designati per la computazione

L'architettura generica di un sistema di task offloading assume che i nodi che generano il task, distribuiscano come meglio credono il carico di lavoro tra le disponibilità che hanno ricevuto dagli altri veicoli o dalle altre infrastrutture, decidendo inoltre se includere le loro risorse di calcolo, oppure delegare completamente l'intera computazione.

I lati positivi dell'utilizzo della tecnica di task offloading sono il minor tempo di latenza nelle applicazioni e la migliore efficienza energetica. Il minor tempo di latenza è dovuto al fatto che la computazione avviene molto vicina ai veicoli, per cui i tempi di risposta sono quasi istantanei.

La migliore efficienza energetica invece è dovuta al fatto che i nodi, distribuendo il carico nel modo migliore possibile, non avranno sempre risorse di computazione occupate al massimo. Ciò comporta, come specificato in precedenza, che le emissioni prodotte dai veicoli o dagli edge-servers siano ridotte.

I lati da tenere in considerazione per una maggiore efficienza invece sono la sicurezza del dato e la gestione delle risorse. Per quanto riguarda la sicurezza del dato si deve tenere in considerazione che, specialmente in ambito veicolare dove si utilizza un mezzo di comunicazione broadcast, tutti i nodi vicini ricevono i messaggi all'interno della comunicazione. Per ovviare a questo problema, si potrebbe introdurre un sistema di autenticazione tra due nodi per una comunicazione più sicura. Il fatto di non avere questo tipo di meccanismo potrebbe portare ad attacchi di tipo *Man In The Middle* (MIDM).

Per gestione delle risorse si intende il fatto che, ad un nodo in rete delegato per la computazione di un dato, non può essere allocata una quantità di dati superiore a quella che esso possiede. Per ovviare a questo problema, una possibile soluzione è quella di ricevere dal nodo che computa la sua disponibilità in termini di risorse di calcolo durante lo scambio iniziale di messaggi. Ciò permetterà a chi genera il task, di distribuire dati agli altri nodi tenendo conto di ciascuna delle loro disponibilità.

1.3 Standard ed architettura

In questa sezione verranno introdotti i principali standard utilizzati per la realizzazione del sistema di comunicazione inter-veicolare adottato dai veicoli in ambiente di simulazione per fare task offloading. Verrà quindi fatta una breve panoramica su *Wireless Access in Vehicular Environments* (`WAVE`), ossia una famiglia di protocolli per la comunicazione diretta tra veicoli. Infine verranno presentati gli standard di

comunicazione IEEE 802.11p e IEEE 1609.4, entrambi appartenenti alla famiglia di protocolli WAVE.

1.3.1 WAVE

Wireless Access in Vehicular Environments (WAVE) è una famiglia di protocolli che si basano sulla tecnologia *Dedicated Short Range Communication* (DSRC). Essa definisce un'architettura ed i servizi necessari ai dispositivi multi-canale per poter comunicare.

Lo stack di protocolli WAVE completo può essere osservato in Figura 5, anche se per questo elaborato ci concentreremo principalmente sugli standard 802.11p e 1609.4, come precedentemente menzionato.

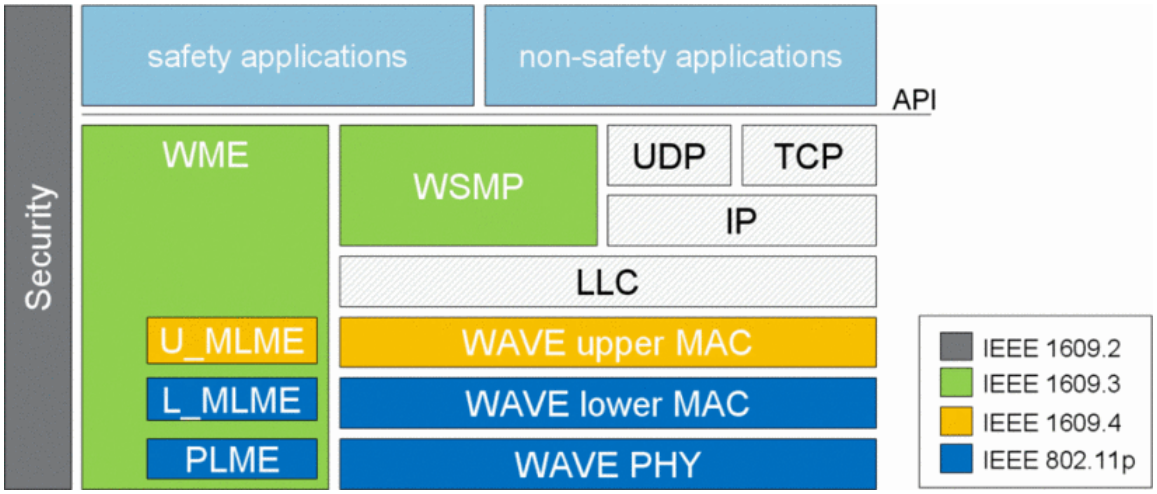

Figura 5: *Stack di protocolli WAVE*

1.3.2 IEEE 802.11p

Nel 1999 la Federal Communication Commission degli Stati Uniti ha allocato 75MHz di spettro DSRC a 5.9GHz utilizzabile esclusivamente per comunicazioni V2V e V2I [5].

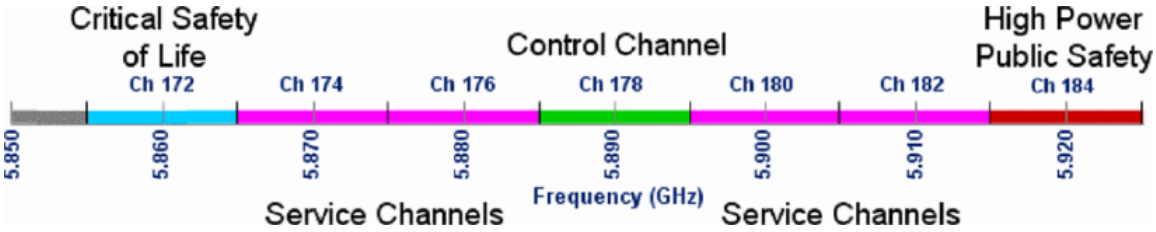

Figura 6: *Canali di servizio e di controllo*

A livello fisico, lo standard `802.11p`, si affida a 7 canali di comunicazione, ciascuno con 10`MHz` di larghezza di banda come possiamo osservare in Figura 6. Il canale 178 è chiamato *canale di controllo* (CCH) ed il suo utilizzo è ristretto alle sole comunicazioni di sicurezza. I due canali all'inizio ed alla fine dello spettro (172 - 184) sono riservati a casi d'uso particolari, mentre tutti gli altri vengono chiamati *canali di servizio* (SCH) e possono essere utilizzati sia per motivi di sicurezza, sia per altre motivazioni.

Di norma, per comunicare, le stazioni in 802.11 devono appartenere allo stesso *Basic Service Set* BSS, cioè avere tutte lo stesso *Service Set Identifier* (SSID). La creazione di un BSS però è una operazione che richiede molto tempo, non compatibile quindi con i tempi richiesti da un ambiente dinamico ed in costante cambiamento come le reti di traffico veicolare.

La principale caratteristica di `802.11p` è quella di estendere il BSS nel seguente modo:

- rimuove i servizi di autenticazione e confidenzialità richiesti da 802.11
- introduce un *Basic Service Set Identification* (BSSID), una wildcard con tutti i bit a 1 (broadcast), che permette di comunicare con tutti i veicoli
- delega l'implementazione di funzionalità di sicurezza ai livelli di rete superiori

1.3.3 IEEE 1609.4

Lo standard IEEE `1609.4` è nato per aiutare il livello data link di `802.11p` per supportare la connettività multi-canale della modalità WAVE, consentendo una migliore gestione ed un miglior coordinamento dei diversi canali [6].

Esso descrive le operazioni di *channel switching* e *channel routing* che vengono fatte sui canali e controlla la divisione tra i CCH ed i SCH di IEEE `802.11p`. Oltre a questo descrive un concetto di un protocollo di suddivisione del tempo, dove quest'ultimo è diviso in intervalli CCH ed SCH, come osservabile in Figura 7.

Se due o più dispositivi volessero scambiare dati sullo stesso canale, è necessaria una sincronizzazione temporale, per cui, quello che viene fatto in `1609.4` è sincronizzare i canali CCH ed SCH sul *Coordinated Universal Time* (UTC).

Oltre a definire intervalli di tempo CCH ed SCH, lo standard `1609.4` definisce anche dei *guard interval* (Figura 7) all'inizio di ogni canale CCH ed SCH. Il guard interval è importante perché permette ai diversi nodi di poter tener conto delle diverse differenze di orario che potrebbero esserci tra di essi. Inoltre, va specificato che, durante il tempo di guard interval, i nodi in rete non hanno il permesso di mandare o ricevere qualsiasi tipo di dato, per cui tutto il traffico di rete è momentaneamente sospeso.

Infine, il valore di un guard interval, è così definito:

$$Sync\ tolerance + Max\ channel\ switch\ time \tag{1}$$

dove *sync tolerance* è la stima della precisione di un clock interno ad un nodo comparata con UTC, mentre il *max channel switch time* è la stima del tempo che un nodo ci mette per sintonizzare la sua radio su un diverso canale. La loro somma (1) è quindi la stima del tempo di sincronizzazione dei diversi nodi all'interno della rete.

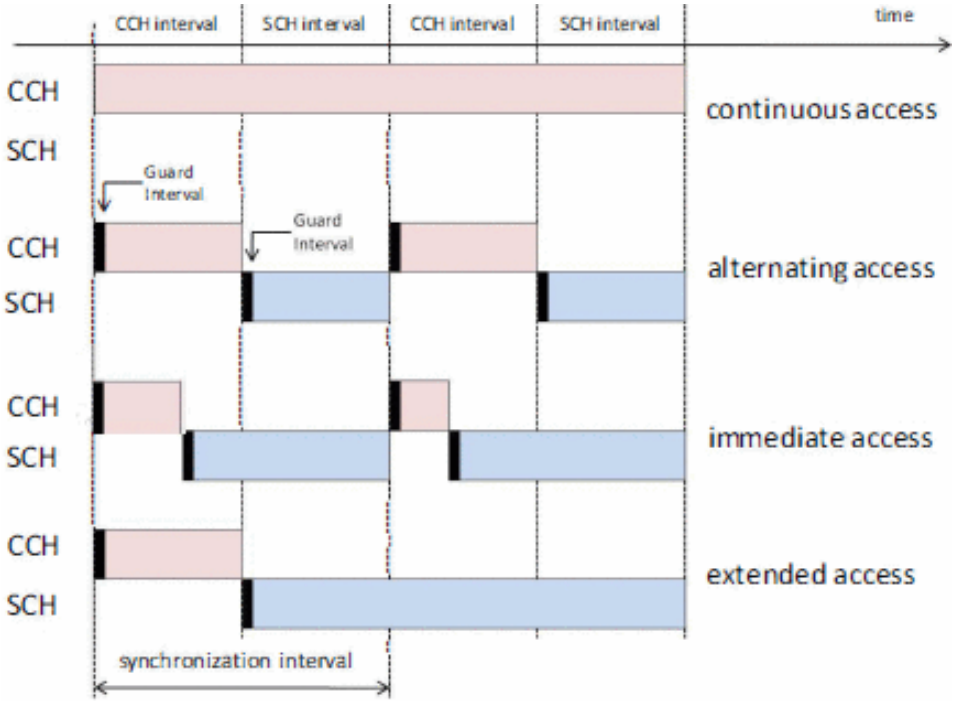

Figura 7: *Tipologie di accesso a canale in 1609.4*

Capitolo 2

Task Offloading

In questo capitolo verrà descritto nel dettaglio il protocollo di comunicazione tra veicoli che è basato sullo scambio di messaggi che avviene tra di essi. Questi messaggi possono contenere sia dati da computare, sia informazioni riguardanti le caratteristiche dei veicoli, utili per l'avanzamento della computazione.

2.1 Protocollo di comunicazione

Gli attori principali della comunicazione sono:

- *Task generator* (o anche detto "generatore"): è colui che da il via all'intera comunicazione tra veicoli. Esso si occupa di creare il *task*, ossia il lavoro di una dimensione fissata a priori che, tramite task offloading, dovrà essere completato con l'ausilio di altri veicoli.

- *Workers* (o anche "lavoratori"): sono comuni veicoli che hanno il ruolo fondamentale di aiutare, per quanto possibile, il generatore a computare piccoli pezzi di dati secondo una disponibilità che viene data in termini di capacità di calcolo.

Durante la creazione del protocollo è stata effettuata una assunzione fondamentale per il corretto funzionamento a livello logico dell'applicazione e della comunicazione

tra veicoli: la funzione da calcolare può essere suddivisa in più parti da parte del task generator e quella di riassemblaggio è **commutativa** ed **associativa**. Ciò significa che il generatore può assumere che le piccole partizioni di dato che invia da computare e che riceve computate da altri veicoli non debbano essere considerate necessariamente ordinate per poter essere interpretate correttamente.

Prima di passare al dettaglio delle diverse fasi di esecuzione del protocollo e dei messaggi che i veicoli si scambiano durante l'esecuzione della applicazione, è opportuno specificare che la comunicazione avviene utilizzando il protocollo `802.11p` (protocollo standard per la trasmissione dati in sistemi di comunicazione per veicoli intelligenti specificato in precedenza) e la tecnologia WAVE.

Le diverse fasi del protocollo sono mostrate in Figura 8 e sono:

- *Richiesta di aiuto alla computazione*: una prima fase in cui il task generator chiede aiuto per la computazione ai veicoli (workers) che sono vicini ed essi, controllando le loro risorse computazionali, decidono se aiutarlo o meno

- *Load balancing*: seconda fase in cui il task generator, con le risorse di calcolo rese disponibili dai workers, decide come distribuire il carico di computazione

- *Scambio di dati*: ultima fase in cui avviene uno scambio di dati da computare e risposte computate tra generatore e workers

2.2 Fasi di esecuzione

In questa sezione, verranno presentate nel dettaglio le diverse fasi di esecuzione per la computazione di una singola partizione di dato all'interno del protocollo di comunicazione, oltre che i messaggi che in ogni fase si scambiano task generator e worker.

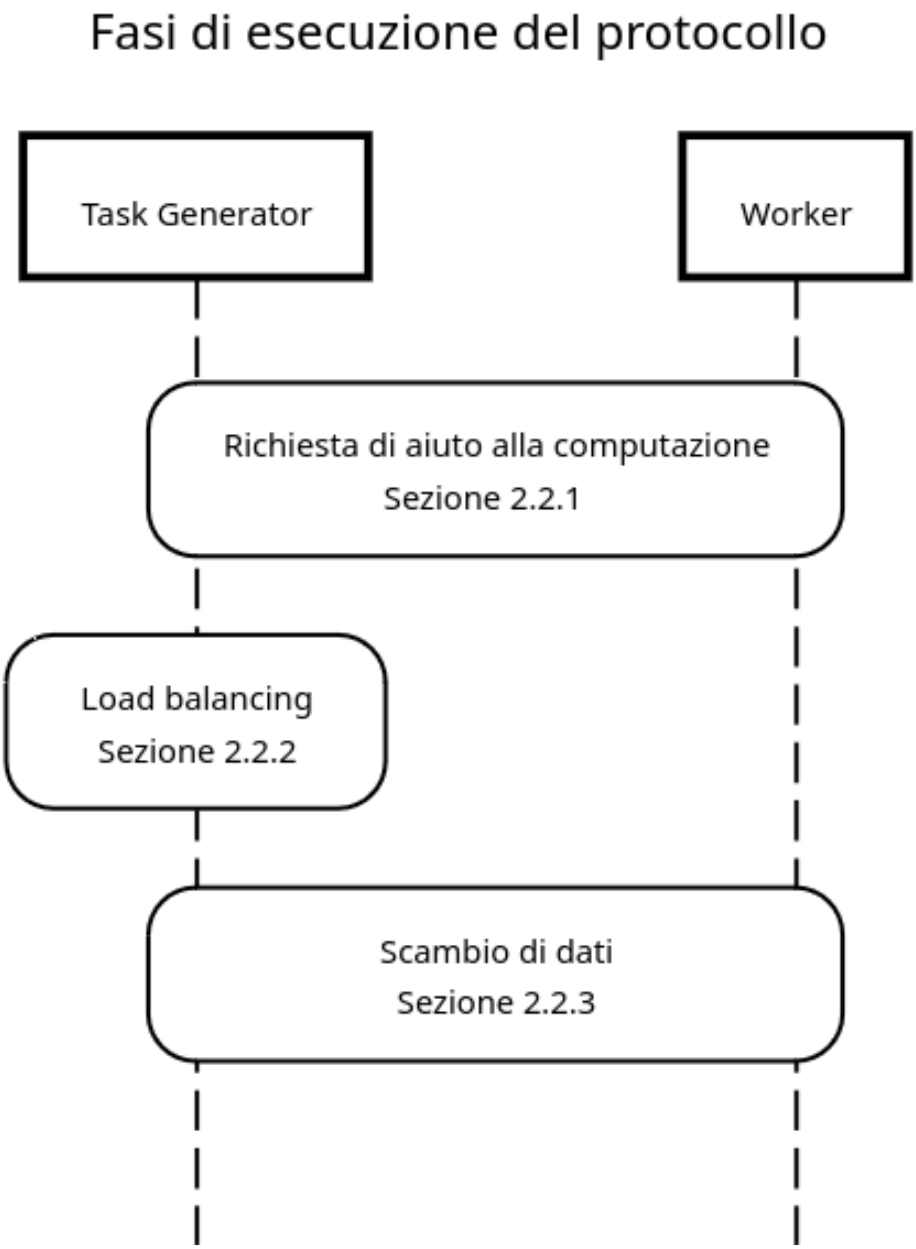

Figura 8: *Fasi del protocollo di task offloading*

2.2.1 Richiesta di aiuto alla computazione

In questa prima fase del protocollo il task generator deve capire se qualche veicolo (worker) è disponibile ad aiutarlo nella computazione. Per raggiungere questo obiettivo vengono scambiate due tipologie di messaggio:

- *Help message*: rappresenta la richiesta di aiuto inviata in broadcast ai workers da parte del task generator
- *Availability message*: è il messaggio che un generico worker invia al task generator per indicargli la sua disponibilità nella computazione

Help message

L'help message è il messaggio iniziale che viene inviato dal task generator in broadcast ai possibili diversi workers che siano abbastanza vicini per poterlo ricevere. Esso contiene diverse informazioni riguardanti le caratteristiche del task che deve essere svolto:

- ID del generatore
- ID del task
- Difficoltà del task (Computing Density - CPI)
- Dimensione del task
- Minima disponibilità richiesta per la computazione

Gli ID del generatore e del task sono identificatori univoci del veicolo che avvia la computazione dei dati e del task che deve essere svolto. L'ID del task è stato pensato in un'ottica futura per permettere la computazione di più di un singolo task per veicolo.

La difficoltà del task (o computing density) è un numero intero che rappresenta quanto è complicato in termini di computazione il lavoro che deve essere svolto. Più è alto questo numero e più sarà alto il tempo totale richiesto per portare a termine il lavoro.

Infine, la minima disponibilità richiesta per la computazione è un dato che serve al worker per valutare internamente se può condividere risorse di calcolo con il generatore, oppure no. La valutazione viene fatta controllando se la sua disponibilità di risorse di calcolo interne sia almeno pari a quella richiesta dal generatore. In base a questa valutazione il veicolo poi deciderà se mandare la sua disponibilità tramite l'apposito messaggio *availability message*, oppure no.

Se dovesse capitare che nessuno dei workers sia disponibile per la computazione, allora il task generator, dopo un tempo t stabilito a priori, ripete il processo mandando un nuovo help message.

Un diagramma di sequenza che mostra il comportamento di help message è mostrato in figura 9.

Availability message

L'availability message (il cui diagramma di sequenza è rappresentato in Figura 10) è il messaggio che un generico worker invia al generatore per indicare la sua disponibilità alla computazione del task e contiene i seguenti dati:

- ID del veicolo
- La sua disponibilità
- La sua frequenza di CPU
- Angolo verso cui è rivolto
- Velocità attuale

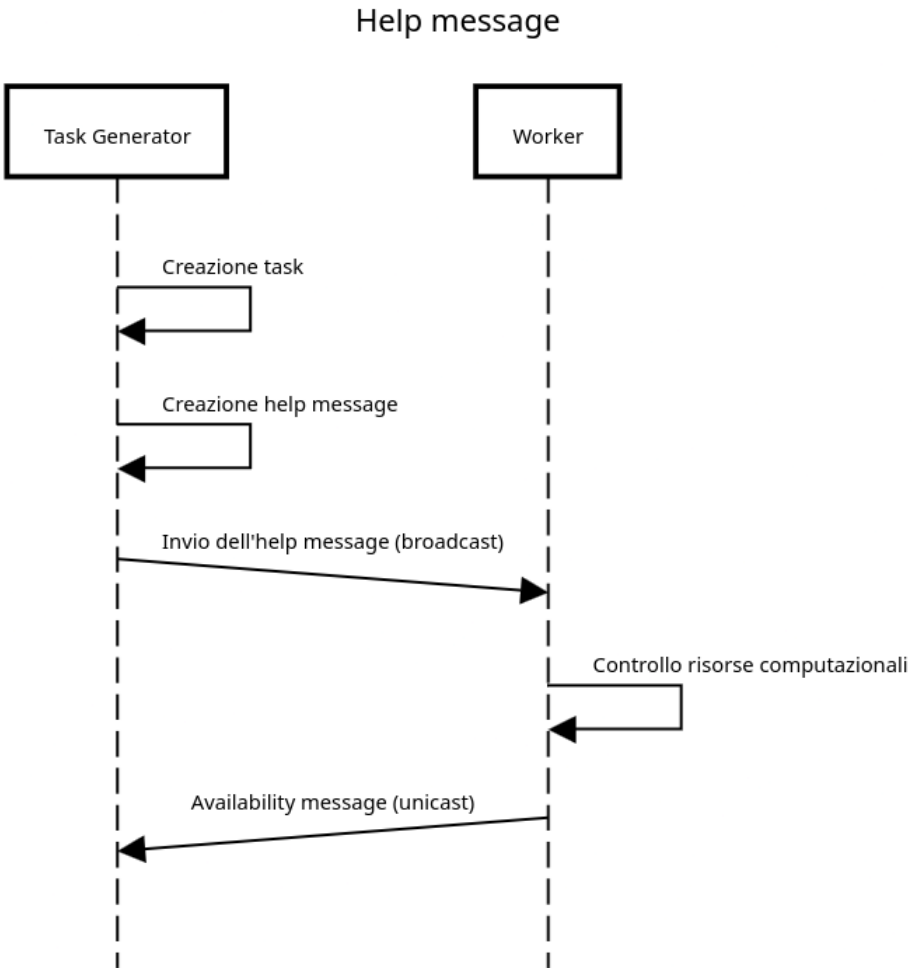

Figura 9: *Diagramma di sequenza per help message*

- Coordinate x ed y per indicare la sua posizione

L'ID del veicolo è un identificatore univoco che serve al generatore per sapere con quale veicolo sta comunicando e per verificare nelle fasi successive della comunicazione quante partizioni di dato ha ricevuto da quel veicolo, oppure che gli ha inviato.

La disponibilità del worker e la sua frequenza di CPU sono dati molto importanti che servono al generatore, sia per il calcolo del numero di partizioni di dato da inviargli, sia per impostare correttamente il timer di attesa per la ricezione della risposta

da parte di quel veicolo.

Gli altri dati (direzione del veicolo, velocità e coordinate) servono al generatore per poter stimare il tempo di permanenza del worker all'interno del range di computazione. Esso potrà in seguito decidere di non inviare dati ad un worker, se la stima non supera una certa soglia s prefissata. Questa soluzione evita, ad esempio, che il generatore invii dati ad un veicolo che sta viaggiando in una direzione di marcia opposta a quella del generatore.

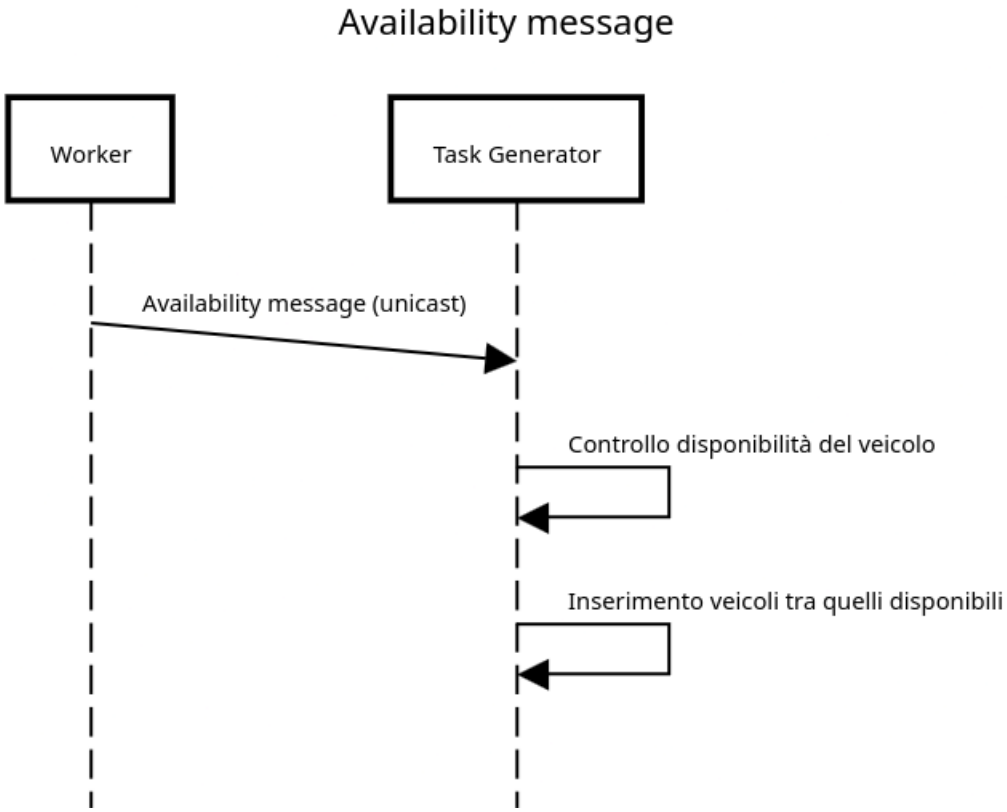

Figura 10: *Diagramma di sequenza per availability message*

2.2.2 Load balancing

In questa seconda fase, dopo che il task generator aspetta per un totale di tempo t definito a priori, smette di accogliere nuove disponibilità e si pone nello stato di load balancing. Da qui in poi tutte le proposte di aiuto dei workers non verranno più considerate.

In questa fase, che riguarda solamente il task generator, non vengono scambiati particolari messaggi. Il load balancing è un'importante funzione che permette al generatore di distribuire il carico nella maniera più ottimale possibile tenendo conto delle risorse che gli sono state messe a disposizione dai lavoratori.

Attualmente sono stati implementati due algoritmi di ordinamento che possono essere adottati per decidere come inviare le partizioni di dato ai veicoli:

- FIFO

- ComputationTime

L'algoritmo *FIFO* invia i dati ai veicoli ordinati per tempo di arrivo della proposta di disponibilità, ossia il primo che manda le sue informazioni con l'availability message al generatore, sarà il primo a ricevere i dati.

L'algoritmo *ComputationTime*, invece, effettua un piccolo calcolo prendendo in considerazione le risorse disponibili di ciascun worker e stimandone il tempo di conclusione della computazione della partizione di dato. La stima viene effettuata utilizzando questa formula:

$$\frac{CPI \cdot Disponbilita'}{frequenza\ CPU} \tag{2}$$

Una volta che vengono ottenuti tutti i tempi di calcolo della partizione di dati utilizzando la formula precedente (2), i lavoratori vengono ordinati per tempo di completamento minore, ciò significa che il primo veicolo a ricevere i dati sarà anche il primo a terminare la computazione. Questo algoritmo è più interessante rispetto a

FIFO dal punto di vista del funzionamento del protocollo, perché nel meccanismo di risposta è stato implementato un dato, la cui importanza verrà spiegata in seguito, che permette al generatore di continuare a tenere in considerazione il veicolo anche una volta finita la computazione e di conseguenza di velocizzare anche il tempo totale per il completamento del task.

Quando il task generator ottiene la lista di veicoli, ordinata secondo l'algoritmo scelto tra uno dei due appena elencati, divide la disponibilità del worker per la massima dimensione del pacchetto UDP consentita, che in questo caso è di `64kB`:

$$numero\ partizioni = \frac{Disponibilita'}{64kB} \tag{3}$$

2.2.3 Scambio di dati

In questa terza ed ultima fase, il task generator, inizia ad inviare partizioni di dato da computare ai diversi veicoli che gli hanno dato disponibilità, essi computano la risposta e poi la inviano la inviano al generatore, attendendo la conferma (ACK) che essa sia arrivata correttamente. I messaggi che vengono scambiati sono i seguenti:

- *Data message*: è il messaggio che contiene la partizione di dati che un worker deve computare
- *Response message*: è il messaggio di risposta contenente la partizione dati computata dal worker
- *Ack message*: è il messaggio che conferma la corretta ricezione di una partizione dati computata al worker

Data message

Questo tipo di messaggio porta con se i dati da computare da parte del worker e contiene i seguenti campi:

- Dimensione della partizione
- Dimensione totale del task
- Tempo di computazione stimato
- ID del generatore
- ID della partizione dati
- ID del task
- ID di load balancing
- Difficoltà del task
- Numero totale di partizioni di dato

L'idea fondamentale su cui si basa questo protocollo è quella di portare a termine la computazione tenendo conto di quanti più possibili casi di errore nella comunicazione (es. perdita di un messaggio dati o di una risposta).

A questo scopo i campi ID della partizione di dati, ID del task, ID di load balancing e numero totale di partizioni di dato servono al worker per poter creare una chiave univoca che identifica una singola partizione di dati all'interno della comunicazione. Ciò gli permette di capire esattamente quali dati computati sono arrivati al generatore e quali deve re-inviare perché sono andati persi. La dimensione della partizione invece, serve per poter simulare il tempo di computazione con le risorse che ha a disposizione.

Quando al worker arriva un *data message*, lo processa e computa la partizione dati che gli è stata assegnata. Non appena finisce, la risposta viene inviata in unicast al generatore utilizzando il suo ID univoco presente all'interno del messaggio, inoltre viene fatto partire un timer di attesa dell'ack. Se il timer dovesse scadere senza la ricezione di un *ack message* da parte del generatore, allora il worker lo interpreterebbe

come una mancata ricezione del messaggio di risposta, per cui essa dovrà essere inviata di nuovo.

Il diagramma di sequenza del data message è rappresentato in figura 11.

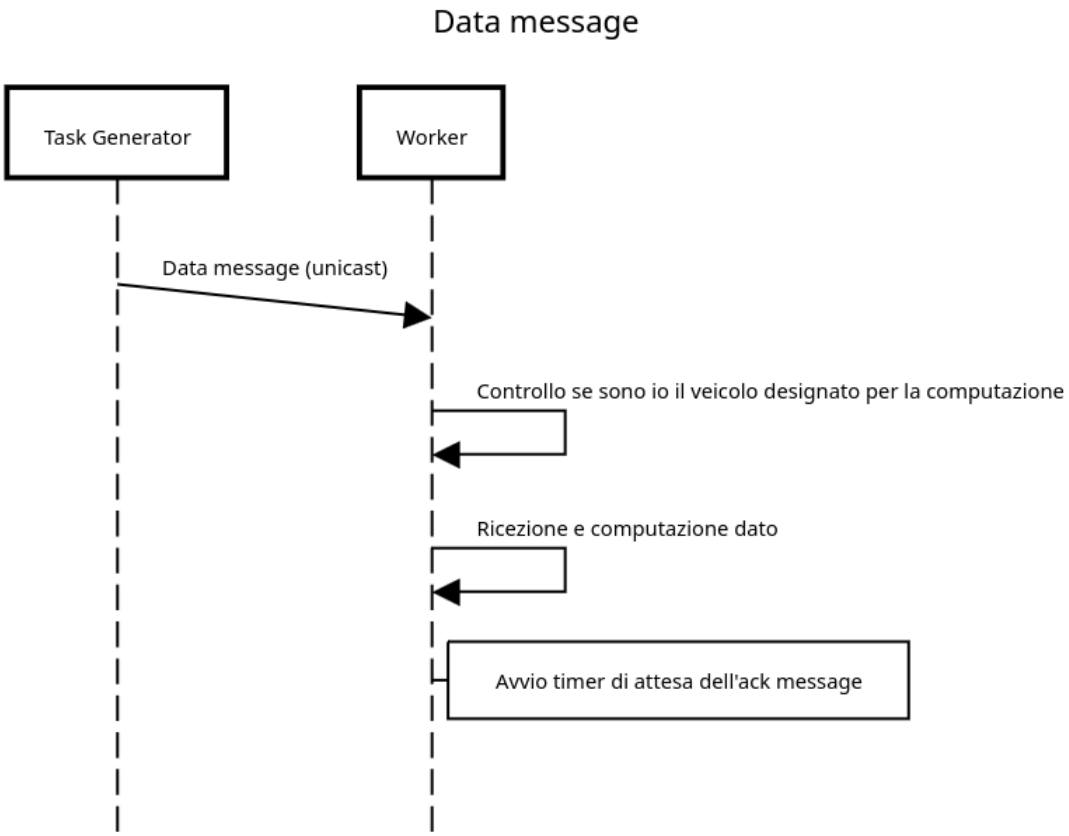

Figura 11: *Diagramma di sequenza per data message*

Response message

Il messaggio di risposta contiene i dati computati dal worker per il task generator e contiene i seguenti campi:

- ID del task
- ID della partizione dati

- ID del worker

- ID del generatore

- Dimensione dati computati

- Tempo di computazione impiegato

- Nuova offerta di aiuto (still available)

Così come per il *data message* gli ID del task, della partizione dati e del worker servono al generatore per identificare univocamente la partizione dati che è stata computata all'interno del protocollo di comunicazione.

Il campo **still available** contiene un dato molto importante, menzionato precedentemente, che permette al worker di indicare al task generator se sia ancora disponibile per computare una nuova partizione dati oppure no. Se questo campo viene impostato a `true`, il generatore potrà assumere che il veicolo sia ancora disponibile per una nuova computazione con le stesse risorse messe a disposizione precedentemente, altrimenti non verrà più considerato. L'importanza di questo campo è data dal fatto che se uno o più veicoli dovessero dare una nuova disponibilità, il task generator anziché tornare in uno stato in cui debba inviare una nuova richiesta di aiuto per la computazione, passerebbe direttamente alla fase di load balancing con le nuove disponibilità date dai workers. L'impatto che questo meccanismo ha sull'intero protocollo di comunicazione è quello di garantire un tempo totale minore di conclusione del task ed un numero totale minore di messaggi generati.

Una volta che il task generator riceve una risposta, processa i dati ed invia subito il messaggio di ack al lavoratore, per far si che quest'ultimo sappia che ha ricevuto la risposta. Se al generatore dovesse arrivare una risposta che ha già ricevuto, quindi viene perso il messaggio di ack, semplicemente quest'ultima viene scartata e viene re-inviato il messaggio di conferma.

Il diagramma di sequenza del response message è rappresentato in figura 12.

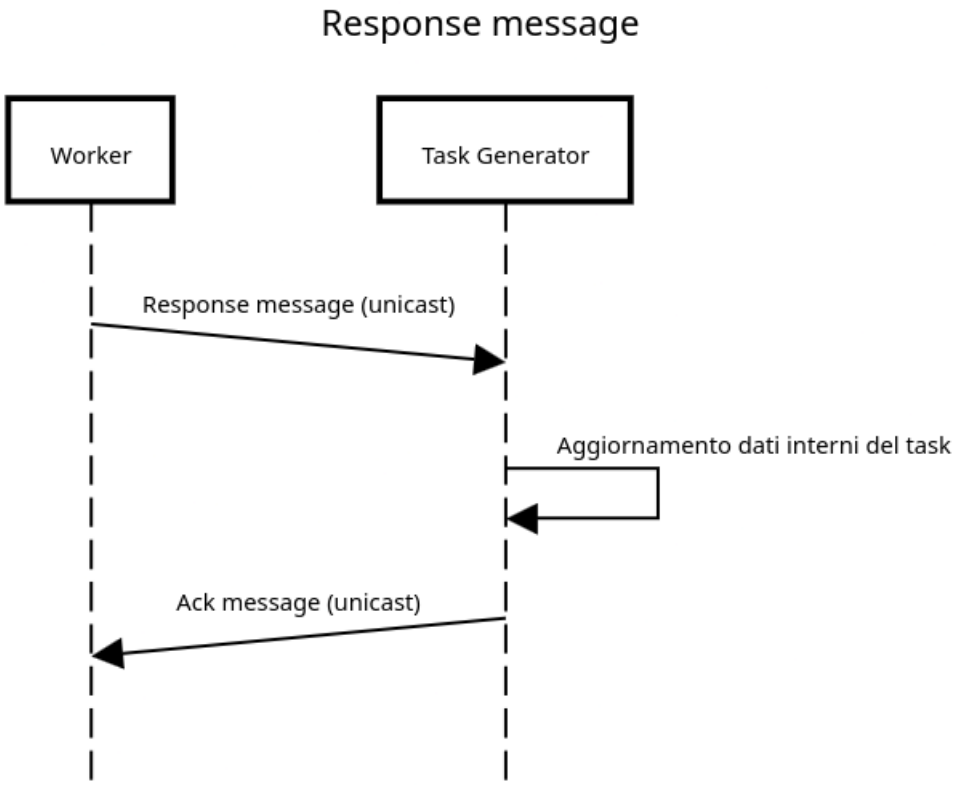

Figura 12: *Diagramma di sequenza per response message*

Ack message

Quando il worker riceve un messaggio di ack (diagramma di sequenza in figura 13) vuol dire che la computazione di una partizione di dati è avvenuta correttamente ed il task generator ha ricevuto la risposta. Esso contiene:

- ID del generatore
- ID del task
- ID della partizione dati

Questi campi permettono, esattamente come *data message* ed *response message*, di identificare univocamente la partizione di dati computata all'interno della comunicazione.

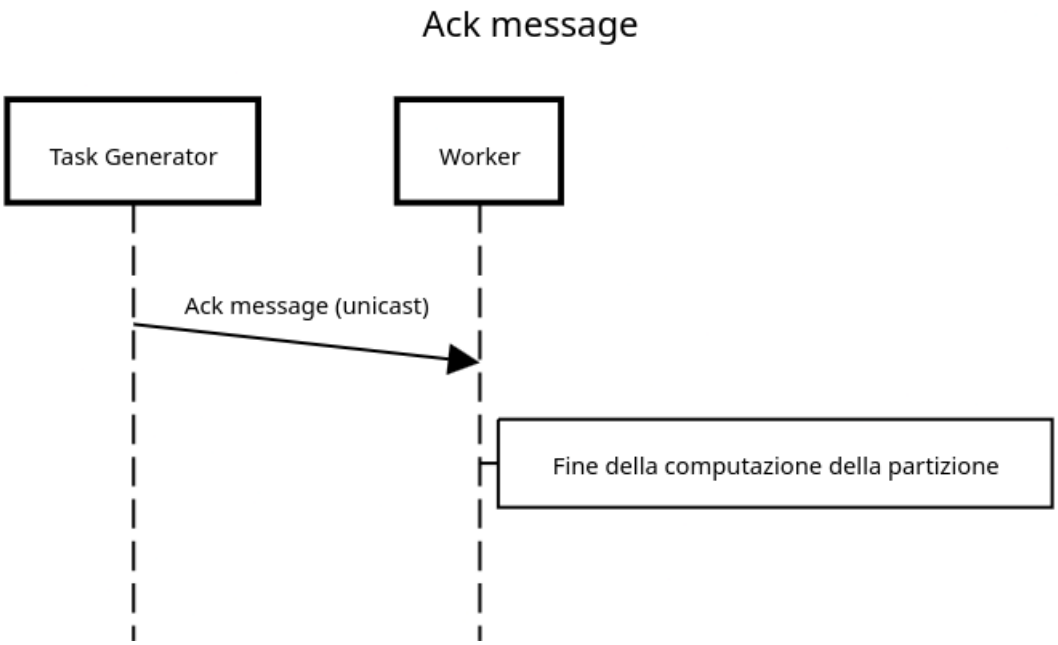

Figura 13: *Diagramma di sequenza per ack message*

2.3 Timers

All'interno del protocollo esistono dei timers che i veicoli impostano per loro stessi. Essi garantiscono che i messaggi di dati e risposta arrivino correttamente sia al task generator che ai workers. Un timer scaduto (tranne nel caso specifico del timer di attesa del generatore per collezionare proposte di aiuto da parte dei workers) corrisponde generalmente ad un messaggio perso che deve essere re-inviato.

2.4 Osservazioni

Il protocollo ideato garantisce che i task iniziati dal task generator vengano sempre conclusi e che tutte le partizioni di dato che vengono inviate tornino computate. Il fatto che, per ogni partizione di dati, venga richiesto un *ack message*, provoca uno scambio di messaggi abbastanza consistente. Un possibile sviluppo futuro del protocollo di comunicazione potrebbe essere quello di creare una coda di ricezione di messaggi lato worker, ciò permetterebbe l'accumulazione di partizioni di dato per poi re-inviarle computate tutte assieme ed attendere un singolo *ack* che confermi, o meno, la corretta ricezione di tutte quante. Di contro, però, questo meccanismo richiederebbe più attenzione e sensibilità all'errore, quindi maggior tempo di computazione totale, perché la ritrasmissione andrebbe adattata solamente alle singole partizioni di dato andate perse e non all'intero dato da computare.

Capitolo 3

Ambiente di simulazione

Per poter valutare le prestazioni del protocollo presentato al capitolo precedente è stato utilizzato il simulatore *OMNeT++* integrato con *SUMO* ed il sotto-progetto *Veins_INET* presente all'interno del framework *Veins*. Questi strumenti permettono la simulazione di reti comunicazione tra veicoli e traffico stradale in ambiente urbano. Nel caso specifico di questo lavoro di tesi, sono state considerate due tipologie di applicazione: *Task Generator* e *Worker*, ognuna associata alla specifica tipologia di veicolo. Inoltre è stato introdotto un modulo *Task*, generato dinamicamente a run-time all'interno del Task Generator, che contiene tutte le informazioni riguardanti il task che deve essere svolto.

3.1 Strumenti utilizzati

Per la realizzazione di questo elaborato, è stato scelto *OMNeT++* (Objective Modular Network Testbed in C++). Esso offre moltissime funzionalità per la simulazione delle reti.

Oltre ad esso è stato anche integrato *SUMO* (Simulation of Urban MObility) all'interno dell'ambiente di simulazione, che permette la modellazione di traffico veicolare in ambienti urbani. Anch'esso offre numerose funzionalità, sia per la gestione

dei veicoli, che della rete di traffico stradale, permettendo all'utilizzatore di specificare come i veicoli debbano comportarsi all'interno di un ambiente urbano completo e simulato.

Veins_INET, invece, estende ancora di più le funzionalità di OMNeT++ e SUMO, interfacciandosi tra di essi e permettendo di simulare scenari realistici per quanto riguarda l'ambito di comunicazione veicolare.

L'utilizzo di questi tre strumenti assieme, ha permesso di sviluppare un ambiente di simulazione che fosse più completo e coerente, con lo scopo di poter analizzare il funzionamento del protocollo di Task Offloading in uno scenario di Vehicular Edge Computing.

3.1.1 OMNeT++

OMNeT++ è un simulatore modulare di reti ad eventi, open-source, estensibile e basato su componenti. I suoi componenti vengono creati utilizzando il linguaggio di programmazione `C++` ed in seguito assemblati in componenti più grandi utilizzando un linguaggio specifico ad alto livello: `NED`.

I principali strumenti messi a disposizione da OMNeT++ sono:

- la libreria kernel di simulazione (`C++`) offre un ausilio allo sviluppatore per poter sviluppare il comportamento dell'applicazione installata sui diversi dispositivi (in questo caso sui veicoli), senza doversi preoccupare di definire il comportamento che i veicoli devono avere in simulazione.

- il linguaggio `NED` permette di descrivere la topologia di rete, specificando i moduli che devono essere utilizzati ed alcuni parametri che ne definiscono il comportamento.

- una GUI di simulazione interattiva `Qtenv` permette di poter osservare a run-time come i veicoli interagiscono tra di loro, i messaggi che si scambiano e gli eventuali errori generati.

I moduli all'interno di OMNeT++ sono strutturati come mostrato in Figura 14. I rettangoli rappresentano le diverse tipologie di moduli (*semplici* e *composti*), mentre le frecce rappresentano come sono connessi.

I moduli comunicano tra di loro utilizzando dei messaggi che possono contenere dati di tipo arbitrario, questo consente una gestione più semplice degli eventi di simulazione. Data la struttura gerarchica dei vari modelli di comunicazione, i messaggi vengono trasmessi tramite una catena di connessioni, per poter partire ed arrivare ad un modulo finale di tipo semplice. Una connessione è semplicemente un collegamento tra una porta di input di un modulo ed una di output di un altro [7].

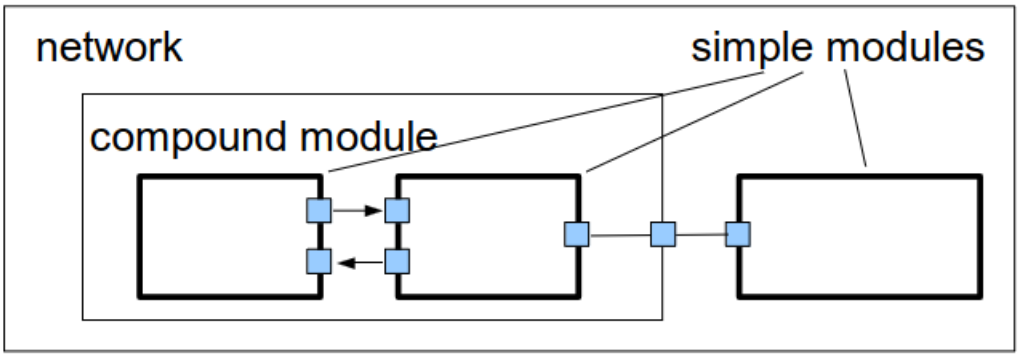

Figura 14: *Struttura dei moduli in OMNeT++*

I moduli infine, possono contenere dei parametri, che vengono tipicamente utilizzati per specificare determinati comportamenti che un modulo può assumere oppure per passare dati di configurazione ai moduli semplici.

3.1.2 SUMO

SUMO (Simulation of Urban MObility) è un software open-source, la cui interfaccia grafica può essere visualizzata in Figura 15, per la simulazione di traffico in ambienti urbani. Permette di modellare qualsiasi sistema di traffico e configurarlo nei minimi dettagli, arrivando persino a specificare il comportamento che possono adottare veicoli, trasporti pubblici e pedoni.

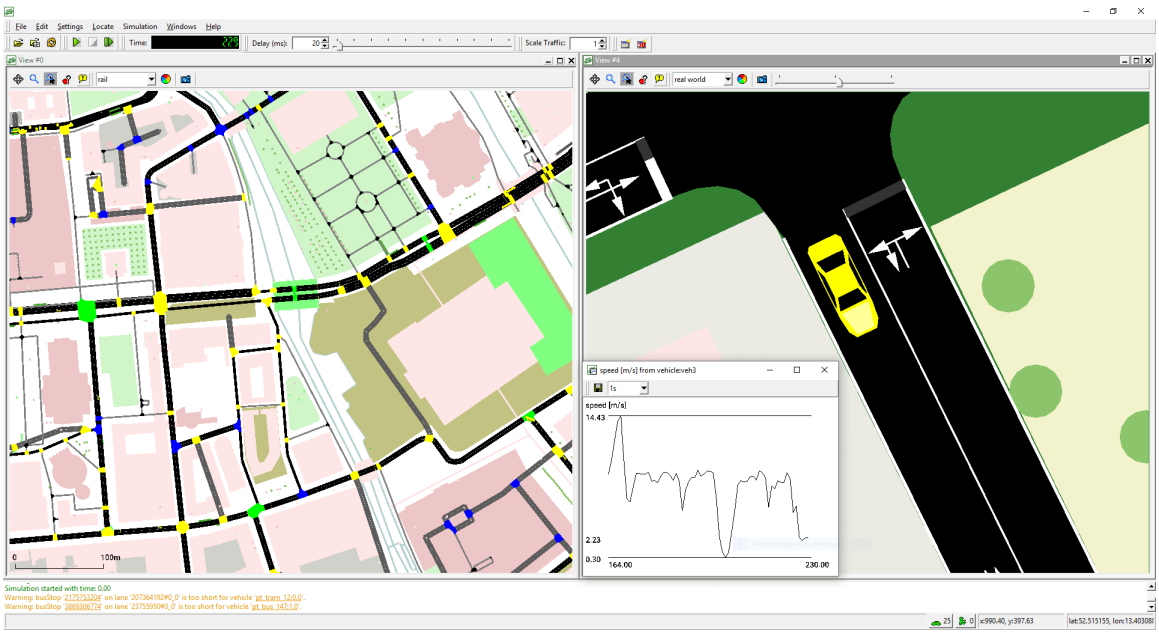

Figura 15: *Interfaccia del simulatore SUMO*

Alcune tra le aree di competenza di questo simulatore [8], per citarne le più importanti, sono: la simulazione dell'effetto del traffico sui veicoli a guida autonoma e sui plotoni di veicoli, la simulazione e la validazione di funzioni di guida autonoma, sicurezza del traffico ed analisi dei rischi ed infine il calcolo delle emissioni di veicoli.

Grazie all'utilizzo di questo software è stato possibile configurare delle simulazioni che hanno tenuto conto dell'interazione tra il traffico veicolare e la rete di comunicazione tra i veicoli, consentendo di creare uno scenario che ha permesso la valutazione

dell'efficacia e delle prestazioni del protocollo introdotto nel precedente capitolo.

Un'altra importante funzionalità di SUMO è data dalla possibilità di scrivere e configurare manualmente una rete stradale utilizzando il software integrato *NetEdit*, per poi esportare la configurazione in formato `XML`. Ciò è risultato molto utile per la cooperazione tra i due software di simulazione utilizzati, SUMO ed OMNeT++, essendo anche quest'ultimo compatibile con il formato `XML`.

Per tutti gli scenari di simulazione è stata utilizzata una mappa preesistente estratta dal database open-source di OpenStreetMap, mostrata in Figura 16. Questa mappa viene visualizzata con la GUI di SUMO e fornisce una visualizzazione dei soli elementi stradali utili ai fini delle simulazioni considerate per questo elaborato.

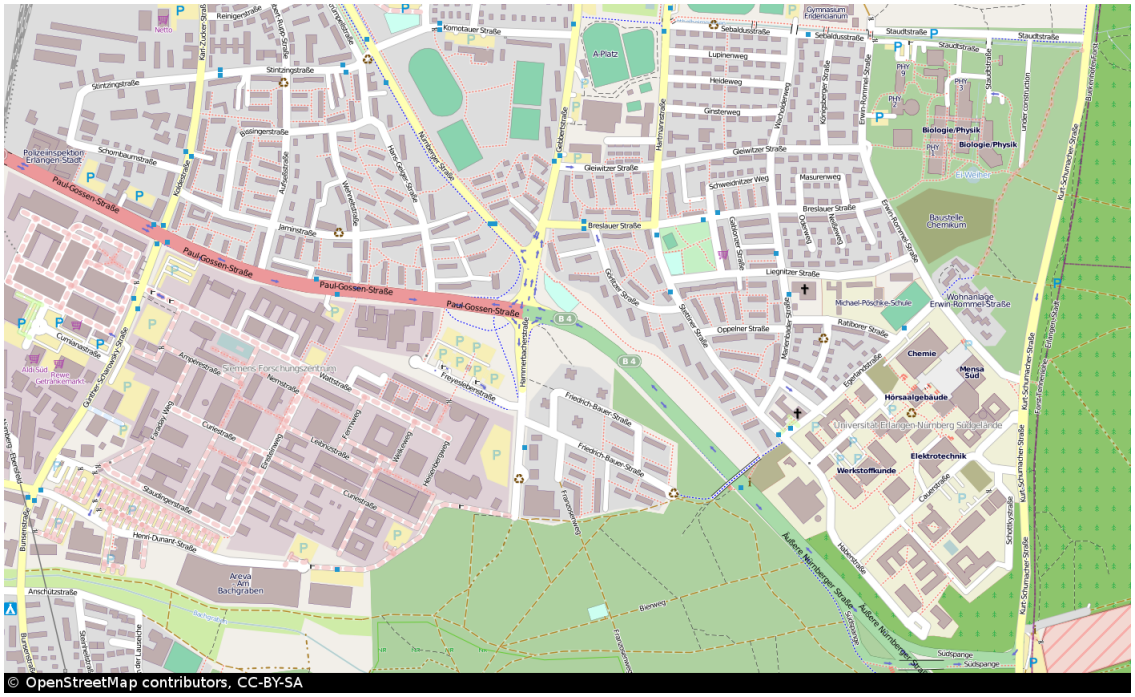

Figura 16: *Mappa della città di Erlangen simulata usando SUMO*

NetEdit

NetEdit è uno strumento di editing visuale per la creazione o modifica di reti di traffico veicolare. Esso permette di poter specificare il comportamento e le configurazioni che i veicoli possono assumere, partendo dalle caratteristiche più generali quali, ad esempio, strada da percorrere, modello e tipologia del veicolo, sino a raggiungere quelle più particolari, come ad esempio velocità massima oppure potenza del motore.

La rete di traffico veicolare, creata utilizzando NetEdit, genera una versione della mappa di Erlangen, visualizzabile in Figura 17.

Figura 17: *Mappa utilizzata negli scenari di simulazione*

3.1.3 INET

Il framework INET è una libreria open-source per l'ambiente di simulazione di OMNeT++. Fornisce protocolli, agenti e modelli preimpostati per facilitare il lavoro di chi opera con le reti di comunicazione. Inoltre, si rivela particolarmente utile per il design e la validazione di nuovi protocolli.

INET contiene modelli per lo stack di protocolli *internet*: `TCP`, `UDP`, `IPv4`, `IPv6`, `OSPF` e `BGP` per citarne i principali, oltre che protocolli a livello *data link*: `Ethernet` ed `802.11` ed un supporto base alla mobilità, che in questo progetto di tesi però sarà gestito da Veins.

3.1.4 Veins

Veins è un framework di simulazione open-source che rende possibile simulare l'interazione tra veicoli all'interno della rete integrando OMNeT++ e SUMO [9]. L'utente per cui dovrà solo occuparsi di scrivere il funzionamento generale dell'applicazione, senza doversi preoccupare, ad esempio, di modellare i livelli inferiori del protocollo oppure alla mobilità dei nodi, perché di queste mansioni se ne occupa già Veins.

Con Veins ogni simulazione viene avviata eseguendo in parallelo OMNeT++ e SUMO e mettendoli in comunicazione tramite protocollo *TraCI* (Traffic Control Interface) che sfrutta una comunicazione realizzata con il protocollo TCP. TraCi consente, quindi, di combinare bidirezionalmente la simulazione del traffico di rete e di quello stradale.

L'interazione tra SUMO ed OMNeT++ avviene, tramite l'utilizzo di Veins, nel seguente modo: quando SUMO simula la partenza di un veicolo (nodo), Veins ne crea il corrispettivo modulo in OMNeT++. Quando il veicolo si muove in SUMO seguendo il percorso designato, Veins mantiene aggiornata la sua posizione all'interno

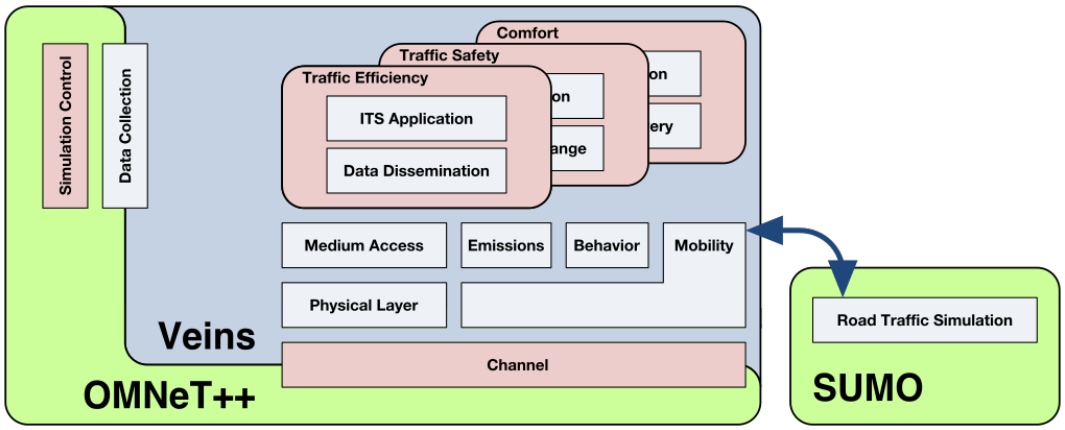

Figura 18: *Architettura ad alto livello di Veins*

della simulazione di OMNeT++. Una volta che esso raggiunge la sua destinazione è sempre Veins che si occupa dell'eliminazione del nodo in OMNeT++.

Veins_INET

Il framework scelto per questo elaborato è *Veins_INET*, un sotto-progetto di Veins che permette l'utilizzo combinato di quest'ultimo come modello di mobilità ed INET. Questo permette di poter utilizzare tutte le features di INET, precedentemente elencate, in una simulazione che utilizza Veins.

La motivazione che ha spinto l'utilizzo di questo sotto-progetto, al posto del framework Veins di base, è che Veins_INET permette di avere tutto lo stack di protocolli `IPv4` ed `IPv6`. Ciò ha permesso di riutilizzare le funzionalità di frammentazione del livello di rete che sono sono presenti nel framework Veins di base, non essendo stato sviluppato per supportare nativamente i servizi di task offloading.

In Figura 19 si può osservare quali sono i moduli presenti all'interno di un veicolo di una tipica simulazione Veins, mentre in Figura 20 si nota come all'interno del veicolo, i moduli presenti di una tipica simulazione Veins_INET, rappresentino tutto lo stack di protocolli di rete.

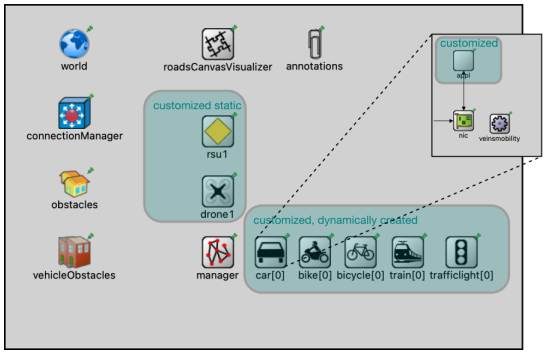

Figura 19: *Moduli Veins*

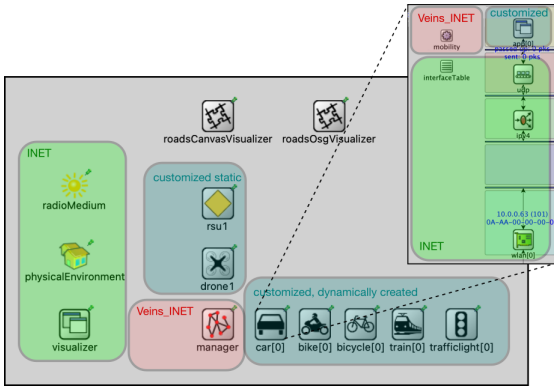

Figura 20: *Moduli Veins_INET*

3.2 Dettagli implementativi

Task Generator

Il task generator è un modulo semplice che contiene al suo interno parametri che configurano:

- il tempo iniziale di attesa per la generazione del task
- la capacità minima richiesta, in termini di risorse computazionali
- una threshold per l'attesa della computazione dei dati da parte dei veicoli
- la dimensione massima del pacchetto UDP

Task

Il task è un modulo generato dinamicamente a run-time all'interno del task generator. Esso contiene tutte le informazioni riguardanti il task che deve essere svolto. La scelta di crearlo come modulo e non come semplice classe C++, è stata pensata in un ottica futura, quando, possibilmente, i task da computare dovranno essere più di uno. Inoltre, essendo un modulo, si possono sfruttare tutte le potenzialità

del linguaggio `NED`, permettendo ad esempio lettura o configurazione di parametri specifici per singolo task.

Worker

Il worker è un modulo semplice che, come il task generator, contiene al suo interno parametri che configurano:

- la frequenza di CPU del veicolo: più è alta e minore sarà il tempo totale di computazione
- una percentuale che indica il livello di carico di lavoro attuale del veicolo
- la massima dimensione, in termini di risorse computazionali, che il veicolo può accettare
- una percentuale che indica la probabilità di essere ancora disponibile una volta finita la computazione

La percentuale che indica il livello di carico attuale del veicolo è una percentuale generata casualmente ogni volta che il parametro viene letto, così come la probabilità di essere ancora disponibile dopo una computazione.

Capitolo 4

Analisi dei risultati

In questo capitolo verranno presentati i risultati delle simulazioni effettuale al fine di valutare le prestazioni del protocollo di comunicazione di comunicazione presentato nel Capitolo 2 e dell'utilizzo dello standard 802.11p per la realizzazione della comunicazione inter-veicolo.

4.1 Scenario di simulazione

Le simulazioni sono state condotte considerando quattro principali metriche che verranno definite nella sezione successiva di questo capitolo. Queste metriche sono state analizzate (escluso il tempo totale di completamento) sia per il task generator, sia per i workers.

Le campagne di simulazione effettuate sono 4, ognuna con 100 ripetizioni, per un totale quindi di 400 ripetizioni di simulazione. L'obiettivo di fare così tante ripetizioni è quello di osservare un comportamento consistente dei veicoli e valutare l'efficacia del protocollo di comunicazione presentato nel capitolo 2 tra le varie simulazioni.

In tutte le simulazioni effettuate è stato assunto che i workers che ricevono l'*help message* siano sempre disponibili, con un carico computazionale equivalente ad 1/5

della dimensione totale del task. Questa scelta è stata fatta per garantire uniformità nei risultati attesi.

In tutte le simulazioni è stato assunto che un gruppo di veicoli, composto da 5 Workers, seguano sempre il Task Generator come mostrato in Figura 21, in modo tale che quest'ultimo abbia sempre possibilità di poter chiedere aiuto a qualcuno. Infine, per tutte le simulazioni è stata impostata la difficoltà del task a $2 \cdot 10^4$.

I parametri che sono stati fatti variare nelle diverse simulazioni eseguite sono:

- Grandezza del task

- Tempo di inizio del task

- Capacità di calcolo dei diversi workers

Nella prima campagna di simulazione la grandezza totale del task è stata impostata a 50`kB`. Nella seconda la sua dimensione è stata impostata a 500`kB`. Nelle ultime due, infine, è stata impostata prima ad 1`MB` ed in seguito a 5`MB`.

Il tempo di inizio del task è sempre compreso tra 30`s` e 40`s`, questo per permettere al simulatore di poter generare correttamente il mondo di simulazione ed i veicoli all'interno di esso.

La capacità di calcolo dei diversi workers viene fatta variare tra 1.20`GHz` e 2.40`GHz`, che è all'incirca la potenza di calcolo degli attuali single board computer presenti sul mercato.

Va specificato infine che, in tutte e 400 le simulazioni effettuate, il task viene sempre portato a termine, per cui la robustezza del protocollo di task offloading è confermata.

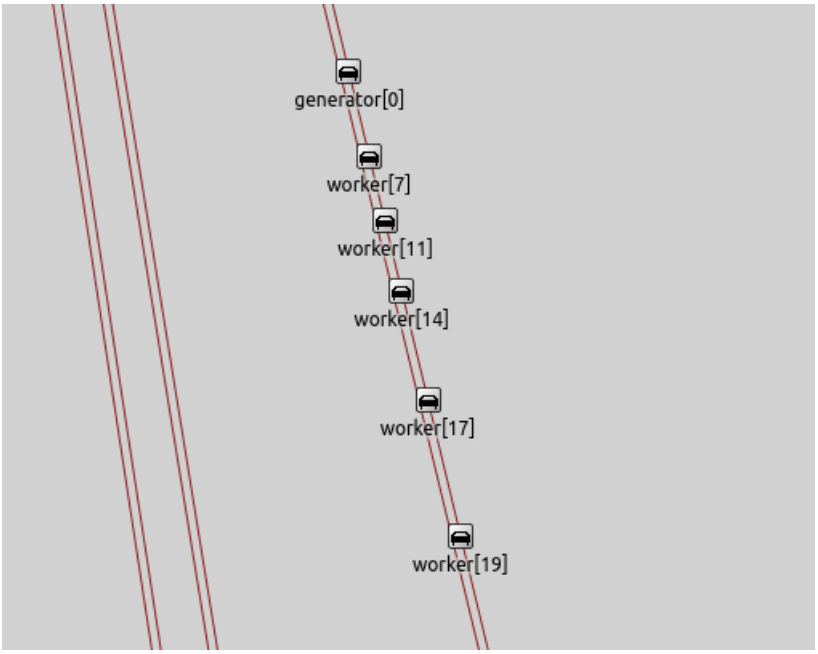

Figura 21: *Scenario di simulazione su OMNeT++*

Parametri di livello fisico

L'antenna utilizzata in tutte le simulazioni dai veicoli è *isotropica*, che ha la caratteristica di irradiare onde radio in tutte le direzioni con la stessa intensità. La modulazione utilizzata per tutti gli scenari di simulazione considerati è `QPSK` con coding rate 1/2 ed una velocità massima di trasmissione possibile (*bitrate*) di `6Mbps`.

4.2 Metriche di simulazione

Sono state considerate diverse metriche per valutare la robustezza e l'efficacia del protocollo di task offloading: il tempo totale di completamento del task, il numero medio di ritrasmissioni generate dai veicoli, il tempo di queueing time del task generator e dei workers ed infine il tempo di trasmissione di un chunk di dati.

Il tempo totale di completamento del task è il tempo totale che il task impiega a finire in base alla sua dimensione.

Il numero medio di ritrasmissioni generate dai veicoli ci permette di valutare, al crescere della dimensione del task, che impatto ha la frammentazione dei dati di task di dimensione maggiore all'interno dello scambio dei messaggi tra veicoli. In genere quello che ci si aspetta è che, in corrispondenza di task più grandi, ci sarà una frammentazione maggiore, per cui il numero di ritrasmissioni sarà più alto.

Il tempo di trasmissione di un chunk di dati è il tempo che il generatore, o il worker, impiegano per la trasmissione di un intero pacchetto di dati che può essere di dimensione variabile, ma al massimo di 65536B, che è la dimensione massima di un pacchetto UDP. Il queueing time, invece, è il tempo che un frame trascorre all'interno della coda dei frames a livello MAC (Medium Access Control) prima di essere trasmesso. Queste metriche ci si aspetta siano correlate direttamente, ossia più tempo i frames di un pacchetto rimangono in coda in attesa di essere processati, più alto sarà il tempo di trasmissione di un intero pacchetto di dati.

4.3 Analisi delle metriche

In questa sezione verranno analizzate nel dettaglio le metriche presentate nella sezione precedente analizzando i risultati ottenuti dalle simulazioni. Questa analisi fornirà un quadro completo delle prestazioni del protocollo ideato e presentato al capitolo 2 di questo elaborato.

4.3.1 Tempo totale di completamento del task

Osservando il grafico della Funzione di Distribuzione Cumulativa Empirica (ECDF) in Figura 22, si può notare come al crescere della dimensione del task aumenti anche il tempo di completamento.

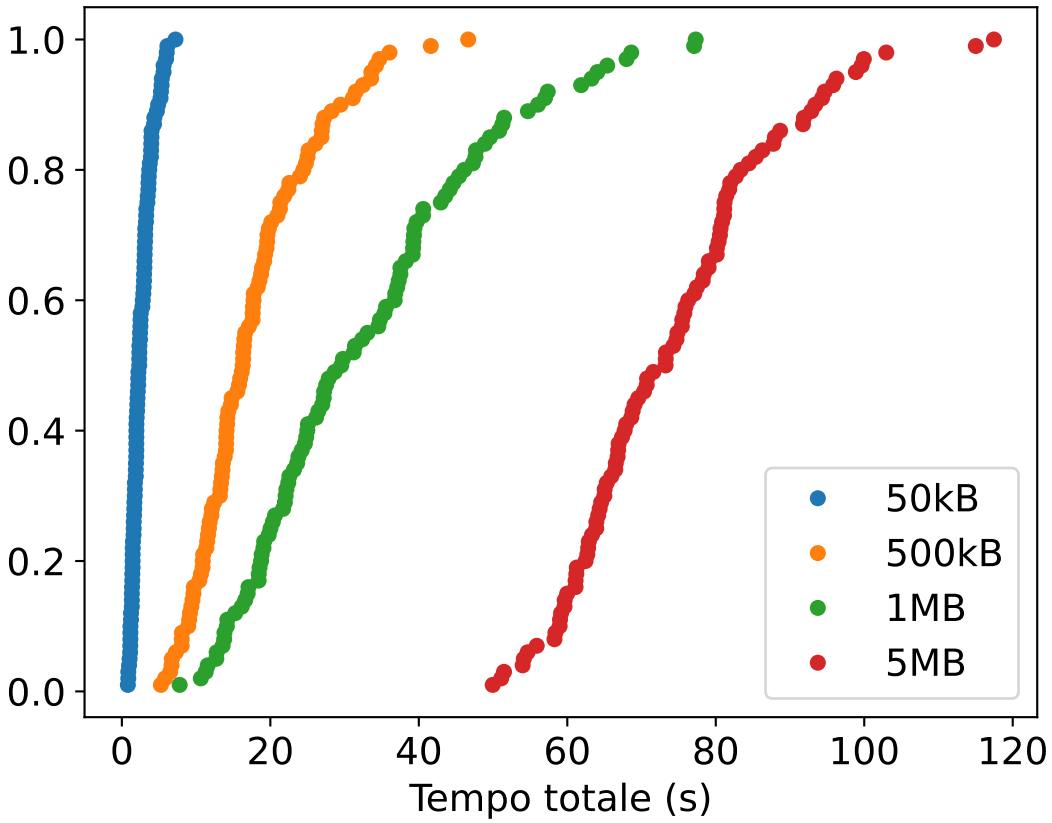

Figura 22: *ECDF del tempo totale di completamento per dimensione del task*

Concentrandosi invece sulle curve dell'ECDF, si può notare come i valori siano sempre più dispersi all'aumentare della dimensione del task. Quello che si evince dal grafico in Figura 22 infatti è che nel caso del task da `50kB` la curva cresce in maniera molto ripida, indicando che i valori che assume il tempo di completamento all'interno delle 100 simulazioni siano concentrati tutti attorno a pochi valori. Nel task da `5MB` invece possiamo osservare una dispersione maggiore dei valori del tempo di completamento assunti dalle diverse run di simulazione. Questo effetto di aumento della dispersione è dovuto sia al numero maggiore di pacchetti che sono generati, sia alle diverse capacità computazionali dei veicoli in termini di potenza del processore. Di conseguenza il range di tempo entro cui un task può finire è molto più ampio.

Dimensione task	50kB	500kB	1MB	5MB
Media	2.69s	17.79s	33.14s	74.16s
Deviazione standard	1.41s	8.26s	16.07s	13.81s

Tabella 1: *Tabella statistiche descrittive del tempo totale di completamento*

Osservando i valori nella Tabella 1 si può notare che, nel caso da 50`kB`, il tempo di completamento è in media 2.69`s`, nel caso da 500`kB` è di 17.79`s`, in quello da 1`MB` è di 33.14`s`, mentre per il task più grande da 5`MB` è di 74.16`s`. Confrontando inoltre le deviazioni standard con ciò che si osserva in Figura 22, si nota come i valori rispecchino ciò che viene visualizzato, ossia che a task di dimensione più piccola corrisponde una minor dispersione dei valori osservati (deviazione standard più bassa), mentre all'aumentare della dimensione del task, corrisponde una dispersione sempre più grande (deviazione standard più alta).

4.3.2 Numero medio di ritrasmissioni generate

Osservando i grafici in Figura 23, si può osservare come al crescere della dimensione del task, cresca anche il numero medio di ritrasmissioni generate all'interno della comunicazione. Questo risultato, conferma ciò che ci si aspetta e che è stato assunto nella sezione precedente.

Osservando i risultati ottenuti si nota innanzitutto come il numero medio di ritrasmissioni generate dai workers sia generalmente minore rispetto a quelle generate dal task generator. Inoltre l'importanza di questo grafico deriva dal fatto che ci permette di poter capire meglio l'aumento del tempo di completamento del task in base alla sua dimensione osservato precedentemente (si veda Figura 22). Ovviamente il tempo maggiore è dovuto soprattutto al numero maggiore di pacchetti generati in base alla

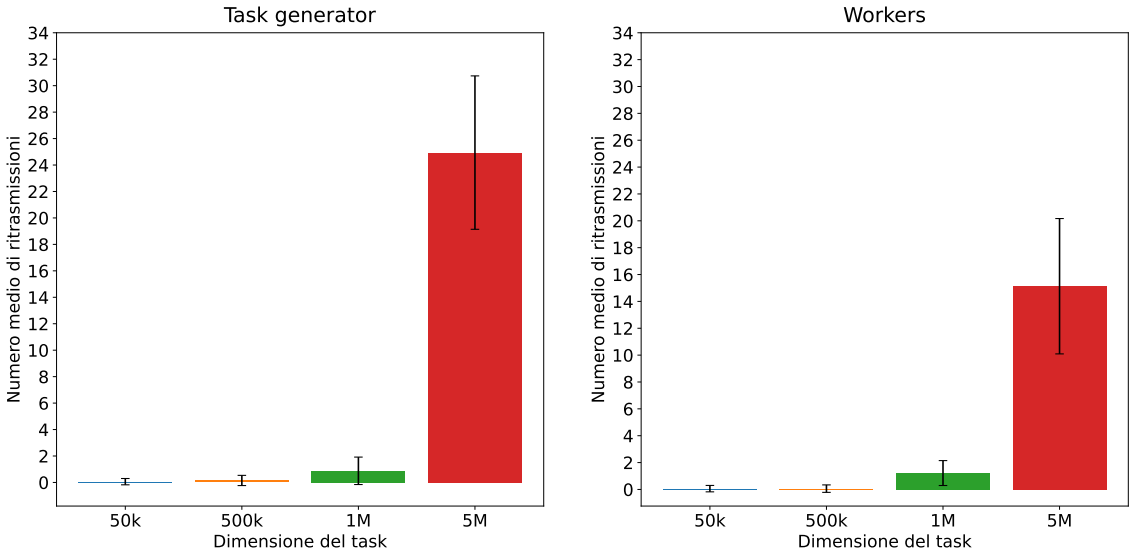

Figura 23: *Numero medio di ritrasmissioni per dimensione del task. A sinistra il numero medio di ritrasmissioni generate dal task generator, mentre a destra quelle generate dai workers*

dimensione del task, però, grazie all'analisi delle ritrasmissioni, si nota come sembra esserci anche una relazione diretta tra il tempo di completamento ed il numero di ritrasmissioni generate. Ciò significa che, osservando l'aumento totale del tempo di completamento nei diversi task, si nota anche un aumento di pacchetti di ritrasmissione generati. L'esempio più evidente è quello del task da `5MB` in cui il numero di ritrasmissioni nel task generator è di 25 circa pacchetti e quello dei workers sia di circa 16 pacchetti.

Ciò è dovuto al fatto che, oltre alla frammentazione della dimensione massima dei pacchetti `UDP` di 64`kB`, viene anche fatta frammentazione dei pacchetti a livello 3 dello stack di protocolli di rete. Di conseguenza bisogna tener conto che, scambiandosi un numero maggiore di pacchetti, viene generato un maggior scambio di messaggi all'interno della comunicazione, per cui, in fase di impostazione dei timer di ritrasmissione va trovato un valido trade-off tra minima e massima congestione del canale, tuttavia,

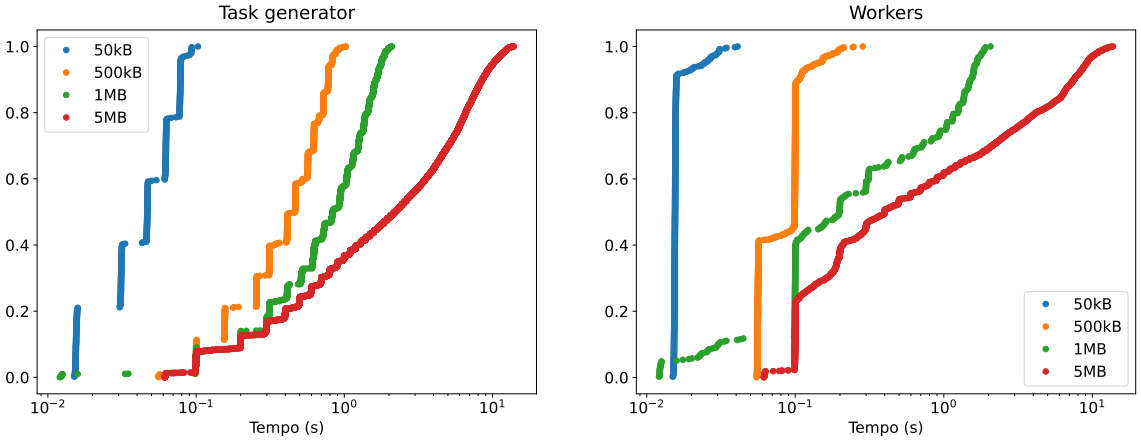

Figura 24: *ECDF del tempo di trasmissione di un chunk di dati per dimensione del task. A sinistra il tempo di trasmissione dal generatore ai workers, mentre a destra il tempo di trasmissione dai workers al generatore*

il mancato coordinamento tra le trasmissioni di diversi veicoli e l'elevato numero di pacchetti causano tempi di attesa maggiori che comportano lo scadere del timer di ritrasmissione.

4.3.3 Tempi di trasmissione dei singoli chunk di dato e di accodamento

In Figura 24 possiamo osservare i tempi di trasmissione di singoli chunk di dato inviati da parte del task generator ai workers e viceversa. La scala dell'asse delle ascisse di questi grafici è logaritmica, ciò ha permesso di espandere la visualizzazione delle curve dell'ECDF dei tempi di trasmissione che altrimenti sarebbe risultata troppo compressa.

Ciò che è interessante notare da questi grafici è il fatto che, da parte del task generator, vengono visualizzati degli step, ognuno dei quali rappresenta la trasmissione

di un singolo chunk di dati. Come menzionato nella sezione 4.1 di questo capitolo è stato assunto che ci siano sempre 5 veicoli disponibili al task generator per la computazione e che ogni veicolo dia 1/5 della dimensione totale del task in termini di disponibilità. Per cui, prendendo in esempio il task da 50`kB`, si può facilmente calcolare che il numero di chunk di dato da computare siano 5 e questo risultato lo ritroviamo esattamente nella visualizzazione del grafico osservando i 5 step.

All'aumentare della dimensione del task, ovviamente il numero di step all'interno del grafico aumentano perché ci sono più chunk di dato che devono essere scambiati. Inoltre alla fine della trasmissione dei chunk si può notare una leggera curva che rappresenta le ritrasmissioni all'interno della comunicazione.

Osservando il grafico dei workers invece, si può notare come ci sia una prima fase iniziale di trasmissione in cui essi rispondono con i chunk di dati che gli sono assegnati (anche qui possono essere presenti gli step se i chunk di dati assegnati ai worker sono più di uno) per la computazione e poi una curva che rappresenta, esattamente come nel task generator, le ritrasmissioni da parte dei workers.

In conclusione ciò che accomuna i due grafici è che, come ci si aspetta, all'aumentare della dimensione del task, aumenta anche il tempo di trasmissione di un singolo chunk sia che un pacchetto venga inviato dal generatore, sia che quest'ultimo sia inviato da un worker.

La peculiarità di questi grafici è che racchiudono, descrivono molto bene e confermano le assunzioni fatte nelle sottosezioni precedenti (4.3.1 e 4.3.2), ossia che all'aumentare della dimensione del task aumentano sia i tempi di completamento, sia il numero di ritrasmissioni generate.

In Tabella 2 possiamo osservare quali siano media, mediana e deviazione standard relative al tempo di trasmissione di un chunk di dati sia dei workers, sia del task generator per ogni dimensione di task analizzata.

Osservando i due grafici in Figura 25, invece, la prima cosa che si nota è che

Dimensione task	Task generator				Worker			
	50kB	500kB	1MB	5MB	50kB	500kB	1MB	5MB
Media	0.047	0.448	0.882	3.445	0.016	0.086	0.523	2.141
Deviazione standard	0.023	0.239	0.538	3.360	0.003	0.031	0.584	3.102
Mediana	0.047	0.464	0.836	2.207	0.015	0.099	0.199	0.399

Tabella 2: *Tabella statistiche descrittive del tempo di trasmissione di un chunk dati*

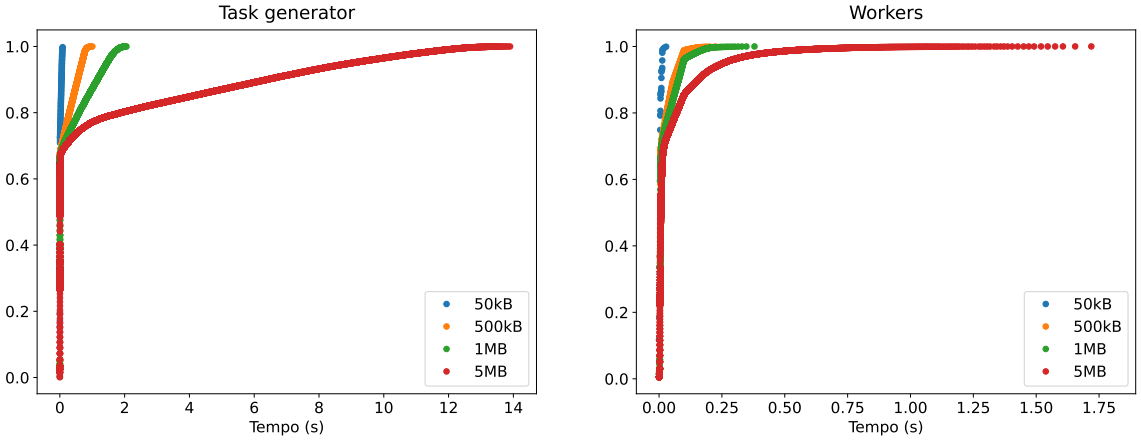

Figura 25: *ECDF del queueing time per dimensione del task. A sinistra quello del generatore, mentre a destra quello dei workers*

sull'asse delle ascisse sono presenti due scale diverse, nonostante i grafici sono riferiti alla stessa metrica. Nel caso del task generator si ha un valore massimo possibile di circa 14`s`, mentre nel caso dei workers il valore massimo raggiunto è di 1.75`s` circa. La presenza di due scale così differenti è dovuta al fatto che, quando inizia la comunicazione tra il task generator ed i veicoli, il canale è occupato quasi esclusivamente dal generatore che comincia ad inviare tutte le partizioni di dato da computare. Quando però i veicoli iniziano a terminare le computazioni ed a rispondere, entra in gioco la politica di *fairness* di `802.11p`, secondo la quale un veicolo non può tenere un

canale costantemente occupato, ma deve poter permettere anche agli altri veicoli che vogliono comunicare in rete di poterlo fare.

In conclusione, il tempo del task generator è decisamente più grande rispetto a quello dei workers, perché nel mentre che sta inviando le partizioni di dato da computare a questi ultimi, può essere che essi inizino a rispondere occupando il canale e facendo si che i pacchetti ancora da computare rimangano in coda più a lungo nel task generator.

Ovviamente, questo effetto di tempo di coda molto alto, si inizia a notare particolarmente quando la dimensione del task inizia a crescere. Ciò avviene perché il numero di pacchetti che devono essere scambiati tra i veicoli è maggiore, per cui anche il numero di ritrasmissioni sarà più alto. Di conseguenza con un accesso al canale condiviso e non sincronizzato è un comportamento coerente con quello che ci si aspetta.

Ciononostante va sottolineato che il tempo di attesa sia del task generator che dei workers nel 70% dei casi sia molto basso, circa 100ms. Questo tempo rimane costante per task di dimensione piccola, come nel caso del task di 50kB, però inizia a crescere man mano che la dimensione del task aumenta, arrivando a valori massimi di circa 1s nel caso da 500kB e circa 2s nel caso da 1MB, fino a raggiungere circa 14s di attesa nella coda dei pacchetti del task generator nel caso peggiore.

Nella coda dei pacchetti dei workers notiamo un comportamento abbastanza simile a quella del task generator, con tempi massimi di coda decisamente inferiori perché i workers non devono gestire tutti i pacchetti di dati da computare come il task generator, ma solamente le partizioni di dato che gli sono assegnate.

In Tabella 3 possiamo osservare, infine, quali siano media, mediana e deviazione standard relative al tempo di accodamento dei frames sia dei workers, sia del task generator per ogni dimensione di task analizzata.

Dimensione task	Task generator				Worker			
	50kB	**500kB**	**1MB**	**5MB**	**50kB**	**500kB**	**1MB**	**5MB**
Media	0.014	0.131	0.272	1.415	0.004	0.017	0.023	0.051
Deviazione standard	0.020	0.228	0.489	2.962	0.004	0.027	0.037	0.114
Mediana	0.005	0.005	0.005	0.005	0.003	0.005	0.005	0.008

Tabella 3: *Tabella statistiche descrittive del tempo di accodamento*

Occupazione del canale

A supporto dell'analisi condotta in questa sezione dell'elaborato (4.3.3), in Figura 26, può essere osservato come varia, nel tempo di simulazione, l'occupazione del canale per dimensione di task. Va specificato che l'inizio della trasmissione dei dati di computazione varia tra i 30 ed i 31 secondi. Questo tempo variabile è dovuto al tempo di attesa degli availability message da parte del task generator. Di conseguenza la densità crescente iniziale è dovuta allo scambio di *help message* ed *availability message* tra generatore e workers.

Osservando quindi nel dettaglio ciò che accade da circa 30.5`s` in poi, si può ritrovare il comportamento che è stato analizzato precedentemente. Quando i workers iniziano a rispondere con i dati computati, si nota che la loro densità di occupazione del canale inizia ad aumentare e contemporaneamente si osserva una riduzione di quella del task generator. Nel caso particolare del task da `5MB`, si distingue anche molto bene la fase di ritrasmissione dei pacchetti successiva a quella iniziale di trasmissione, che nel grafico corrisponde alla risalita della curva della densità del task generator.

Infine, si può notare come l'occupazione del canale nel caso di task più piccoli sia più concentrata nel tempo, rispetto a quella dei casi di task più grandi. Questo fenomeno avviene perché la quantità di dati e di pacchetti che deve essere scambiata è

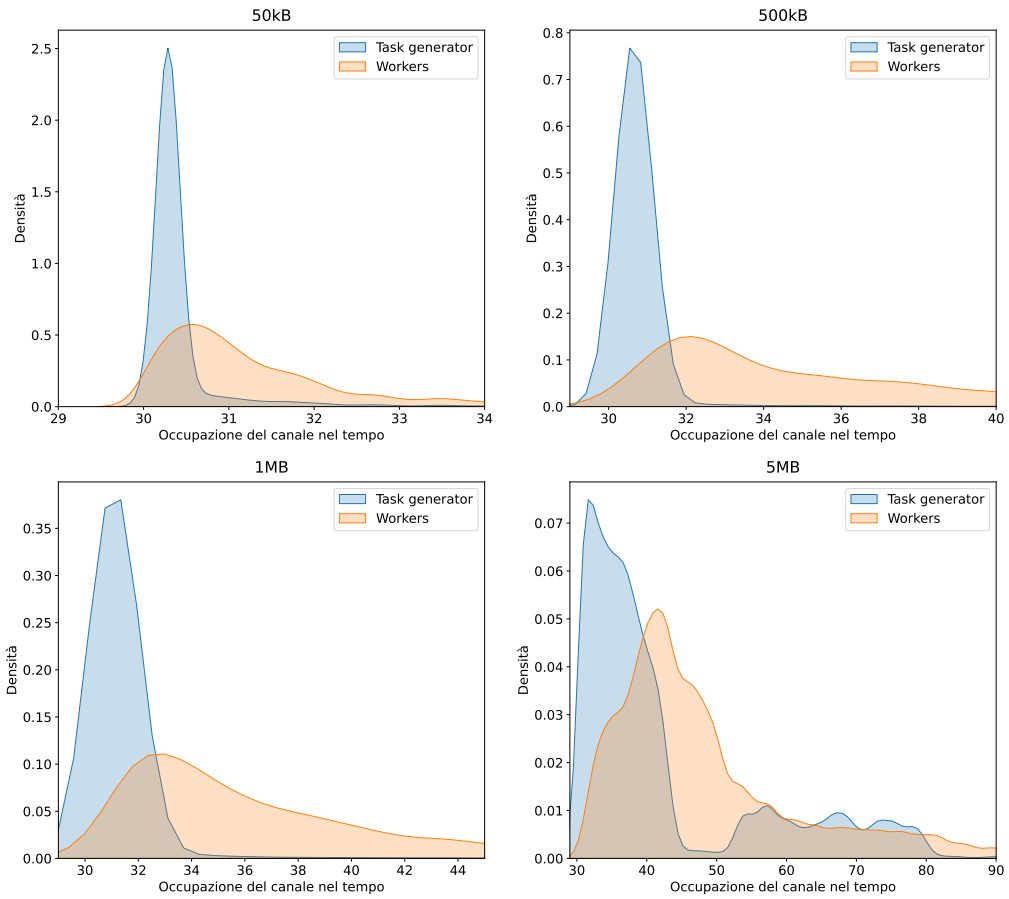

Figura 26: *Kernel density estimator dell'occupazione del canale nel tempo per dimensione di task*

inferiore, di conseguenza la trasmissione dei dati avviene in maniera molto più rapida, rispetto al tempo più alto richiesto dai task di dimensione maggiore.

4.4 Osservazioni

Per quanto riguarda l'analisi effettuata è emerso che la mancanza di un tempo di attesa per il trasferimento completo delle partizioni di dato dal task generator ai workers da parte del protocollo di comunicazione, può rappresentare un problema

per quanto riguarda il tempo totale di completamento del task. Ciò significa che, se i workers prima di iniziare a rispondere attendessero che il task generator finisca di trasmettere tutte le partizioni di dato, si otterrebbe un tempo di queueing time minore da parte del generatore, oltre che un tempo minore generale di completamento del task dovuto alle minori ritrasmissioni che si andrebbero a generare.

Ovviamente, l'introduzione di questo meccanismo introdurrebbe anche alcuni punti su cui porre particolare attenzione. Uno dei quali potrebbe essere il fatto che, se questo tempo di attesa venisse stimato da parte dei workers, la stima potrebbe non essere sempre corretta, per cui ci si potrebbe ritrovare ad aspettare un tempo inferiore di quello effettivamente richiesto, cioè ottenendo una situazione come quella attuale, oppure un tempo maggiore, ottenendo così un tempo totale di completamento più alto a discapito di un minor tempo di queueing time.

Conclusioni

In questo lavoro di tesi è stato progettato un protocollo di task offloading in un contesto di vehicular edge computing e condotta un'analisi preliminare delle prestazioni del protocollo. In fase di progettazione è stato tenuto conto dei principali scenari di un contesto volatile ed in costante cambiamento come quello delle reti di traffico veicolare. L'analisi è stata eseguita su dati ottenuti dalle simulazioni di OMNeT++ come specificato al capitolo 3, facendo variare diversi parametri come la dimensione del task e le capacità di calcolo dei diversi workers.

Osservando i risultati ottenuti, è emerso che il tempo di completamento del task aumenta in proporzione alla dimensione di quest'ultimo, ad un tasso nettamente superiore a quello atteso. Ciò avviene principalmente per una gestione non coordinata dell'accesso al canale, che ha come conseguenza il fatto di generare diverse ritrasmissioni dei pacchetti da parte dei veicoli. Questo avviene perché i veicoli workers, non attendendo il completamento dell'invio delle partizioni da parte del task generator, inviano subito il dato computato non appena la computazione è terminata, portando inevitabilmente a tempi di attesa in coda più lunghi, specialmente quando la dimensione del task inizia ad essere consistente. Nonostante questo i risultati hanno confermato la robustezza del protocollo di comunicazione, dato che il task, in tutte le simulazioni eseguite, riesce sempre ad essere portato a termine.

Dall'analisi effettuata sono anche emersi possibili sviluppi futuri per quanto riguarda il miglioramento delle prestazioni del protocollo presentato al capitolo 2. Un primo miglioramento per quanto riguarda le performance del tempo di computazione può essere ottenuto, ad esempio, facendo attendere i workers il tempo di invio di tutti i pacchetti da parte del task generator, prima che essi inizino a rispondere. Ciò porterebbe possibilmente ad un numero ridotto di ritrasmissioni da parte di tutti i veicoli ed un miglioramento generale del queuing time. Un altro possibile sviluppo futuro potrebbe essere quello di integrare la rete veicolare considerata (V2V), con un paradigma di comunicazione V2I. Questo permetterebbe un tempo di computazione ancora migliore rispetto a quello osservato, estendendo la computazione dei dati anche alle RSU presenti a bordo strada e non solo ai veicoli. Ciò implicherebbe anche una maggiore potenza di calcolo disponibile, rispetto a quella piuttosto limitata di questi ultimi.

Ringraziamenti

Vorrei esprimere la mia sincera gratitudine verso la mia famiglia, coloro che mi hanno supportato in ogni decisione presa e mi hanno permesso di non arrendermi mai, anche quando andare avanti di un singolo passo mi sembrava una cosa impossibile.

Ad Alessia, nonostante ad oggi siano cambiate molte cose, sei stata una persona fondamentale per il mio percorso. Un rifugio sicuro nella quale tutte le paure ed insicurezze sparivano ed una fonte di ispirazione da cui partire per dare sempre il meglio di me. Grazie per avermi permesso di non lasciar mai nulla al caso, per il tuo sostegno unico e per il tuo amore incondizionato.

Desidero ringraziare inoltre il mio relatore, Christian Quadri, per la sua infinita pazienza, disponibilità ed incredibile umanità, oltre che per avermi trasmesso una enorme passione per il suo specifico ambito di ricerca.

Ringrazio infine tutti i miei amici che mi hanno sempre sostenuto, accettato e voluto bene per ciò che sono oggi e non per ciò che sono stato ieri o che un domani io possa diventare.

A tutti voi, grazie di cuore.

Bibliografia

[1] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Future Generation Computer Systems*, vol. 97, pp. 219–235, 2019.

[2] R. Xie, Q. Tang, Q. Wang, X. Liu, F. R. Yu, and T. Huang, "Collaborative vehicular edge computing networks: Architecture design and research challenges," *IEEE Access*, vol. 7, pp. 178942–178952, 2019.

[3] F. Dressler, C. F. Chiasserini, F. H. Fitzek, H. Karl, R. L. Cigno, A. Capone, C. Casetti, F. Malandrino, V. Mancuso, F. Klingler, and G. Rizzo, "V-edge: Virtual edge computing as an enabler for novel microservices and cooperative computing," *IEEE Network*, vol. 36, no. 3, pp. 24–31, 2022.

[4] W. Fan, Y. Su, J. Liu, S. Li, W. Huang, F. Wu, and Y. Liu, "Joint task offloading and resource allocation for vehicular edge computing based on v2i and v2v modes," *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 4, pp. 4277–4292, 2023.

[5] D. Jiang and L. Delgrossi, "Ieee 802.11p: Towards an international standard for wireless access in vehicular environments," in *VTC Spring 2008 - IEEE Vehicular Technology Conference*, pp. 2036–2040, 2008.

[6] M. van Eenennaam, A. van de Venis, and G. Karagiannis, "Impact of ieee 1609.4 channel switching on the ieee 802.11p beaconing performance," in *2012 IFIP Wireless Days*, pp. 1–8, 2012.

[7] A. Varga, "OMNeT++," in *Modeling and Tools for Network Simulation* (K. Wehrle, M. Güneş, and J. Gross, eds.), pp. 35–59, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[8] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner, "Microscopic traffic simulation using sumo," in *The 21st IEEE International Conference on Intelligent Transportation Systems*, IEEE, 2018.

[9] C. Sommer, R. German, and F. Dressler, "Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis," *IEEE Transactions on Mobile Computing*, vol. 10, pp. 3–15, January 2011.