Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Scienze e Tecnologie Informatiche

SIMULAZIONE DI MOBILITÀ IN AMBITO VEICOLARE

Relazione Finale in Reti di Calcolatori

Relatore:
GABRIELE D'ANGELO

Presentata da:
ANTONIO MAGNANI

Sessione II
Anno Accademico 2011-2012

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

La continua evoluzione delle tecnologie wireless, l'abbattimento dei costi e l'esponenziale diffusione di terminali senza fili stanno determinando un cambiamento radicale nel modo di concepire le moderne reti informatiche. Nell'arco di pochi anni infatti abbiamo assistito, o per meglio dire partecipato, ad una vera e propria metamorfosi, sia da un punto di vista scientifico e tecnologico, sia da un punto di vista prettamente sociale. L'esplosione della telefonia cellulare, l'avvento dei primi laptop e la sempre più proliferante diffusione di smartphone e tablet sono gli esempi più classici per descrivere quanto le reti mobili abbiano velocemente penetrato la società moderna. Al giorno d'oggi è normale per qualsiasi utente utilizzare il proprio PDA o smartphone al fine di inviare o ricevere mail, collegarsi e navigare su Internet in aeroporti, stazioni, metropolitane, ristoranti, caffè, parchi, piazze, oppure utilizzare terminali GPS per visualizzare mappe, percorsi o informazioni turistiche. Nel 2011 il numero di abbonamenti alla rete mobile cellulare ha raggiunto i 6 miliardi di utenze, corrispondenti ad una penetrazione globale dell'86%, mentre, per quanto riguarda la banda larga mobile, è stata superata la soglia del miliardo e mezzo di abbonamenti con una crescita del 40% rispetto al 2010 [1]. Le reti mobili hanno dunque cambiato la nostra vita sotto molteplici aspetti ma la costante evoluzione, la diffusione e le prestazioni delle nuove tecnologie wireless rendono quasi impredicibile il futuro di questa tipologia di rete.

Nel corso degli anni lo sviluppo delle connessioni tra dispositivi wireless si è concentrato sull'utilizzo di infrastrutture fornite da Internet Service Pro-

vider (ISP) oppure all'interno di reti private; ad esempio, nel caso di reti cellulari, la connessione tra due telefoni viene realizzata tramite i Base Station Controller (BSC), che svolgono il ruolo di ponte con i Mobile Switching Center (MSC); un ulteriore esempio può essere dato da un laptop connesso ad Internet tramite l'utilizzo di un access point wireless: in entrambi i casi è evidente la presenza di un'infrastruttura di rete [2]. Per quanto le reti di questo tipo forniscano un ottimo modo per ottenere servizi di rete, la creazione di infrastrutture ed i costi associati alla realizzazione di esse può essere anche molto elevato. Vi sono inoltre situazioni in cui l'infrastruttura richiesta dall'utente non è disponibile, non può essere installata oppure non può essere installata e configurata in tempo utile in una determinata area geografica. In queste situazioni per garantire connettività e servizi di rete si rende necessario l'utilizzo di una rete mobile ad-hoc [3]. Per queste ragioni, e con il significativo svilupparsi di nuove tecnologie e standard, possibili soluzioni alternative hanno via via guadagnato maggior attenzione. Queste si focalizzano sull'idea di dispositivi mobili in grado di collegarsi reciprocamente all'interno di un raggio di trasmissione, tramite una configurazione automatica e mediante la creazione di una rete ad-hoc che sia allo stesso tempo flessibile e potente. In questo modo non solo i nodi mobili possono comunicare tra loro, ma anche ricevere servizi Internet attraverso nodi gateway, estendendo a tutti gli effetti sia la rete che i servizi Internet ad aree non coperte da infrastrutture. Queste reti possono anche divenire l'unica soluzione in ambienti difficili quali possono essere uno scenario militare oppure nei casi di *"disaster recovery"* e in tutti i contesti dove, come già affermato, non sia possibile predisporre un'infrastruttura. I network ad-hoc quindi acquisiscono sempre maggior importanza, diventando terreno fertile per la ricerca di soluzioni tecnologiche in grado di supportare le criticità intrinseche di questa tipologia di rete, stimolando così lo sviluppo di nuovi progetti da parte del mondo accademico nonché di quello industriale [3]. La progettazione di reti mobili ad-hoc (MANET, Mobile Ad-hoc NETwork) e dei suoi protocolli deve considerare numerose problematiche [2] legate in particolare al routing, non

avendo una topologia fissa ed essendo i nodi totalmente liberi di muoversi, e, ad esempio, al problema del broadcast storming [4].

Tra le reti MANET sempre più interesse stanno acquisendo le reti veicolari ad-hoc (VANET, Vehicular Ad-hoc NETwork). I nodi costituenti questa tipologia di rete sono prevalentemente veicoli (pubblici o privati). Possono prendere parte a questa rete anche dispositivi fissi, chiamati Road Side Unit (RSU), posti lungo strade o in posizioni utili (incroci, caselli, strade ad alto traffico) il cui scopo è quello di estendere la rete ad-hoc ed inoltrare dati utili ai veicoli in avvicinamento, o ad altri RSU, al fine di scambiare informazioni sul traffico oppure, ad esempio, sulla sicurezza e la viabilità di una particolare arteria stradale. Le VANET offrono un nuovo modo di interpretare la mobilità veicolare e le possibilità applicative sono molteplici [5]; tra gli obiettivi principali di ricerca, industriale ed accademica, un ruolo di primaria importanza è rivestito dalla sicurezza stradale: le comunicazioni tra veicoli possono informare il conducente riguardo alle condizioni di un particolare percorso allertandolo anticipatamente nel momento in cui vi siano possibili elementi di rischio oppure, in caso di incidente o pericolo imminente, le vetture coinvolte potrebbero inviare un messaggio broadcast ad altri veicoli nelle vicinanze al fine di evitare possibili collisioni o, comunque, invitare il conducente ad una maggiore attenzione mediante apposite segnalazioni visive od uditive. Un elenco dei principali domini applicativi nelle VANET è il seguente [5]:

- **Servizi informativi generali**: servizi per i quali la perdita di messaggi, o il loro arrivo ritardato, non costituiscono un problema. Esempi di queste applicazioni possono essere:
  - informazioni di carattere atmosferico;
  - informazioni generiche sul traffico stradale;
  - pubblicità;
  - intrattenimento.
- **Servizi informativi di sicurezza**: servizi per i quali la perdita di messaggi, o il loro arrivo ritardato, può compromettere la sicurezza

della guida o può rendere inutile l'applicazione. Esempi di queste applicazioni possono essere:

  - condizioni dell'asfalto;
  - comportamento anomalo di veicoli.

- **Controllo individuale dei tragitti**: servizi che forniscono messaggi al conducente per migliorare la sicurezza della guida. Esempi di queste applicazioni possono essere:

  - informazioni su veicoli fermi;
  - informazioni su lavori in corso.

- **Controllo globale dei tragitti**: servizi che cercano di regolare il flusso generale dei veicoli al fine di migliorare per tutti le condizioni di sicurezza e velocità di guida:

  - distribuzione e bilanciamento dei flussi di traffico;
  - percorsi alternativi per la salvaguardia ambientale.

La valutazione di protocolli e applicazioni idonee alle VANET potrebbe essere effettuata tramite sperimentazioni reali; tuttavia i costi per ottenere risultati significativi, in termini di risorse e tempistiche, sarebbero eccessivi. I processi di installazione, configurazione ed eventuale correzione dei dispositivi necessari alla realizzazione della rete possono essere estremamente lunghi e dispendiosi in quanto ogni nodo dovrà essere predisposto per poter inviare e ricevere informazioni. Nel caso in cui si vogliano testare protocolli su scenari macroscopici, quali possono essere una metropoli o un'autostrada, tempi e costi aumentano significativamente all'aumentare del numero di veicoli [6, 7, 8, 9].

La simulazione è l'imitazione del funzionamento di un reale processo o sistema nel tempo [10], più specificatamente intenderemo la simulazione come quella branca dell'informatica che, tramite modellazione della realtà, si occupa di valutare e prevedere lo svolgersi dinamico di una serie di eventi o

processi susseguenti all'imposizione di certe condizioni da parte dell'analista o dell'utente [WIKI]. La simulazione è uno strumento efficace, di più semplice realizzazione e poco dispendioso per la valutazione di protocolli e architetture per queste reti. Tramite la simulazione inoltre è possibile effettuare, senza troppi problemi, valutazioni legate alla scalabilità, tuttavia i risultati ottenibili tramite il processo simulativo dipendono fortemente dall'accuratezza del modello utilizzato nel rappresentare la realtà [6, 7, 11]. Bisogna inoltre considerare che molto spesso, ovviamente, i fenomeni e gli scenari del mondo reale non possono essere riprodotti e necessitano di approssimazioni, in ogni caso la simulazione è un ottimo punto di partenza in questo emergente campo di ricerca.

Gli obiettivi di questa tesi sono lo studio di MoViT (Mobile network Virtualized Testbed) [12]: software suite per l'emulazione di reti MANET \VANET che verrà descritta dettagliatamente al capitolo 1, del suo "generatore di mobilità" SUMO (Simulation of Urban Mobility) [13] descritto al capitolo 2, e dell'analisi delle problematiche ad esso associate per cui verranno proposte alcune soluzioni riguardanti, in particolare, re-instradamento di veicoli e TraCI (Traffic Control Interface): interfaccia utilizzata da SUMO per realizzare un'interazione client-server tra simulatore ed applicativi esterni. Di quest'interfaccia verrà realizzata un'implementazione in linguaggio ANSI C, linguaggio maggiormente performante rispetto a Python, attualmente utilizzato.

Capitolo 1

MoViT

Le reti veicolari rappresentano una notevole sfida per la progettazione e modellazione di protocolli e applicativi; le loro performance infatti dipendono da molteplici parametri quali la propagazione dei segnali elettromagnetici, il routing di pacchetti, lo schema di mobilità e ovviamente dalla configurazione hardware e software dei dispositivi on-board. Ognuno di questi parametri deve essere incluso nel modello. La simulazione rappresenta correttamente aspetti quali il canale, il routing e la mobilità, presenta tuttavia carenze per quel che concerne la modellazione hardware e software. Nelle VANET questo è un limite significativo in quanto molti protocolli e applicazioni veicolari sono limitati da un punto di vista hardware. I testbed reali rimangono dunque gli strumenti più vicini all'approssimazione della realtà in quanto possono riprodurre, in piccola scala, i problemi di una distribuzione massiccia, purtroppo i testbed sono dispendiosi da realizzare e da mantenere [6, 7]; possono inoltre comportare difficoltà per la ripetibilità dell'esperimento per via delle difficoltà nel coordinamento dei conducenti e per fattori esterni quali semafori e traffico.

MoViT, emulatore di reti mobili wireless, si propone di realizzare un approccio a metà strada: utilizza macchine virtuali (VM) che eseguono codice reale su core dedicati in grado di eseguire qualsiasi applicazione e dunque di sperimentare tutte le problematiche relative allo sviluppo di codice reale. La

mobilità veicolare, che può anche essere la riproduzione di un esperimento reale, è simulata mentre le caratteristiche relative alla rete (propagazione, accesso al canale ecc.) sono opportunamente modellate in tempo reale. Lo scopo di MoViT è quindi quello di consentire lo sviluppo e la valutazione di protocolli e applicazioni su reti mobili a larga scala pur mantenendo realismo dal punto di vista hardware e software. Questo obiettivo viene raggiunto virtualizzando i nodi e modellando le caratteristiche della rete mentre ogni dispositivo virtuale esegue applicazioni su un proprio sistema operativo utilizzante una pila protocollare Internet reale. MoViT fornisce e replica tutti i dettagli di modellazione (mobilità, parametri di rete, vincoli urbani) ai vari nodi con il fine di offrire un ambiente realistico e totalmente controllabile nel quale sviluppare e valutare applicazioni mobili. Altro obiettivo che si prefigge MoViT è rendere facilmente implementabili, su dispositivi reali, i software sviluppati.

1.1 Architettura di MoViT

MoViT è un sistema distribuito eseguito su diverse macchine fisiche chiamate host. Ogni host ha due interfacce fisiche connesse a due differenti reti: l'**experimental network** e la **control network**. L'*experimental network* ha lo scopo di emulare il comportamento di una rete mobile, ogni host mantiene e gestisce un insieme di macchine virtuali, ciascuna rappresentante un nodo della rete sperimentata. MoViT riproduce la connettività della rete emulata filtrando i flussi di pacchetti tra coppie di macchine virtuali. Ad esempio, nella rete in Figura 1.1 i nodi 8 e 21 sono vicini (*one-hop neighbors*) e possono scambiarsi informazioni reciprocamente; al contrario i nodi 1 e 2 non sono vicini e la comunicazione tra essi sarà bloccata. Se la rete emulata è mobile allora è lecito aspettarsi continui cambiamenti della sua topologia e quindi della sua connettività. Per seguire questi cambiamenti dinamici le informazioni riguardanti la connettività devono essere distribuite a tutti gli host. La *control network* facilita ciò e, in aggiunta, consente l'ac-

cesso senza restrizioni alle macchine virtuali. Pertanto, MoViT fornisce un set di macchine virtuali totalmente controllabili la cui connettività si evolve nel tempo rispettando quella della rete emulata. La distribuzione dei dati di connettività è gestita dall'*experiment controller* tramite la *control network*. L'*experiment controller* è una raccolta di software che, oltre a distribuire dati di connettività, gestisce anche l'esecuzione degli esperimenti.

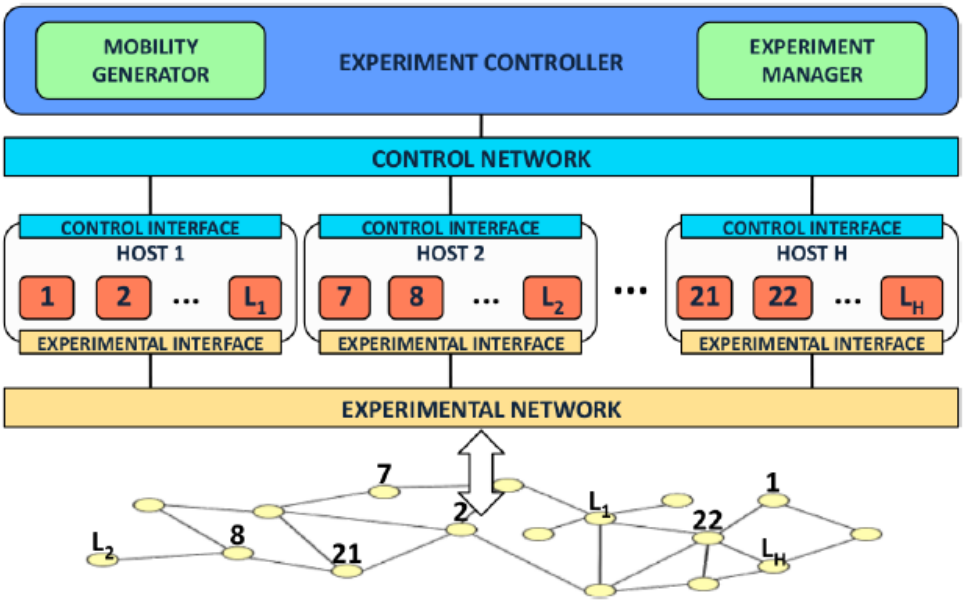

Figura 1.1: MoViT: Architettura *(tratto da [12])*

1.2 Experiment Controller

L'*experiment controller* gestisce tutte le funzionalità necessarie all'esecuzione di un esperimento. Queste funzionalità includono l'istanziazione delle macchine virtuali, l'inizializzazione delle applicazioni sperimentali, la generazione e distribuzione dei dati di mobilità e la raccolta dei risultati della sperimentazione. Possiamo classificarne le funzionalità in due categorie.

1.2.1 Gestione della mobilità

L'*experiment controller* è responsabile della generazione dei dati di mobilità, li contestualizza all'ambiente dove l'esperimento viene eseguito e li ridistribuisce ai vari host. In particolare l'*experiment controller* produce snapshot sequenziali ad intervalli periodici rappresentanti la mobilità in un determinato istante. La mobilità può essere simulata o riprodotta da una traccia preesistente. Data la natura distribuita di MoViT si rende necessario considerare un lasso di tempo sufficiente tra la generazione di un determinato snapshot di mobilità e la sua effettiva emulazione nell'*experimental network*, consentendo così la distribuzione dei dati di mobilità a tutti gli host. Per questo motivo ad ogni snapshot viene assegnato un *"tempo di applicazione"* allo scadere del quale la connettività di rete, corrispondente allo snapshot, verrà emulata sulla rete sperimentale. Per esempio, si assuma che la mobilità sia composta da snapshot effettuati ad ogni secondo; il primo snapshot avrà un tempo di applicazione corrispondende all'istante in cui l'esperimento avrà inizio, tutti i seguenti avranno un tempo di applicazione incrementale pari ad un secondo ognuno.

Come precedentemente affermato, ogni VM rappresenta un nodo della rete emulata. L'*experiment controller* tiene traccia delle corrispondenze tra gli identificativi dei nodi emulati e delle VM, assicurando consistenza agli esperimenti.

1.2.2 Gestione dell'esperimento

Un esperimento su MoViT è suddivisibile in tre differenti fasi: istanziazione, sperimentazione, raccolta e valutazione dei risultati. Per eseguirle viene utilizzato l'Orbit Control and Management Framework [OMF]. OMF è un framework utilizzato per il controllo, la gestione della strumentazione e di piattaforme di sperimentazione; è uno strumento estremamente versatile e stabile che fornisce un controllo totale oltrechè varie funzionalità di gestione e misurazione [14]. Le modifiche ad OMF, necessarie per integrarlo a MoViT,

sono minime; la più importante di queste è l'obbligo per OMF di assicurare che una macchina virtuale sia attiva in fase di istanziazione ed in caso contrario avviarla. Questo requisito viene implementato tramite semplici richieste alle socket delle varie *control interface* nei vari host connessi alla *control network*. L'inizializzazione dell'esperimento si compone di: una valutazione delle risorse necessarie all'esecuzione dell'esperimento (host e macchine virtuali), l'istanziazione di queste risorse ed infine l'avvio del software distribuito di emulazione. Successivamente OMF, attraverso la *control network*, esegue le applicazioni sperimentali sulle macchine virtuali e raccoglie i risultati una volta terminata la sperimentazione. Le applicazioni gestite con OMF offrono la possibilità di raccogliere risultati sperimentali direttamente mentre l'esperimento viene computato, consentendo la visualizzazione di tabelle e grafici in tempo reale.

1.3 Configurazione degli host

Come precedentemente affermato su ogni host viene eseguito un set di macchine virtuali rappresentanti i nodi della rete. Le macchine virtuali possono essere eseguite utilizzando qualsiasi tecnica di virtualizzazione. In ogni host il **Connectivity Manager** modella la connettività tra macchine virtuali; in particolare di ciò si occupa un componente specifico chiamato **Network Shaper** che, mediante l'applicazione di *tassi di rilascio (drop rates)* e *ritardi (delays)*, esegue il filtraggio tra macchine virtuali secondo le regole di connettività generate dal **Channel Module**, un altro modulo facente parte del *Connectivity Manager*. Il *Channel Module* genera queste regole basandosi sui dati di mobilità generati dall'*experiment controller*.

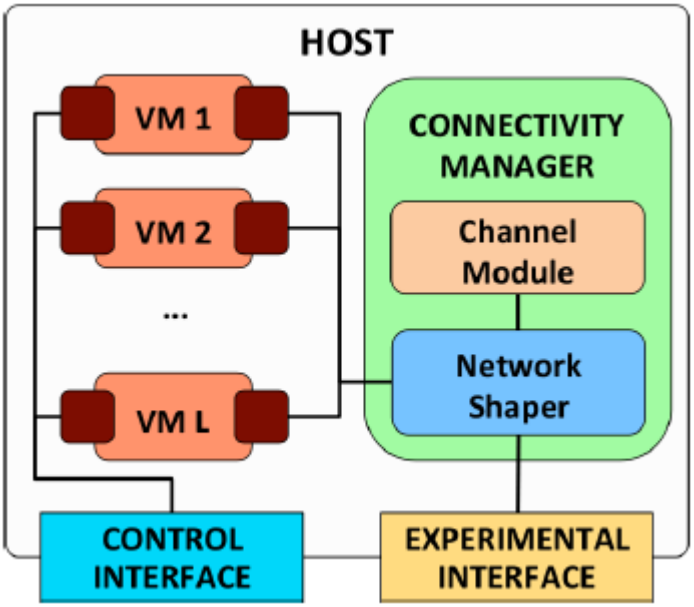

Figura 1.2: Configurazione host in MoViT *(tratto da [12])*

1.3.1 Connectivity Manager

Lo scopo di questo modulo è modellare la connettività di rete tra macchine virtuali. Ciò è possibile attraverso l'applicazione di politiche di filtraggio dei flussi di pacchetti tra tutte le coppie sorgente-destinazione. Questo filtraggio consiste nell'utilizzo di tassi di rilascio artificiali e ritardi addizionali. I tassi di rilascio e i ritardi devono necessariamente essere modificati in maniera dinamica secondo la topologia di rete che deve essere riprodotta. Il *Connectivity Manager* è suddiviso in due differenti moduli: il *Channel Module* che ha il compito di produrre le regole di connettività relative alla mobilità ed il *Network Shaper* il cui scopo è rimodellare la rete tramite filtraggio dei flussi di pacchetti.

1.3.1.1 Channel Module

Al fine di realizzare connettività tra i nodi della rete, le regole di connettività (tassi di rilascio e ritardi addizionali) devono essere presenti in ogni host partecipante all'esperimento. Il tasso di rilascio e il ritardo sperimentato in

un collegamento wireless tra 2 nodi, dipende da vari fattori quali la perdita di percorso, il numero di nodi contendenti il canale e le interferenze esterne. Le interferenze esterne non possono essere direttamente computate e necessitano un'approssimazione tramite modelli statistici. Differentemente, la perdita di percorso e la concorrenza al canale possono essere direttamente dedotte dalla mobilità dei nodi in considerazione dell'ambiente in cui si stanno muovendo.

Ogni host necessita soltanto delle regole relative alle macchine virtuali eseguite su di esso. Assumiamo dunque che le macchine virtuali siano uniformemente distribuite nei vari host con L macchine virtuali per ogni host. Allora $L=N/H$, dove H è il numero totale di host e N è il numero di nodi che partecipano all'esperimento. In ogni host avremo quindi bisogno di $L*N$ regole di connettività. MoViT decentralizza la computazione delle regole di connettività: il *Channel Module* di ogni host computerà le sole $L*N$ regole necessarie. Risulta evidentemente come sia più conveniente effettuare $O(L*N)$ computazioni su H macchine piuttosto che eseguirne $O(N^2)$ su una singola macchina. Per poter realizzare questa decentralizzazione e poter ridurre il tempo necessario alla consegna dei dati di mobilità, i *Channel Module* sono organizzati in un albero strutturato. L'*experiment controller* consegnerà i dati di mobilità solamente al primo livello dell'albero; successivamente ogni *Channel Module* inoltrerà i dati al livello sottostante. In questo modo il tempo necessario a distribuire i dati di mobilità è $O(logH)$ invece di $O(H)$ ottenibile con una struttura piatta [12].

1.3.1.2 Network Shaper

MoViT implementa uno strumento in grado di riprodurre la connettività ed i continui cambiamenti di topologia di una rete dinamica. Questo strumento è il *Network Shaper* che utilizza meccanismi di filtraggio per raggiungere il suo scopo. Per poter attuare un efficace filtraggio dei flussi di pacchetti a livello di collegamento e per poter gestire i continui cambiamenti dei set di regole di connettività, MoViT implementa un modulo kernel che può essere associato ad una politica di accodamento (*QDisc, Queueing Discipline*) [15].

Le *QDisc* permettono al modulo di applicare il filtraggio all'interno del kernel, ponendosi tra i driver e la network stack. In Figura 1.3 viene presentata l'architettura del *Network Shaper* e la sua interazione con il *Channel Module*.

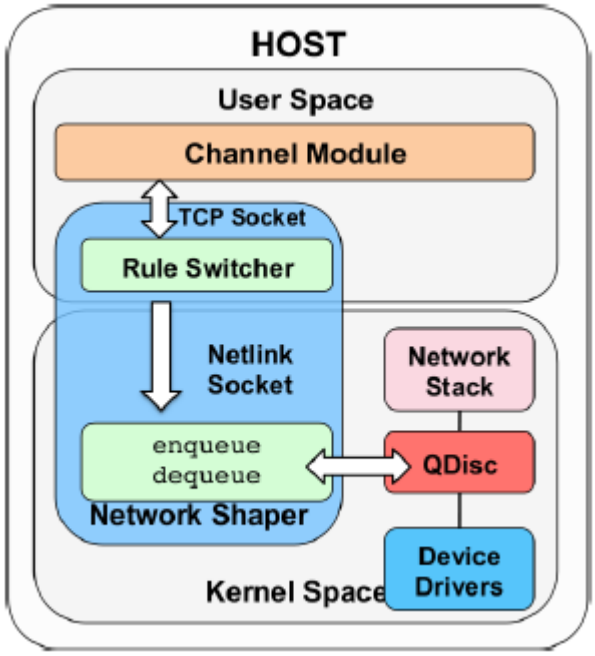

Figura 1.3: Network Shaper ed interazione con il Channel Module *(tratto da [12])*

Filtraggio. Modellare la rete richiede un'architettura di sistema che sia in grado di prendere il controllo di tutto il traffico generato dalle macchine virtuali sull'*experimental interface*. Questo livello di granularità può essere ottenuto attraverso l'utilizzo del sistema delle *QDisc* implementato nel kernel linux. Le *QDisc* sono meccanismi che permettono l'interazione tra la *network stack* ed i driver del dispositivo; le *QDisc* gestiscono il flusso di pacchetti tra hardware e software catturando tutti gli interrupt generati e collengandoli alle politiche di accodamento registrate. Registrando il *Network Shaper* come politica di accodamento sulle *QDisc* che gestiscono le *experimental interface* delle macchine virtuali, tutto il traffico di rete in ingresso ed in uscita dovrà

essere processato dal *Network Shaper*. Il posizionamento del modulo permette al filtraggio di avvenire al livello più basso possibile cioè esattamente prima che i pacchetti siano trasferiti all'interfaccia fisica. La Figura 1.4 sintetizza il flusso di pacchetti tra macchine virtuali sullo stesso host e su host differenti.

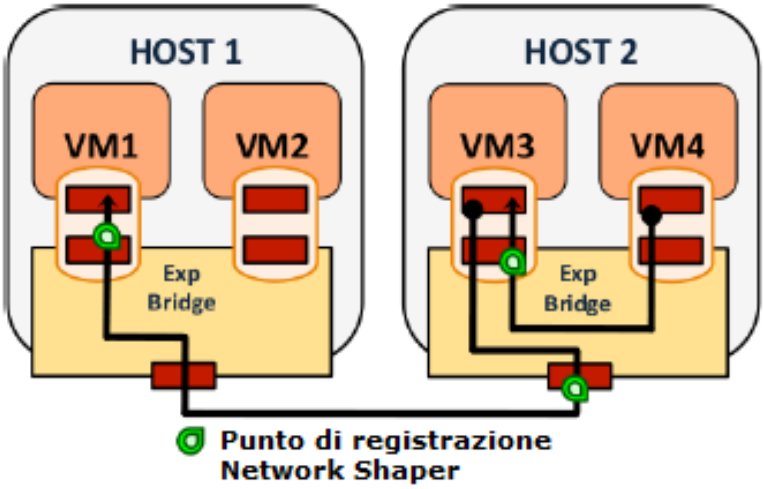

Figura 1.4: Flusso di pacchetti tra VM sullo stesso host o su host differenti *(tratto da [12])*

Le interfacce virtuali e l'*experimental interface* fisica (in rosso in Figura 1.4) sono connesse mediante l'*experimental bridge*. Ogni interfaccia di rete, virtuale o fisica, ha il proprio sistema di *QDisc* collegato ad entrambi i buffer in entrata ed in uscita. Il *Network Shaper* è registrato sul buffer in uscita per le seguenti ragioni:

- il filtraggio sui pacchetti in uscita minimizza il quantitativo di traffico superfluo attraversante la rete;
- la percezione delle macchine virtuali della rete è coerente e consistente con la connettività sperimentata;
- il filtraggio di pacchetti broadcast può essere effettuato a destinazione.

Elaborazione dei pacchetti. Le *QDisc* invocano il *Network Shaper* in due differenti casi: quando un pacchetto è generato dalla network stack e quando il driver del dispositivo è pronto ad inviare un pacchetto. Questi due casi vengono gestiti dal *Network Shaper* tramite due differenti funzioni: `enqueue` e `dequeue`. La funzione `enqueue` riceve un pacchetto dalla *Network Stack* e, secondo le regole di connettività, decide se deve essere consegnato immediatamente oppure per quanto tempo dovrà essere ritardato. Se ci troviamo nel secondo caso allora verrà attribuito un *time_to_send* basato sul ritardo assegnato. Il pacchetto sarà allora immagazzinato in una coda a priorità, interna al *Network Shaper*, dove la priorità è data dal *time_to_send*. Quando viene invocata la funzione `dequeue`, il *Network Shaper* determina se c'è un pacchetto in coda il cui *time_to_send* è scaduto; in tal caso il pacchetto verrà passato al sistema *QDisc* per essere inviato al driver. Questa semplice soluzione permette l'applicazione di accurati ritardi senza l'utilizzo di timer. In Figura 1.4 vediamo i differenti punti in cui il *Network Shaper* viene registrato nella rete; seguendo il flusso dei pacchetti si osserva che, in alcuni casi, lo stesso pacchetto può attraversare il *Network Shaper* più di una volta. In queste situazioni la funzione `enqueue` dovrà determinare in quale punto della rete viene elaborato. Mediante la suddivisione in pacchetti broadcast ed unicast, è possibile identificare quattro classi di pacchetti che verranno processate in maniera differente.

	Esterno	**Interno**
Broadcast	Nessun ritardo applicato. Ulteriore elaborazione alla destinazione.	Ritardato o consegnato secondo le regole di connettività.
Unicast	Ritardato o consegnato secondo le regole di connettività.	Controllo delle regole di connettività sia alla sorgente che alla destinazione.

Tabella 1.1: Classificazione pacchetti Network Shaper

I casi di doppio controllo consentono al *Network Shaper* di scartare dalle interfacce delle macchine virtuali il traffico che non è generato dalla rete sperimentale. Il rimodellamento della rete richiede che ogni singolo pacchetto venga processato. Per evitare che il *Network Shaper* diventi il collo di bottiglia della rete, tale elaborazione deve essere quanto più rapida possibile; pertanto il numero di operazioni di recupero e applicazione delle regole di connettività deve essere minimizzato.

Recupero delle regole. Le regole di una determinata macchina virtuale e della sua interfaccia sono recuperate utilizzando un indice. La selezione dell'indice implica sempre un compromesso tra l'utilizzo della memoria e il tempo di accesso; in questo caso la scelta migliore è una tabella hash. Al fine di evitare sprechi di memoria dovuti all'utilizzo di generiche tabelle hash, MoViT immagazzina le regole utilizzando *consistent hashing* [16]. Dal momento che MoViT ha il controllo delle interfacce ethernet delle macchine virtuali, utilizza gli ultimi 4 byte dell'indirizzo MAC delle VM come indice per la tabella delle regole di connettività.

Applicazione delle regole. Per ridurre il tempo di applicazione di una regola, MoViT utilizza una tabulazione dei tassi di rilascio e dei ritardi da

applicare: tasso di rilascio, ritardo medio e varianza del ritardo sono registrati in una tabella bidimensionale in cui l'indice di riga rappresenta lo stato di connettività (ad esempio: la perdita di percorso) e l'indice di colonna rappresenta la lunghezza del pacchetto. Una *regola di connettività*, tra una sorgente ed una destinazione, è quindi composta da un set di tre indici di riga rappresentanti il relativo tasso di rilascio, ritardo medio e varianza del ritardo. Delegare la definizione del tasso, del ritardo medio e della varianza al *Channel Module* rende il *Network Shaper* totalmente indipendente dal modello di canale utilizzato per riprodurre la connettività. L'applicazione di una regola di connettività coinvolge soltanto una serie di ricerche in memoria che la rendono un processo molto veloce ed efficiente. Per mantenere le politiche di filtraggio del *Network Shaper* indipendenti dal modello di canale che si vuole riprodurre, le tabelle sono un input per il *Network Shaper* insieme alle regole di connettività. Tenendo conto del differente comportamento dei pacchetti broadcast ed unicast, il *Network Shaper* prende come input un doppio set di tabelle.

Switch delle regole. Ad ogni snapshot di mobilità è assegnato un tempo di applicazione che definisce quando un set di regole di connettività deve essere trasferito simultaneamente a tutti i *Network Shaper*. Questa sincronizzazione (nell'ordine dei millisecondi) è ottenuta con l'ausilio del *Network Time Protocol* [NTP], protocollo utilizzato per sincronizzare gli orologi all'interno di una rete a commutazione di pacchetto. Grazie ad esso il trasferimento delle regole di connettività a tutti gli host avviene **approssimativamente** allo stesso tempo. Lo switch tra due set di regole successivi deve essere un processo estremamente efficiente e quanto più possibile indipendente da fattori esterni (ad esempio: il carico della rete o dell'host). Il tempo di transizione da un set di regole ad un altro è trascurabile per quel che concerne i ritardi di rete (nell'ordine dei microsecondi). Per queste ragioni il processo di switch tra due set di regole è gestito da una specifica applicazione, il **Rule Switcher** (eseguito in user space), che ottiene le regole di connettività dal

Channel Module e le trasferisce al *Network Shaper*. Questa soluzione consente al *Rule Switcher* l'indipendenza dal *Channel Module*, permettendo così l'utilizzo di differenti modelli di propagazione. Il trasferimento delle regole da user space a kernel space avviene utilizzando un metodo standard di comunicazione ovvero le *Netlink socket*.

Al crescere della rete emulata si avrà un'ovvia crescita delle dimensioni dei set di regole. Per garantire scalabilità e ridurre il carico sulla pila protocollare,vengono trasferiti al *Network Shaper* soltanto la locazione di memoria e la dimensione dei set di regole.

1.4 MoViT per le VANET

Finora MoViT è stato descritto come un sistema di emulazione generale. La comunicazione tra tutti i moduli software precedentemente descritti segue un protocollo ben definito, consentendo la sostituzione di ciascuno di essi nel caso lo si ritenga opportuno. L'architettura di MoViT può essere utilizzata per emulare qualsiasi tipo di rete, ma il suo studio ha riguardato prevalentemente le reti mobili veicolari. La parte centrale di questo studio è contenuta all'interno dell'*experiment controller* ed è il modulo software che si occupa della generazione di mobilità.

1.4.1 Generatore di mobilità

Vista l'importanza che è stata dedicata in questa tesi a questo particolare modulo si è scelto di descrivere separatamente il software responsabile per la generazione della mobilità. La mobilità è il principale responsabile dei cambiamenti di topologia in una VANET e gran parte delle topologie dinamiche sono peculiari delle reti veicolari. MoViT offre la possibilità di riprodurre mobilità partendo da tracce veicolari pre-registrate, oppure di simularla in tempo reale secondo requisiti end-to-end predefiniti. Il generatore di mobilità utilizzato in MoViT è il simulatore SUMO [SUMO]. SUMO permette la simulazione di migliaia di veicoli in tempo reale, inoltre offre la possibi-

lità di controllare dinamicamente l'ambiente simulato tramite TraCI (Traffic Control Interface) [17]. TraCI consente il controllo dei veicoli simulati dal momento in cui la simulazione viene avviata, permettendo la valutazione di una grande varietà di applicazioni inerenti la sicurezza veicolare oppure sistemi di trasporto intelligenti (ITS). Entrambi gli strumenti verranno dettagliatamente descritti nel prossimo capitolo.

1.4.2 Modellazione della connettività

Il *Network Shaper* applica tassi di rilascio e ritardi addizionali ai pacchetti secondo i set di regole di connettività. Tali regole sono costituite dagli indici di riga delle tabelle contenenti i valori effettivi del tasso di rilascio e del ritardo da applicare. Per riprodurre il comportamento di una VANET sono stati utilizzati dei modelli rappresentanti le caratteristiche tipiche di una rete veicolare.

Packet Error Rate. Il *Packet Error Rate (PER)* rappresenta il numero di pacchetti ricevuti in maniera non corretta rispetto al numero totale di pacchetti ricevuti. Un pacchetto viene dichiarato incorretto se almeno un bit è incorretto. In ambienti wireless il PER è espresso in funzione del *Bit Error Rate (BER)* e della lunghezza del pacchetto; può essere espresso tramite la seguente espressione:

$$PER = 1 - (1 - BER)^{N_b} \tag{1.1}$$

dove N_b è la lunghezza del pacchetto in bit. Il *BER* per reti IEEE 802.11 può essere calcolato nel modo seguente [18]:

$$BER = Q(\sqrt{11 - SNR}) \tag{1.2}$$

dove *SNR (Signal to Noise Ratio)* rappresenta il rapporto segnale/rumore, che esprime quanto il segnale sia potente rispetto al rumore nell'ambiente esaminato. Nel caso di MoViT, *SNR* viene calcolato considerando l'attenuazione del segnale valutata tramite il modello di propagazione *CORNER*

[19]. *CORNER* è un modello di propagazione realistico per scenari urbani che prende in considerazione la presenza di ostacoli per la propagazione del segnale.

MoViT utilizza il *PER* dell'equazione (1.1) nel caso di pacchetti broadcast, per quanto riguarda i pacchetti unicast viene calcolata una tabella separatamente. Infatti i pacchetti unicast in reti IEEE 802.11 utilizzano meccanismi di ACK (*Acknowledge, risposta alla ricezione*) e conseguenti ritrasmissioni in caso di fallimento; pertanto il tasso di rilascio è computato in funzione del numero massimo di ritrasmissioni consentite. Il *PER* per pacchetti unicast può essere approssimato con la seguente espressione:

$$PER_{unicast} = 1 - \sum_{i=0}^{r_{max}-1} (PER)^i (1 - PER) \tag{1.3}$$

dove r_{max} è il numero massimo di ritrasmissioni specificato dallo standard IEEE 802.11.

Ritardo dei pacchetti. Nelle reti wireless il ritardo sperimentato da ogni pacchetto è la somma di due componenti, il ritardo di trasmissione d_t e il ritardo del canale d_c:

$$d = d_t + d_c \tag{1.4}$$

Il ritardo di trasmissione d_t può essere calcolato in modo deterministico secondo la velocità di trasmissione. Il ritardo del canale d_c può essere determinato soltanto in maniera statistica. Si approssima d_c ad una distribuzione normale:

$$d_c = N\left(\overline{d_c}, \frac{\overline{d_c}}{2}\right) \tag{1.5}$$

dove $\overline{d_c}$ è il valore medio del ritardo del canale. Come affermato precedentemente, nello standard IEEE 802.11 i pacchetti unicast possono essere ritrasmessi mentre i pacchetti broadcast non possono essere ritrasmessi. Per questo motivo d_c ha espressioni diverse dipendentemente dalla tipologia di pacchetto:

$$\overline{d_c} = \overline{d_a} \ , \qquad \text{broadcast}$$
$$\overline{d_c} = \overline{d_a} + \overline{d_r} \ , \qquad \text{unicast}$$

dove d_a è il ritardo di accesso mentre d_r è il ritardo di ritrasmissione. d_a rappresenta il ritardo sperimentato da un nodo per eseguire un accesso casuale al canale wireless. Per approssimare questo parametro viene utilizzato il modello di Bianchi [20]:

$$\overline{d_a} = \sigma \frac{1 - P_{TX}}{P_S P_{TX}} + T_C \left(\frac{1}{P_S} - 1 \right) \tag{1.6}$$

$$P_{TX} = 1 - (1 - \tau)^n \tag{1.7}$$

$$P_S = \frac{n\tau(1 - \tau)^{n-1}}{1 - (1 - \tau)^n} \tag{1.8}$$

$$T_C = \frac{packet\ length}{\gamma} \tag{1.9}$$

dove τ è un parametro rappresentante la porzione di tempo in cui il canale è occupato, σ è definito dallo standard e n è il numero di nodi che si contendono il canale al tempo di trasmissione, tale valore viene approssimato da MoViT al numero di nodi vicini utilizzando il modello CORNER. Rispettando il modello di Bianchi abbiamo che d_a è espresso in funzione di τ, dal numero di nodi vicini e dalla lunghezza del pacchetto.

Il ritardo medio di ritrasmissione per pacchetti unicast è in funzione del *PER* e conseguentemente del *SNR*. $\overline{d_r}$ può essere così rappresentato:

$$\overline{d_r} = \sum_{r=1}^{r_{max}} (PER)^r d_t \tag{1.10}$$

1.5 Progetti simili

Negli ultimi anni sono stati sviluppati diversi ibridi tra strumenti di simulazione ed emulatori wireless che, come MoViT, basandosi su hardware

reale, sono in grado di aumentare il realismo rispetto alla pura simulazione. Tra questi possiamo citare *TWINE* [21] e la sua evoluzione *WHYNET* [22] che integrano simulazione, emulazione ed hardware reale. Tuttavia, nonostante l'integrazione tra i vari approcci, non riesce a superare i limiti dei testbed reali in quanto le sue performance risentono fortemente dell'aumento del numero di nodi. Un altro esempio è *QOMET* [23], un emulatore wireless multiuso in grado di fornire un ambiente flessibile e altamente scalabile. Purtroppo QOMET permette una limitata modellazione della mobilità e manca di modellazione dinamica a livello di collegamento, pertanto l'utente deve definire staticamente limitazioni di banda, ritardi e perdita di pacchetti, a differenza di MoViT dove la modellazione del canale viene dinamicamente calcolata con l'esecuzione dell'esperimento. Un ulteriore esempio è *Emulab* [24] che riproduce la mobilità della rete controllando fisicamente il movimento di robot reali in un ambiente controllato. Questo approccio è maggiormente realistico, in quanto le comunicazioni avvengono in un reale canale wireless, ma gli scenari sono estremamente limitati ed è inoltre vincolato all'utilizzo di una certa tipologia di hardware. In *NRL Mobile Network Emulator (NME)* [25] per ogni nodo è richiesto l'utilizzo di un PC reale e l'utilizzo di sistemi operativi linux. La connettività è emulata utilizzando *iptables*[1] e la mobilità viene emulata tramite GPS. Il sistema emula la connettività di rete mediante cambiamenti delle regole di iptables secondo i modelli di propagazione del segnale e mobilità. La sostanziale differenza con MoViT è l'utilizzo di PC reali, rispetto alle macchine virtuali, utilizzati per emulare i nodi della rete, inoltre MoViT è indipendente dal sistema operativo utilizzato. La scelta di utilizzare dei PC comporta anche degli evidenti limiti per ciò che concerne la realizzazione di esperimenti in larga scala.

[1] ***iptables*** è un software utilizzato nei sistemi di tipo Unix che permette di definire le regole per il filtraggio o il reindirizzamento di pacchetti in una rete.

Capitolo 2

SUMO

In questo capitolo verrà illustrato uno dei più importanti e diffusi software di simulazione di mobilità veicolare: SUMO. Questo simulatore è stato studiato sotto molteplici aspetti con il fine ultimo di ottimizzare le prestazioni dell'ambiente simulato, evitando così possibili "colli di bottiglia" nella generazione dei dati di mobilità utilizzati e distribuiti da MoViT.

La prima versione di SUMO, totalmente open source, viene rilasciata nel 2002 da parte del Centro Aerospaziale Tedesco (DLR). La scelta di realizzare un software open source ha permesso una costante evoluzione del software che ora non si limita ad essere soltanto un simulatore di traffico ma anche un insieme di applicativi in grado di preparare e gestire la simulazione [26].

2.1 Concetti di base

SUMO viene concepito per simulare una rete stradale delle dimensioni di una grande città. Tale simulazione è "multi-modale", il che significa che non vengono modellate le sole autovetture ma anche altre categorie di veicoli quali, ad esempio, i sistemi di trasporto pubblico, le reti ferroviarie, motocicli e via dicendo. L'elemento atomico della simulazione è un singolo essere umano. Il movimento di un individuo è descritto da un istante di partenza e

da un percorso che esso andrà a compiere; questo percorso è a sua volta composto da sotto-percorsi che descrivono le singole modalità di spostamento. Un individuo può, ad esempio, utilizzare la propria automobile per raggiungere la più vicina stazione di trasporto pubblico e continuare il suo viaggio utilizzando altri mezzi. Oltre agli spostamenti tramite veicoli motorizzati una persona ha la facoltà di spostarsi camminando; questa modalità non è direttamente simulata ma viene modellata stimando il tempo necessario per raggiungere, a piedi, una determinata destinazione.

I flussi di traffico sono simulati in maniera microscopica, ovvero ciascun veicolo all'interno della rete è modellato individualmente ed ha una propria posizione ed una propria velocità istantanea. Ad ogni step di simulazione, che ha la durata di un secondo, questi valori sono aggiornati in funzione del veicolo precedente rispetto al senso di marcia e della tipologia di strada in cui il veicolo si sta muovendo. La simulazione dei veicoli è a tempo discreto e spazio continuo. Durante la simulazione gli attributi che caratterizzano una determinata strada, quali il senso di marcia e il limite di velocità, vengono rispettati dai veicoli che la percorrono.

I conducenti e le loro modalità di guida sono simulati utilizzando un'estensione del modello di Gipps [27]. Questo modello è in grado di descrivere accuratamente le caratteristiche principali della mobilità, sia nel caso di flussi congestionati dal traffico, sia nel caso di strade non trafficate. Ad ogni step della simulazione la velocità di un veicolo viene adattata secondo la velocità del veicolo che lo precede, ciò produce un sistema in grado di evitare collisioni nel successivo step. Questa velocità viene chiamata velocità di sicurezza (v_{safe}) e viene calcolata mediante la seguente espressione:

$$v_{safe}(t) = v_l(t) + \frac{g(t) - v_1(t)\tau}{\frac{\overline{v}}{b(\overline{v})} + \tau} \tag{2.1}$$

dove $v_l(t)$ è la velocità del veicolo precedente rispetto al senso di marcia al tempo t, $g(t)$ (gap) è il divario da tale veicolo al tempo t, τ è il tempo di reazione del conducente (solitamente 1 secondo) e b è la funzione di decelerazione. Per associare l'accelerazione (a) alle capacità fisiche del veicolo, la

risultante velocità "desiderata" (v_{des}) è calcolata come il minimo tra:

$$v_{des}(t) = min[v_{safe}, v(t) + a, v_{max}] \tag{2.2}$$

Ovvero tra la velocità di sicurezza (v_{safe}), la velocità del veicolo sommata all'accelerazione massima ($v(t) + a$) e la velocità massima raggiungibile dal veicolo (v_{max}). In questo modo un veicolo non si muoverà o accelererà più di quanto gli sia fisicamente possibile. Il conducente viene simulato assumendo la possibilità che possa commettere errori e che non sia quindi in grado di raggiungere perfettamente la velocità desiderata; questa possibilità viene modellata sottraendo "errore umano" scelto casualmente (ϵ) dalla velocità desiderata:

$$v_t = max[0, rand[v_{des}(t) - \epsilon a, v_{des}(t)]] \tag{2.3}$$

Siccome durante la simulazione un veicolo non può spostarsi in retromarcia, il massimo tra la velocità calcolata e zero sarà la velocità finale del veicolo in una determinato istante t.

Per eseguire una simulazione in SUMO è necessario innanzitutto definire lo scenario nel quale i veicoli potranno circolare ovvero la rete stradale comprensiva di corsie, semafori, incroci e altre strutture utili alla rappresentazione della realtà; successivamente bisogna creare la "domanda di mobilità" che si intende simulare ovvero le tipologie di veicoli, i possibili percorsi e l'elenco dei veicoli che prenderanno parte alla simulazione. Verranno ora descritte le fasi e la definizione dei file di input, schematizzati in Figura 2.1, necessari ad avviare una simulazione.

2.2 Creazione di reti stradali

Un file di rete SUMO (formato `.net.xml`) descrive la parte della mappa interessata dal traffico. Questi file definiscono quindi la rete stradale e tutte le informazioni utili a indirizzare e gestire i flussi di traffico (incroci, corsie, semafori ecc.). I file di rete, essendo in formato XML (*eXtensible*

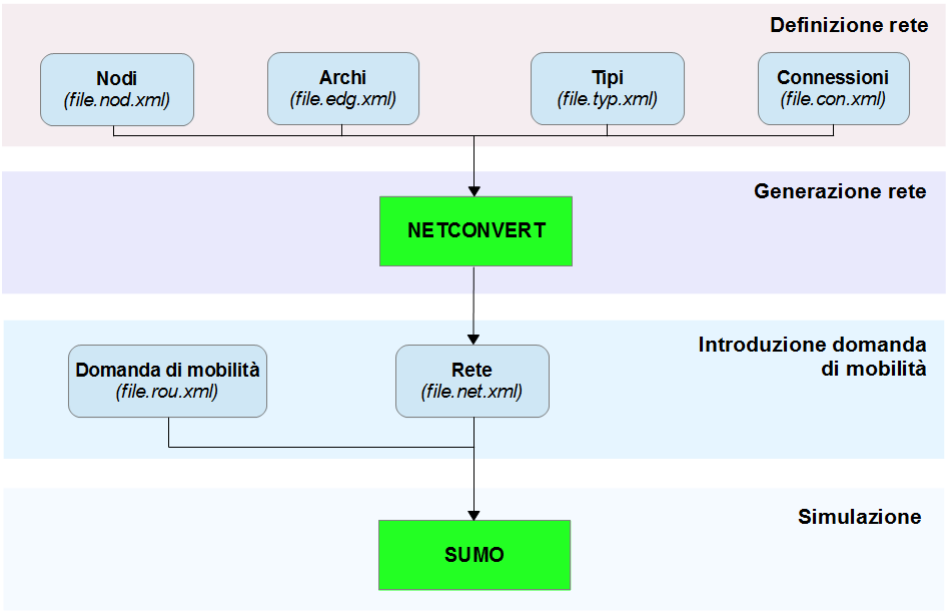

Figura 2.1: Input per una simulazione in SUMO

Markup Language), sono leggibili direttamente dall'utente, tuttavia, vista la loro complessità al crescere della rete rappresentata, non sono pensati per essere modificati manualmente; per questo motivo SUMO fornisce un insieme di applicativi in grado di creare scenari basati su descrizioni XML. Tali descrizioni possono essere definite, tramite editor di testo, da parte dell'utente oppure, preferibilmente, importate da altre fonti grazie all'ausilio di applicativi, quali ad esempio NETCONVERT, che consentono la conversione di formati diversi in file di rete SUMO.

2.2.1 Definizione della rete

Generalizzando è possibile definire una rete stradale come un grafo orientato dove i **nodi** sono rappresentati da incroci e intersezioni mentre gli **archi** sono le strade ed i vari collegamenti tra nodi. In aggiunta a questa struttura di base bisogna considerare anche la presenza di altre informazioni e strutture in grado di completare la rappresentazione della rete:

- ogni strada (arco) si compone di una o più *corsie*;
- ogni *corsia* ha una propria posizione, una propria dimensione ed un proprio limite di velocità;
- la presenza di vincoli relativi ai *sensi di marcia*;
- le *connessioni* tra corsie nei vari incroci (nodi);
- la posizione e la logica dei *semafori* presenti.

2.2.1.1 Nodi

I nodi componenti la rete vengono descritti tramite file con estensione `.nod.xml`. Ogni nodo viene definito in una singola linea XML con il seguente formato:

```
<node id="<STRING>" x="<FLOAT>" y="<FLOAT> ... />
```

Nella Tabella 2.1 viene presentato un elenco degli attributi associabili ad ogni nodo.

Attributo	Tipo di Dato	Descrizione
`id`	String	L'identificativo univoco del nodo.
`x`	Float	La posizione del nodo, in metri,rispetto all'asse delle ascisse.
`y`	Float	La posizione del nodo, in metri, rispetto all'asse delle ordinate.
`type`	Enum	La tipologia del nodo.

Tabella 2.1: Descrizione dei nodi

Nel caso in cui non venga dichiarato il parametro `type`, le applicazioni (in particolare NETCONVERT) sceglieranno autonomamente una delle tipologie di nodo previste da SUMO, ovvero:

- `priority`: i veicoli provenienti da un arco devono attendere che altri veicoli, provenienti da archi con priorità maggiore, abbiano attraversato il nodo.
- `traffic_light`: il nodo è controllato da una semaforo.
- `right_before_left`: realizza meccanismi di precedenza ovvero i veicoli lasceranno transitare altri veicoli provenienti dalla loro destra.
- `unregulated`: il nodo non è controllato in alcun modo e tutti i veicoli possono percorrerlo senza frenare; la presenza di questi nodi ha un'alta probabilità di generare incidenti.

L'utente non può ampliare questa classificazione con nuovi tipi di nodo. Ogni file `.nod.xml` deve contenere le definizioni dei vari nodi all'interno dei tag `<nodes>` e `</nodes>`.

2.2.1.2 Archi

I file che, invece, si occupano di descrivere gli archi presenti nella rete (ovvero le strade e le varie vie di comunicazione) utilizzano il formato `.edg.xml`. Un singolo arco viene definito mediante la seguente sintassi:

```
<edge id="<STRING>" from="<NODE_ID>" to="<NODE_ID> ... "/>
```

Possono essere inoltre aggiunti vari attributi in grado di descrivere più o meno dettagliamente la tipologia di strada che si intende realizzare. In Tabella 2.2 tutti gli attributi assegnabili ad un arco.

Tabella 2.2: Descrizione degli archi

Attributo	Tipo di Dato	Descrizione
`id`	String	L'identificativo univoco dell'arco.
`from`	String (id nodo)	L'identificativo di un nodo precedentemente definito rappresentante l'inizio dell'arco.

Continua nella prossima pagina

Tabella 2.2: continua dalla pagina precedente

Attributo	Tipo di Dato	Descrizione
`to`	String (id nodo)	L'identificativo di un nodo precedentemente definito rappresentante la fine dell'arco.
`type`	String (id type)	Il nome di una delle tipologie di arco definite.
`numLanes`	Int	Il numero di corsie dell'arco.
`speed`	Float	La velocità massima (in m/s) permessa nell'arco.
`priority`	Int	La priorità dell'arco.
`length`	Float	La lunghezza dell'arco in metri.
`shape`	Lista di posizioni (x,y)	Descrive le posizioni attraversate dall'arco tra il nodo di partenza e quello di arrivo.
`spreadType`	Enum ("right", "center")	Descrive in che modo verranno posizionate le corsie dichiarate in numLanes.
`allow`	Lista classi di veicoli	Definisce quali classi di veicoli possono circolare in un determinato arco.
`disallow`	Lista classi di veicoli	Definisce quali classi di veicoli non possono circolare in un determinato arco.

Ogni file `.edg.xml` deve contenere le definizioni dei vari archi all'interno dei tag `<edges>` e `</edges>`.

Tipi. In SUMO è inoltre possibile introdurre nuove tipologie di arco all'interno di file `.typ.xml`. Ad esempio, con le seguenti linee di testo possiamo definire la categoria "autostrada", caratterizzata da un nome, una priorità,

un numero di corsie ed un limite di velocità espresso in metri al secondo (per quanto riguarda la normativa italiana tale limite è di 33.1 m/s ovvero 130 km/h):

```
<types>
 <type id="autostrada" priority="15" numLanes="3" speed="33.1" ... />
</types>
```

Tramite l'attributo `type` di un arco sarà quindi possibile specificare che l'arco in questione è, ad esempio, un'autostrada. Infine, SUMO consente di dettagliare le caratteristiche delle singole corsie rendendo possibile descriverne la velocità, la lunghezza e a quali classi di veicoli è consentito (ad esempio mezzi di soccorso o trasporto pubblico) o non è consentito l'accesso.

2.2.1.3 Connessioni

Una volta definiti gli archi, ed opzionalmente le varie tipologie, si rende necessario specificare come il traffico potrà muoversi in essi. Questa specifica viene realizzata tramite quelle che vengono chiamate "connessioni". Con questo meccanismo sarà dunque possibile definire quali spostamenti potranno essere compiuti dai flussi di traffico e in che modo le varie corsie saranno collegate tra loro. Nel caso in cui non vengano specificate connessioni NETCONVERT definisce autonomamente le informazioni mancanti utilizzando delle euristiche; ciò potrebbe comportare la generazione di situazioni poco realistiche come, ad esempio, incroci in cui sia possibile effettuare inversioni di marcia[1]. Per definire le connessioni si utilizza la seguente sintassi XML:

```
<connections>
   <connection from="<EDGE_ID>" to="<EDGE_ID>" fromLane="<INT>"
     toLane="<INT>" />
   ...
</connections>
```

[1]In questo caso si fa riferimento al contesto stradale italiano. In alcune legislazioni degli Stati Uniti, ad esempio, è possibile effettuare inversioni di marcia in corrispondenza di incroci o persino svoltare a destra qualora il semaforo sia rosso (rispettando comunque la precedenza dei veicoli sopraggiungenti).

Avendo definito tutte le strutture ed i file di input è possibile generare la rete vera e propria utilizzando l'applicazione NETCONVERT.

2.2.2 Generazione della rete

Seguendo lo schema riportato in Figura 2.1, una volta definiti i file `.nod.xml`, `.edg.xml`, `.typ.xml` e `.con.xml`, si può procedere alla creazione del file di rete (`.net.xml`) mediante il modulo NETCONVERT. Lo scopo di questa potente applicazione è quello di importare reti stradali da fonti differenti e generare reti direttamente utilizzabili dagli applicativi SUMO. Un elenco dei formati e degli applicativi più importanti con cui può lavorare NETCONVERT è il seguente:

- Descrizioni native SUMO (`.edg.xml`,`.nod.xml`,`.con.xml`,`.net.xml`);
- OpenStreetMap [OSM] (`.osm.xml`);
- VISUM [VISUM] (`.net`);
- Vissim [VISSIM];
- openDRIVE [OPENDRIVE] (`.xodr`);
- MATsim [MATSIM] (`.xml`);
- ArcView [ARCVIEW].

Essendo NETCONVERT un'applicazione a linea di comando risulta conveniente definire un file di configurazione (`.netc.cfg`) contenente i riferimenti ai descrittori della rete ed il riferimento al file output, in formato `.net.xml`, che si intende creare. In alternativa è necessario specificare tutti i parametri di input ed output (nodi, archi, connessioni ecc.) tramite riga di comando. Vista la complessità degli scenari realizzati durante questo lavoro di tesi, NETCONVERT è stato prevalentemente utilizzato con mappe provenienti dai database di OpenStreetMap [OSM]: la definizione delle medesime reti stradali, tramite le modalità precedentemente descritte, non sarebbe stata altrimenti realizzabile. Per poter convertire tali mappe è stato utilizzato il seguente comando:

```
netconvert --osm-files rete.osm.xml -o rete.net.xml
```

Alcuni esempi di scenari generati tramite l'utilizzo di NETCONVERT verranno esposti al capitolo 3. La conversione è personalizzabile sotto molteplici aspetti secondo le esigenze dell'utente; per un completo elenco delle funzionalità che vengono messe a disposizione da NETCONVERT si rimanda al manuale dell'applicazione in [SUMO].

Generazione casuale di reti

SUMO offre la possibilità di creare reti astratte direttamente utilizzabili dagli altri moduli del simulatore. L'applicazione NETGEN, basandosi su parametri definiti dall'utente, permette di realizzare tre differenti tipologie di rete: a griglia, a ragnatela e casuali. Nel primo caso, specificando la tipologia di rete, il numero di incroci, la distanza tra gli incroci ed il file di output, NETGEN realizzerà una maglia di strade perpendicolari tra loro. Nel caso di reti a ragnatela viene prodotto un insieme di perimetri stradali concentrici tra loro, tagliati da uno specificato numero di archi. Le reti casuali, invece, vengono generate tramite l'uso di parametri quali la scelta casuale della lunghezza, dell'angolo e delle distanze degli archi limitrofi.

2.3 Domanda di mobilità

Una volta generata una rete si può utilizzare l'interfaccia grafica SUMO-GUI per visualizzarla; si avrà così modo di esplorare lo scenario realizzato, contenente tutte le vie di comunicazione, la segnaletica stradale e le strutture precedentemente definite. Per completare l'input necessario ad avviare una simulazione bisogna generare ed introdurre nella rete stradale i veicoli che andranno a formare i flussi di traffico. Questa viene detta *"domanda di mobilità"* (*traffic demand*) e viene definita tramite file in formato `.rou.xml` che, al loro interno, specificano:

- le **tipologie** di *veicoli*;

- i **percorsi** percorribili dai *veicoli*;
- i **veicoli** che prenderanno parte alla simulazione.

2.3.1 Tipologie di veicoli

All'interno dei file `.rou.xml` è necessario innanzitutto definire le categorie di veicoli partecipanti alla simulazione. In questo modo sarà possibile classificare i singoli veicoli in modo che rispecchino le macro-caratteristiche della tipologia di appartenenza. Gli attributi descrivono i veicoli in base alle loro caratteristiche fisiche come, ad esempio, accelerazione, velocità e lunghezza. Gli attributi assegnabili ad una determinata tipologia sono descritti nella Tabella 2.3.

Tabella 2.3: Descrizione delle tipologie di veicoli

Attributo	Tipo di Dato	Descrizione
`id`	String	Il nome del tipo di veicolo.
`accel`	Float	La capacità di accelerazione (in m/s^2) dei veicoli di questo tipo.
`decel`	Float	La capacità di decelerazione (in m/s^2) dei veicoli di questo tipo.
`sigma`	Float	Parametro corrispondente alle imperfezioni del conducente (compreso tra 0 e 1).
`length`	Float	La lunghezza del veicolo in metri (default: 5m).
`minGap`	Float	Lo distanza dal veicolo precedente in metri (default: 2.5 m)
`maxSpeed`	Float	La velocità massima raggiungibile dal veicolo (default: 70 m/s).

Continua nella prossima pagina

Tabella 2.3: continua dalla pagina precedente

Attributo	Tipo di Dato	Descrizione
`color`	Colore RGB	Il colore della tipologia di veicoli (default: 1,1,0 - giallo) .
`vClass`	Class (Enum)	Una classe astratta di veicoli (default: "Unknown").
`emissionClass`	Class (Enum)	Descrive l'emissioni in base a classi astratte (default: "Unknown").
`guiShape`	Shape (Enum)	Come il veicolo verrà visualizzato graficamente (default: "Unknown").
`guiWidth`	Float	La larghezza del veicolo in metri (default: 2 m).

La dichiarazione XML di una tipologia di veicoli avviene secondo il seguente formato:

```
<vType id="<STRING>" accel="<FLOAT>" maxspeed="<FLOAT>" ... />
```

Tramite l'attributo `vClass` è dunque possibile assegnare ad una categoria di veicoli, una classe astratta predefinita da SUMO. Queste classi vengono anche utilizzate dagli attributi `allow` e `disallow`, durante la definizione di strade e corsie; il loro scopo è quello di consentire o vietare l'accesso, ad una determinata strada o corsia, ad una o più classi astratte di veicoli. Tra le principali classi astratte previste da SUMO si possono elencare:

- private;
- public_transport;
- public_emergency
- taxi;
- bus;

- motorcycle;
- bicycle;
- pedestrian.

Tali classi vengono definite "astratte" in quanto specificano solamente una classificazione nominale e non implicano alcun vincolo sulla creazione del veicolo o sulla definizione di una tipologia di veicoli; ad esempio, si può tranquillamente generare un veicolo appartenente alla classe astratta "bus" della lunghezza di 0.5 m.

L'attributo `emissionClass`, invece, definisce la classe relativa alle emissioni inquinanti a cui appartiene la tipologia di veicolo che si intende creare, secondo le categorie ed i parametri specificati dall'HBEFA (*HandBook of Emission FActor*). Tramite l'utilizzo di questo attributo è possibile fare valutazioni riguardanti i principali agenti inquinanti (*C02*, *C0*, *HC*, *NOx*, *PMx*) ed i consumi di carburante dei veicoli partecipanti alla simulazione.

2.3.2 Percorsi

Definiamo due tipologie di percorso: **trip** e **route**. Con *trip* si intende il movimento di un veicolo da una posizione ad un'altra dello scenario, tale spostamento è definito da: un identificativo, un arco (strada) di partenza, un arco destinazione ed un tempo di partenza. Con *route*, invece, si definisce un "trip esteso" ovvero non vengono specificati solamente gli archi di partenza ed arrivo ma tutti gli archi che dovranno essere attraversati dal veicolo. Il file `.rou.xml` deve contenere al suo interno l'elenco delle possibili *route* effettuabili dai veicoli. La dichiarazione di una *route* avviene secondo il seguente formato:

```
<route id="<STRING>" edges="<EDGE_ID_LIST>" ... />
```

È inoltre possibile assegnare ad una determinata *route* un certo colore tramite l'attributo `color`. Per generare una simulazione realistica, la lista di archi componenti una *route* deve presentare archi sequenzialmente connessi; nel caso ciò non avvenga un veicolo percorrente tale tragitto verrà teletrasportato da un arco all'altro della mappa nel momento in cui l'arco successivo

non sia connesso. Oltre alla definizione manuale XML, SUMO fornisce gli strumenti per generare autonomamente le *route* secondo input specificati dall'utente. Tra i più importanti, e tra i più utilizzati durante questa tesi, vi è DUAROUTER il cui scopo è calcolare percorsi "ottimi", ovvero i percorsi più brevi congiungenti archi sorgente ed archi destinazione. Oltre alla rete `.net.xml`, DUAROUTER richiede in input un file `.xml` che definisca i *trip* su cui calcolare i percorsi ottimi; si ricorda che con *trip* si intende la specifica di un identificativo, un arco sorgente, un arco destinazione ed un tempo di partenza. Il formato con cui è possibile definire i possibili *trip* è il seguente:

```
<trips>
 <tripdef id="<STRING>" depart="<INT>" from="<EDGE_ID>" to="<EDGE_ID>" />
 ...
</trips>
```

Una volta specificati i *trip* è possibile utilizzare DUAROUTER che, mediante la seguente chiamata a riga di comando:

```
duarouter -t trips.xml -n rete.net.xml -o routes.rou.xml
```

produrrà il file `routes.rou.xml` contenente una definizione "minimale" della domanda di mobilità e dei veicoli (id veicolo, tempo di partenza e percorso effettuato da ogni veicolo). I percorsi ottimi forniti da DUAROUTER vengono computati mediante l'algoritmo di Dijkstra descritto in [28].

Considerate le dimensioni degli scenari utilizzati in questa tesi, la definizione dei trip è stata effettuata mediante uno script Python (`randomTrips.py`) fornito da SUMO. Specificando la rete, ed altri parametri quali, ad esempio, l'intervallo di tempo in cui i veicoli prenderanno parte alla simulazione e la frequenza di partenza dei veicoli, `randomTrips.py` genera *trip* scegliendo in maniera uniformemente casuale archi sorgente ed archi destinazione e produce un file, in formato `.xml`, direttamente utilizzabile da DUAROUTER.

2.3.3 Definizione veicoli

Avendo dichiarato le tipologie di veicoli ed i percorsi che essi possono compiere, è possibile completare la generazione della domanda di mobilità

definendo nello specifico i singoli veicoli. Come anticipato nel paragrafo precedente è comunque possibile creare una domanda di mobilità essenziale (id veicolo, tempo di partenza e percorso) tramite l'utilizzo dell'applicativo DUAROUTER. Le definizioni descritte nei paragrafi precedenti hanno il fine di rendere la creazione dei veicoli quanto più dettagliata e realistica possibile. La composizione degli attributi descriventi un singolo veicolo è esposta in Tabella 2.4.

Tabella 2.4: Descrizione dei veicoli

Attributo	Tipo di Dato	Descrizione
`id`	String	L'identificativo univoco del veicolo.
`type`	String	L'identificativo della tipologia, precedentemente definita, del veicolo.
`route`	String	L'identificativo del percorso, precedentemente definito, del veicolo.
`color`	Color	Il colore del veicolo in formato RGB.
`depart`	Int	Lo step in cui il veicolo dovrebbe prender parte la simulazione.
`departLane`	Int - String (≥0) - ("random", "free", "departlane", "allowed", "best")	La corsia dell'arco iniziale in cui il veicolo verrà inserito. "departlane" identifica la prima corsia definita di un arco.
`departPos`	Float (m) - String ("random", "free", "random_free", "base")	La posizione iniziale del veicolo rispetto all'arco di partenza. "base" posiziona il veicolo all'inizio dell'arco.

Continua nella prossima pagina

Tabella 2.4: continua dalla pagina precedente

Attributo	Tipo di Dato	Descrizione
`departSpeed`	Float (m/s) - String (≥0) - (“random”, “max”)	Specifica la velocità di partenza del veicolo.
`arrivalLane`	Int-String (≥0) - (“current”)	Specifica la corsia in cui il veicolo dovrebbe terminare la simulazione.
`arrivalPos`	Float (m) - String (≥0) - (“random”, “max”)	Specifica la posizione, rispetto all’arco finale, in cui il veicolo dovrebbe terminare il suo percorso.
`arrivalSpeed`	Float (m/s) - String (≥0) - (“current”)	Specifica la velocità finale del veicolo all’ultimo step di simulazione.

Riassumendo quanto descritto precedentemente, la generazione della domanda di mobilità, e conseguentemente del file `.rou.xml`, avviene rispettando la seguente struttura:

```
<routes>
  <vType id="<STRING>" accel="<FLOAT>" maxspeed="<FLOAT>" ... />
  <vType ... />

  <route id="<STRING>" edges="<EDGE_ID_LIST>" color="<RGB color>" />
  <route ... />

  <vehicle id="<STRING>" type="<TYPE_ID>" route="<ROUTE_ID>"
   depart="<INT>" ... />
  <vehicle .../>
</routes>
```

Nel caso in cui si voglia assegnare ad un veicolo particolare un percorso che lui solo effettuerà, si può utilizzare il seguente formato:

```
<routes>
```

```
  ...
  <vehicle id="<STRING>" type="<TYPE_ID>" depart="<INT>" >
     <route edges="<EDGE_ID_LIST>" />
  </vehicle>
  ...
</routes>
```

2.4 Simulazione

Una volte completate le fasi di definizione dell'input, avendo quindi a disposizione sia una rete su cui effettuare la simulazione sia la domanda di mobilità, è possibile procedere con la simulazione vera e propria. Bisogna fare delle precisazioni circa gli input precedentemente descritti. Innanzitutto è necessario che la domanda di mobilità sia ordinata temporalmente; la ragione di ciò è che SUMO vuole realizzare simulazioni di grandi reti stradali contenenti fino a diversi milioni di differenti *route*. Utilizzando un comune personal computer ciò è possibile solamente se non tutti i veicoli ed i loro percorsi sono caricati in memoria. Il contenuto della domanda di mobilità è quindi letto sequenzialmente: partendo dallo step iniziale le nuove *route* vengono caricate in memoria ogni n step in modo da poter eseguire i successivi n step di simulazione. Si può variare il valore di n utilizzando l'opzione di SUMO `--route-steps <INT>` dove un valore negativo forza il caricamento completo del file `.rou.xml`. Il fetching delle *route* per i successivi step implica che le tipologie di veicoli e le *route* globali siano definite anticipatamente in modo che i veicoli dichiarati possano utilizzare tali parametri.

Per raffinare lo scenario realizzato è possibile includere nella simulazione file aggiuntivi che descrivono:

- *particolari logiche di funzionamento dei semafori*;
- *percorsi, comprensivi di fermate, per il trasporto pubblico*;
- *segnali stradali a velocità variabile*;

- *percorsi alternativi*.

Questi file incrementano il livello di dettaglio della simulazione ma non sono necessari ai fini degli obiettivi di questa tesi. La simulazione si compone dunque delle seguenti fasi:

1. Lettura della rete `.net.xml`;
2. Apertura del file `.rou.xml` e lettura dei primi n step;
3. Lettura (completa) di eventuali file aggiuntivi;
4. Ogni n step, lettura dei successivi n step.

Ogni simulazione ha un proprio tempo di esecuzione, questo viene specificato a riga di comando tramite le opzioni `--begin-time <INT>` ed `--end-time <INT>`. Nel caso tali parametri non vengano dichiarati, la simulazione verrà avviata allo step 0 e terminerà quando l'ultimo veicolo attivo avrà concluso il suo percorso. Tutti i veicoli con istante di partenza inferiore al `--begin-time` dichiarato vengono scartati senza prender parte alla simulazione. La simulazione viene eseguita *step-by-step*, dove ogni step rappresenta, di default, un secondo dello scenario simulato. SUMO offre la possibilità di modificare la durata degli step mediante l'opzione `--step-length <FLOAT>`, tuttavia è sconsigliato modificare tale parametro in quanto i modelli descritti ad inizio capitolo si basano su step di simulazione della durata di un secondo.

La simulazione può essere avviata da terminale utilizzando i comandi:

```
sumo -c config.sumocfg ...
sumo-gui -c config.sumocfg ...
```

Nel secondo caso sarà possibile visualizzare graficamente lo svolgersi della simulazione, purtroppo l'utilizzo di un'interfaccia di questo tipo appesantisce la simulazione: è dunque necessario precisare che le valutazioni effettuate nei prossimi capitoli sono state ottenute senza l'ausilio dell'applicativo SUMO-GUI. All'interno del file di configurazione (`config.sumocfg`) vengono specificati i vari file di input, descritti nei paragrafi precedenti, secondo la seguente struttura:

```
<configuration>
      <input>
        <net-file value="rete.net.xml"/>
        <route-files value="route.rou.xml"/>
        <additional-files value="aggiuntivi.add.xml"/>
    </input>
</configuration>
```

L'applicazione SUMO prevede varie opzioni oltre a quelle sopra specificate; un elenco completo è disponibile in [SUMO]. In questa tesi un ruolo rilevante è stato ricoperto dalle opzioni di re-instradamento (*rerouting*) che verranno descritte approfonditamente nel prossimo capitolo.

2.4.1 TraCI

TraCI è l'acronimo di *"Traffic Control Interface"* e l'idea su cui si fonda è quella di fornire l'accesso ad una simulazione mentre essa viene eseguita. Per fare questo utilizza un'architettura TCP (*Transmission Control Protocol*) client/server in cui SUMO riveste il ruolo di server ed altri applicativi esterni il ruolo di client. Tramite l'utilizzo dell'opzione `--remote-port <INT>` (dove l'intero rappresenta un numero di porta TCP, default 8813) SUMO ricoprirà il ruolo di server e conseguentemente attenderà un client che gli impartisca dei comandi. In questa fase di ascolto SUMO si limita a caricare in memoria la rete e gli input specificati nel file di configurazione secondo le modalità precedentemente descritte.

Il client può impartire comandi per controllare l'esecuzione della simulazione, per influenzare il comportamento dei singoli veicoli oppure per chiedere al server dettagli riguardanti l'ambiente simulato. SUMO risponderà ad ognuno di questi comandi con degli status di risposta e informazioni aggiuntive a seconda della tipologia di comando. Lo studio delle risposte e della composizione dei messaggi verrà approfondito nel capitolo 4.

Il client dovrà provvedere anche all'avanzamento della simulazione; infatti, ogni qual volta lo ritenga necessario, impartirà a SUMO il comando

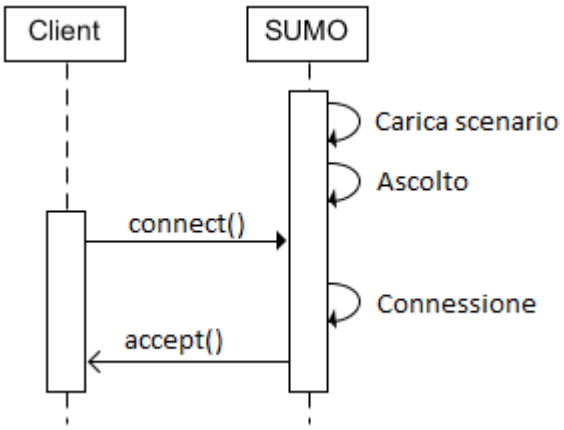

Figura 2.2: TraCI: apertura di una connesione *(tratto da [SUMO])*

responsabile del passaggio allo step successivo. L'applicativo esterno è inoltre tenuto a terminare la connessione attraverso un comando di chiusura; nel caso tale comando non venga inviato la simulazione potrebbe, in linea ipotetica, proseguire all'infinito.

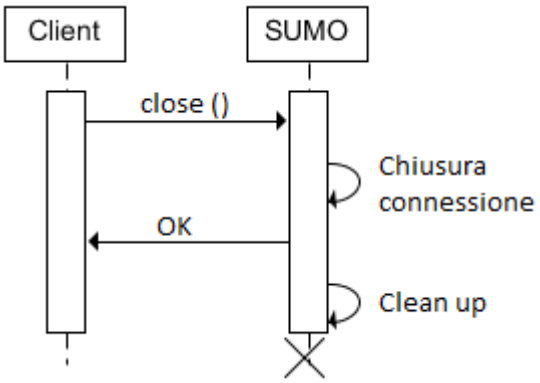

Figura 2.3: TraCI: chiusura di una connesione*(tratto da [SUMO])*

Capitolo 3

Problematiche di rerouting

Le prime sperimentazioni di MoViT, effettuate dagli sviluppatori dell' *Università della California di Los Angeles*, sono state soddisfacenti [12]; tali esperimenti hanno permesso di confrontare la realtà emulata rispetto ad uno scenario in piccola scala realmente eseguito. Come si può notare da [12], l'emulazione di MoViT è consistente rispetto alla realtà. Sono state inoltre svolte alcune emulazioni a larga scala con il fine ultimo di verificare che la generazione dei dati di mobilità prodotti da SUMO e la successiva distribuzione ai vari *Channel Module* (procedimento descritto in 1.2.1) non comportasse ritardi eccessivi al crescere del numero di veicoli riprodotti. In condizioni "standard", in uno scenario descrivente un'area urbana di 16 k/m^2 con 700 veicoli attivi, oltre il 99% dei dati di mobilità viene mediamente consegnato ai *Channel Module* entro 300 ms. Considerando la struttura ad albero descritta in 1.3.1.1, il ritardo massimo sperimentato (con un tasso di ricorrenza di $\sim 10^{-5}$) è stato di 960, 970, e 820 ms in configurazioni composte rispettivamente da un numero massimo di figli pari a 2, 3 e 10. Nel paragrafo 1.2.1 è stata definita l'esigenza di considerare un sufficiente lasso di tempo tra la generazione di uno step di mobilità e la sua effettiva applicazione nella rete emulata; i risultati sopra esposti consentono di definire un *lower bound* per tale intervallo pari ad 1 secondo. L'utilizzo di intervalli di tempo ridotti, tra generazione ed applicazione di uno step di mobilità, consente l'emulazione

di una vasta gamma di applicazioni e protocolli che richiedono un'interazione a *"ciclo chiuso"* tra i nodi della rete ed il simulatore di mobilità. Tra questi applicativi, ad esempio, troviamo gli ottimizzatori di flussi di traffico veicolare, nei quali è necessario re-instradare i veicoli in reazione ad eventi come ingorghi o incidenti. Le prime fasi di test hanno tuttavia evidenziato un eccessivo rallentamento nella produzione di dati di mobilità in condizioni di *re-routing*. In questo capitolo verrà verificata la veridicità di questa affermazione e successivamente verrà analizzato, quantificato e motivato tale ritardo.

3.1 Rerouting

Con *rerouting*, letteralmente re-instradamento, si intende la procedura di assegnamento di un nuovo percorso ad uno o più veicoli. Tale procedura comporta una valutazione della rete e di tutti i possibili percorsi congiungenti un arco sorgente ad un arco destinazione. SUMO offre la possibilità di effettuare rerouting in 2 modalità: utilizzando file addizionali oppure tramite l'aggiunta di opzioni alla simulazione.

Nel primo caso è necessaria la creazione esplicita di file addizionali contenenti la dichiarazione XML di uno o più *"rerouter"*. La dichiarazione di un *rerouter* si compone di un identificativo, di una lista di archi, di una probabilità di applicazione e del riferimento ad un altro file XML che descriva il comportamento del *rerouter* (chiusura di una strada, nuova destinazione oppure cambio di percorso) in corrispondenza della lista di archi precedentemente definita. Nel secondo caso, invece, si impongono possibili reroute simulando la presenza di un dispositivo di instradamento interno ad un veicolo (ad esempio un navigatore satellitare) in grado di definire il percorso ottimo. La prima modalità consente di modellare particolari situazioni offrendo un maggior livello di dettaglio alla simulazione che si intende realizzare, tuttavia, visti gli scopi di questa valutazione, si è scelto di utilizzare le opzioni fornite da SUMO, più immediate e funzionali rispetto alla definizione dei vari

rerouter.

Sono state prevalentemente utilizzate 2 delle opzioni messe a disposizione del simulatore: la prima, `--device.rerouting.explicit <VEHICLE ID>`, impone la presenza di un dispositivo di routing al veicolo specificato. L'opzione `--device.rerouting.probability <FLOAT>`, invece, tramite la definizione di un numero reale compreso tra 0 e 1, definisce la probabilità che un veicolo sia dotato di un dispositivo di instradamento e conseguentemente la probabilità di rerouting del veicolo. In entrambi i casi il calcolo del percorso ottimo avviene all'ingresso del veicolo nell'ambiente simulato. Come per l'applicazione DUAROUTER i percorsi ottimi vengono determinati utilizzando l'algoritmo di Dijkstra (descritto in [28]) basato sugli archi della rete.

3.2 Definizione scenari

Per la sperimentazione e l'analisi delle prestazioni di SUMO si è scelto di definire ed utilizzare scenari ad urbanistica stradale reticolare, modello tipico su cui sono basate, ad esempio, le città statunitensi. Ulteriore motivo di questa scelta è la creazione automatica delle *connessioni*, descritte al capitolo precedente, da parte di NETCONVERT; le *connessioni* create dall'applicazione rispondono più fedelmente a scenari statunitensi piuttosto che europei. In questa fase di selezione degli scenari lo strumento maggiormente utilizzato è stato il database a licenza libera fornito da OpenStreetMap [OSM]. Visto l'interesse nello sviluppare simulazioni su scenari piuttosto vasti, si è resa necessaria la ricerca di depositi di mappe OSM in scala continentale o statale, come ad esempio [GEOFABRIK]. Motivo di ciò è il vincolo imposto da OpenStreetMap riguardante il download di aree contenenti un numero di nodi inferiore a 50.000. Nello specifico sono state utilizzate due mappe rappresentanti rispettivamente la costa occidentale e la costa orientale degli Stati Uniti. Successivamente, per ridurre le dimensioni dell'area interessata a quelle di una grande città, è stato utilizzato il software Java OpenStreetMap

[JOSM]. Tramite JOSM è possibile selezionare l'area geografica che si intende esportare e, nel caso lo si ritenga opportuno, modificarla tramite l'interfaccia grafica del software. È bene far presente che risulta conveniente modificare la rete stradale utilizzando JOSM piuttosto che alterare successivamente i file XML che verranno generati ed utilizzati per la simulazione. La scelta degli scenari è ricaduta su una porzione centrale dell'area metropolitana di Los Angeles (composta da circa 70.000 nodi) e una larga parte della città di New York (l'intera Manhattan e alcune zone nelle immediate vicinanze dell'isola; lo scenario si compone di circa 100.000 nodi). Una volta definite le 2 aree con l'ausilio di JOSM, sono stati generati i rispettivi file in formato `.osm`. Come illustrato al capitolo 2, lo strumento utilizzato in questa fase è stato NETCONVERT il cui scopo è la conversione e la creazione di reti idonee a SUMO. Tramite il seguente comando:

```
netconvert --osm-files NewYork.osm.xml -o NewYork.net.xml
```

è stato possibile convertire la mappa OSM, relativa in questo caso allo scenario di New York, in una rete `.net.xml` utilizzabile dal simulatore. Il risultato finale della creazione degli scenari è visibile alle Figure 3.1 e 3.2.

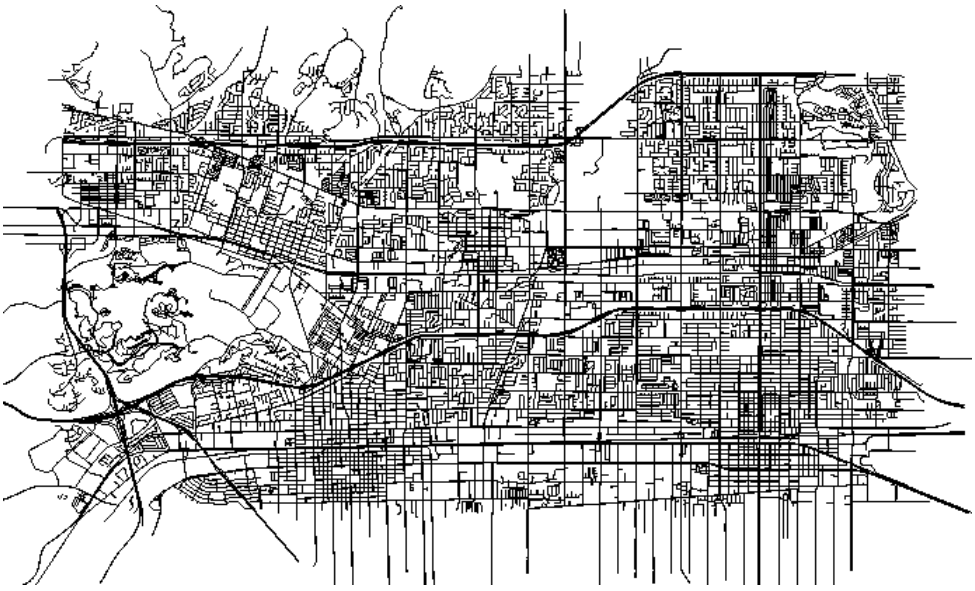

Figura 3.1: Scenario: Los Angeles

Figura 3.2: Scenario: New York

Per la creazione dei *trip* è stato utilizzato lo script python `randomTrips.py` precedentemente descritto. Alla generazione casuale dei *trip* è seguito l'utilizzo di DUAROUTER per la definizione delle *route*. Con il fine di valutare le prestazioni del simulatore al crescere della domanda di mobilità, sono state realizzate differenti configurazioni composte da 50, 500, 1000 e 5000 veicoli.

3.3 Sperimentazione

Una volta definite la rete stradale e la domanda di mobilità, è stata verificata la veridicità dell'affermazione riguardante i ritardi in fase di rerouting sperimentati durante i test di MoViT. Sono state dunque effettuate le simulazioni delle configurazioni precedentemente create per gli scenari di Los Angeles e New York con tassi di probabilità di rerouting crescenti. I tempi di calcolo sono una media di 10 esecuzioni differenti effettuate con le medesime configurazioni e le stesse opzioni di simulazione. I risultati sono esposti nelle Tabelle 3.1 e 3.2.

Tabella 3.1: Rerouting Los Angeles

N.Veicoli	**% Rerouting**	**Tempo** *(ms)*
50	0	5154
	1	47338
	10	48530
	30	49254
	50	49589
	90	49957
500	0	12991
	1	87136
	10	89609
	30	90250
	50	93534

Continua nella prossima pagina

Tabella 3.1: continua dalla pagina precedente

N.Veicoli	% Rerouting	Tempo *(ms)*
	90	96721
1000	0	24860
	1	165281
	10	166263
	30	168141
	50	173441
	90	179897
5000	0	72818
	1	304953
	10	310262
	30	312904
	50	343349
	90	373027

Tabella 3.2: Rerouting New York

N.Veicoli	% Rerouting	Tempo *(ms)*
50	0	11723
	1	100913
	10	103151
	30	106878
	50	108009
	90	110910
500	0	25896
	1	177224
	10	179023

Continua nella prossima pagina

Tabella 3.2: continua dalla pagina precedente

N.Veicoli	**% Rerouting**	**Tempo** *(ms)*
	30	182358
	50	185939
	90	189843
1000	0	35412
	1	278427
	10	280731
	30	283423
	50	288557
	90	296503
5000	0	154263
	1	690249
	10	700418
	30	710981
	50	720027
	90	745845

3.4 Analisi problematiche rerouting

Analizzando le tabelle sopra esposte è facilmente osservabile come una richiesta di rerouting, senza la specifica di altre opzioni di simulazione, implichi un'esplosione dei tempi di computazione. È evidente che tale incremento non dipenda dalla percentuale di veicoli che si ha intenzione di re-instradare; si può infatti notare come la differenza tra tempi di computazione della medesima configurazione sia, in rapporto alla probabilità applicata, sempre massima tra una simulazione "libera" (senza re-instradamento) ed una simulazione con l'1% di veicoli re-instradati; la differenza che, invece, intercorre tra un reroute all'1% di probabilità ed un reroute, ad esempio, al 90% si è sempre rivelata, in proporzione, inferiore a quella precedentemente descritta. Sono

stati inoltre effettuati alcuni test con reroute esplicito di un solo veicolo: il risultato in tempi di computazione si è rivelato il medesimo di un reroute a probabilità qualsiasi, con tempi di calcolo "non consoni" rispetto ad una simulazione "libera". Utilizzando i report prodotti in fase di simulazione emerge che l'algoritmo di Dijkstra implementato da SUMO impiega, nel caso peggiore, qualche decina di millisecondi per computare il percorso ottimo di un determinato veicolo. Considerando inoltre che i tempi di calcolo non risentono eccessivamente della computazione di più percorsi durante una simulazione, è plausibile ritenere che le cause di questi ritardi siano dovute ad altri fattori.

3.4.1 Motivazione del ritardo

A seguito dei risultati delle tabelle 3.1 e 3.2, ed in considerazione del fatto che la documentazione del simulatore non offre spiegazioni riguardanti questo apparentemente ingiustificato incremento dei tempi di calcolo, si è resa necessaria un'approfondita fase di analisi e debugging del codice sorgente di SUMO[1]. Lo scopo della prima fase di debugging è stato quello di evidenziare quali classi e metodi venissero richiamati dall'applicazione in presenza di comandi di reroute rispetto a simulazioni senza re-instradamenti. Una volta identificate tali porzioni di codice si è provveduto ad analizzarle tramite debug passo a passo.
La principale differenza tra una simulazione avviata con una qualsiasi opzione di reroute ed una simulazione avviata senza la specifica di alcuna opzione è l'intervento del metodo `execute`, appartenente alla classe `MSEventControl`, definito all'interno del file `MSEventControl.cpp`. Lo scopo di questa classe è immagazzinare gli eventi temporalmente dipendenti ed eseguirli, tramite il metodo `execute`, al momento opportuno. Gli eventi di reroute fanno dunque parte degli eventi gestiti dalla classe `MSEventControl`. L'esecuzione di

[1]SUMO è interamente scritto in C++, eccezion fatta per alcuni tool utilizzanti Python. Il debugging di SUMO è stato effettuato tramite Microsoft Visual Studio 2010 [VS2010], utilizzando quindi la build (versione 0.15.0) per ambienti Windows.

eventi di reroute comporta la computazione del percorso ottimo a cui segue il re-instradamento del veicolo interessato. Infine, gli eventi di questo tipo, si concludono con l'adattamento dei pesi degli archi componenti la rete stradale. Tramite il debugging del codice è stato possibile identificare la fase di adattamento come "collo di bottiglia" degli eventi di reroute.
Questo aggiornamento avviene alla fine di ogni step di simulazione tramite la chiamata del metodo `adaptEdgeEfforts` appartenente alla classe `MSDevice_Routing`; lo scopo di questo metodo è aggiornare il peso di tutti gli archi componenti la rete simulata tenendo in considerazione il peso di un determinato arco allo step precedente. La documentazione di SUMO non fornisce spiegazioni riguardanti al come il peso degli archi venga aggiornato, infatti la descrizione di tale processo viene, ad oggi, classificata nella *"Todo list"* ufficiale. Tramite l'analisi del codice è stato tuttavia possibile definire i parametri ed i fattori determinanti tale adattamento. L'equazione 3.1 descrive la procedura di adattamento del peso a cui è sottoposto, ad ogni step t, ciascun arco E componente la rete stradale:

$$E_{(t)} = E_{(t-1)}\alpha + P_{E_{(t)}}(1-\alpha) \tag{3.1}$$

Dove $P_{E_{(t)}}$ è il tempo di percorrenza di un arco in un determinato step e α è un parametro reale compreso tra 0 ed 1 (di default impostato a 0.5) che identifica l'adattamento del peso a cui si vuole sottoporre l'arco. Il tempo di percorrenza $P_{E_{(t)}}$ è il valore che viene restituito dalla funzione `getCurrentTravelTime`. Questa funzione calcola la somma delle velocità medie di tutte le corsie componenti l'arco; successivamente divide tale somma per il numero di corsie così da ottenere una velocità media dell'intero arco. Nel caso in cui la velocità media calcolata sia pari a 0 (ovvero il traffico veicolare sia fermo) allora `getCurrentTravelTime` restituirà il valore simbolico di "1000000", ovvero un valore molto alto il cui scopo è rendere il peso dell'arco estremamente elevato; in caso contrario, invece, il valore restituito sarà il tempo di percorrenza (in secondi) dell'arco E allo step t, ovvero:

$$P_{E_{(t)}} = \frac{L_E}{\overline{V}_{E_{(t)}}} \tag{3.2}$$

dove L_E è la lunghezza dell'arco E mentre $\overline{V}_{E_{(t)}}$ è la velocità media dell'arco E allo step t calcolata come segue:

$$\overline{V}_{E_{(t)}} = \frac{\sum_{c=1}^{n} \overline{V}_{c_{(t)}}}{n} \tag{3.3}$$

dove n è il numero di corsie c componenti l'arco E e $\overline{V}_{c_{(t)}}$ è la velocità media della corsia c all'istante t. Infine, $\overline{V}_{c_{(t)}}$ è così calcolata:

$$\overline{V}_{c_{(t)}} = \frac{\sum_{v=1}^{m} V_{v_{(t)}}}{m} \tag{3.4}$$

in cui V_v è la velocità allo step t del veicolo v ed m rappresenta il numero di veicoli percorrenti una determinata corsia ad un dato istante t. Nel caso in cui m sia uguale a 0, ovvero non vi siano veicoli percorrenti la corsia considerata, allora il valore di $\overline{V}_{c_{(t)}}$ sarà pari alla velocità massima consentita in tale corsia e definita dal file `.net.xml` descrivente la rete.

Il procedimento esposto, come già affermato, viene applicato ad ogni step di simulazione su tutti gli archi della rete. È evidente quanto tale processo sia computazionalmente esoso in considerazione al numero di archi presenti negli scenari realizzati (146.000 archi per lo scenario di Los Angeles, 215.000 per New York). Bisogna inoltre tenere conto del fatto che la procedura di adattamento viene eseguita anche in presenza del reroute esplicito di un solo veicolo e continua anche qualora il veicolo abbia già abbandonato la simulazione. Questo motiva le ridotte differenze tra i tempi al crescere della percentuale di reroute.
Per la generazione dei dati di mobilità in tempi accettabili, in caso di reroute di uno o più veicoli, è dunque necessario rinunciare o perlomeno modificare il processo di aggiornamento dei pesi. SUMO offre 2 opzioni di simulazione che consentono di migliorare o addirittura annullare l'adattamento; tuttavia queste opzioni sono scarsamente documentate e non è presente una reale descrizione del perchè sia conveniente utilizzarle o di che cosa realmente rappresentino. Soltanto grazie all'analisi del codice è stato possibile determinarne l'importanza e, inoltre, considerando i tempi di calcolo registrati

è quantomeno necessario informare l'utilizzatore riguardo ai pesanti ritardi introdotti dal ri-adattamento della rete in funzione del traffico. La prima di queste opzioni, `--device.rerouting.adaptation-weight <FLOAT>`, consente la modifica del parametro α descritto all'equazione (3.1). La seconda opzione, `--device.rerouting.adaptation-interval <INT>`, ha un ruolo estremamente importante in quanto consente la modifica dell'intervallo di tempo tra una procedura di adattamento dei pesi e l'altra. Impostando tale intervallo a 0 si rinuncia all'aggiornamento dei pesi; in conseguenza di ciò, l'algoritmo di Dijkstra computerà percorsi ottimi sulla base dello stato iniziale della rete ovvero senza tenere conto della presenza di traffico. Eliminando l'adattamento, il veicolo reinstradato sceglierà il percorso più breve senza valutare le reali condizioni di traffico ovvero considerando il peso dell'arco come se la strada rappresentata sia libera. In conseguenze di ciò potrebbe scegliere una strada congestionata oppure una strada ad una sola corsia percorsa da un veicolo estremamente lento. Facendo un esempio esplicativo è come se il veicolo in questione fosse fornito di un dispositivo non in grado di ricevere aggiornamenti riguardanti il traffico; l'intervallo di adattamento di default (ogni step) consente invece una scelta dei percorsi in "tempo reale". L'eliminazione o l'allungamento dei tempi del processo di adattamento non ha risvolti sulla normale percorrenza di un arco da parte di un veicolo. Il peso dell'arco, e quindi l'adattamento di esso, sono funzionali solamente all'algoritmo di rerouting.

Nel prossimo capitolo verranno effettuate alcune valutazioni e confronti tra le prestazioni di:

- Reroute di veicoli tramite SUMO con `adaptation-interval` pari a 0;
- Reroute di veicoli tramite l'interfaccia di controllo del traffico TraCI, implementata in Python, fornita dal pacchetto SUMO;

- Reroute di veicoli tramite un'implementazione, realizzata per questa tesi, in linguaggio C di TraCI.

Capitolo 4

TraCI: Implementazione e valutazione sperimentale

I risultati ottenuti in presenza di opzioni di re-instradamento hanno inizialmente portato alla ricerca di soluzioni in grado di ovviare a tale problema; è prevalentemente per questo motivo che le valutazioni svolte in questo capitolo hanno riguardato in maniera particolare il re-instradamento di veicoli. L'analisi del codice descritta al capitolo precedente ha tuttavia consentito una parziale risoluzione delle problematiche di rerouting tramite le opzioni fornite dal simulatore stesso. L'idea alla base della ricerca di soluzioni alternative è quella di cedere il controllo della simulazione ad un'applicazione esterna impartendo a SUMO, tramite TraCI, soltanto i comandi strettamente necessari, con la speranza di ridurre così il carico di lavoro. Utilizzando TraCI è inoltre possibile modellare la simulazione in fase di esecuzione, permettendo di variare il comportamento dei singoli veicoli o di modificare lo scenario in cui essi si muovono. In questo capitolo verrà proposta un'implementazione alternativa di TraCI in ANSI C, a cui seguiranno una serie di valutazioni prestazionali.

4.1 Descrizione del protocollo

Nel capitolo 2 sono state introdotte le principali caratteristiche di TraCI. In particolare sono stati descritti i flussi di apertura e chiusura della connessione e l'opzione necessaria ad avviare SUMO in modalità server (`--remote-port <PORT>`). TraCI utilizza una connessione TCP per permettere la comunicazione tra client e server; una volta che la connessione è stabilita, il client controlla il simulatore tramite un protocollo di scambio dati. Tale scambio di dati consiste nell'invio di richieste (sotto forma di comandi) verso il server; a seguito della ricezione di una richiesta, SUMO computa le operazioni necessarie all'esecuzione di essa e, per ogni comando eseguito, invia una o più risposte che ne descrivono l'esito. La sincronizzazione temporale tra client e server è gestita dal client attraverso l'invio periodico di comandi necessari all'avanzamento della simulazione.

4.1.1 Struttura dei messaggi

Ogni segmento TCP funge da contenitore per una lista di comandi o di risultati. Ciascun pacchetto pertanto si compone di una breve intestazione (*header*), che descrive la dimensione complessiva del messaggio in byte, e di un insieme di comandi accodati ad essa. La lunghezza e l'identificativo del comando vengono posizionati immediatamente prima del contenuto di esso. In Figura 4.1 viene rappresentata la composizione di un pacchetto TraCI contenente n comandi. In alcuni casi la lunghezza standard di un singolo comando, in byte, può non essere sufficiente, in quanto limitata al valore di 255 (corrispondente al valore massimo rappresentabile mediante un byte senza segno). In questi casi si può utilizzare una "lunghezza estesa": impostando il campo *Lunghezza* di un determinato comando pari a 0 e aggiungendo un campo di tipo intero a 32 bit sarà possibile descrivere lunghezze rappresentanti fino a $2^{31} - 1$ byte. Questo meccanismo è specificato in Figura 4.2.

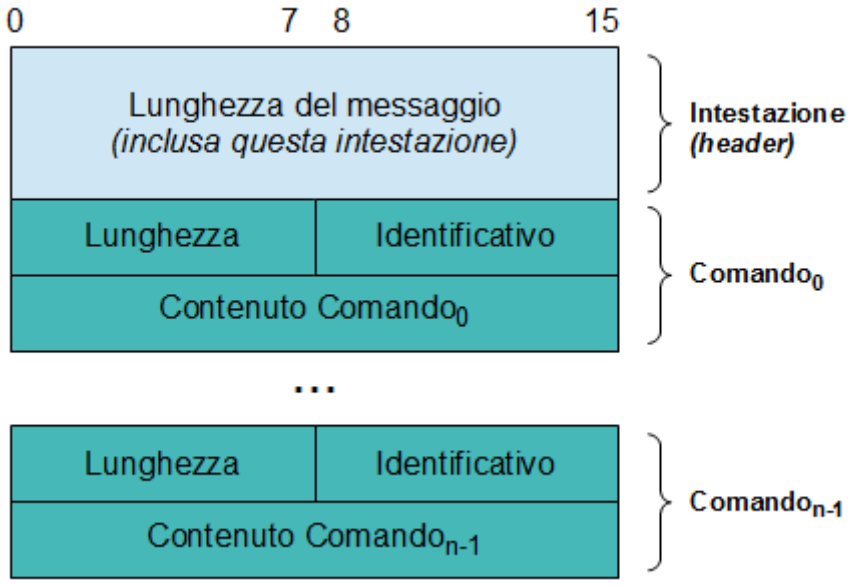

Figura 4.1: Formato dei messaggi in TraCI: i messaggi si compongono di un header e di un numero variabile di comandi.

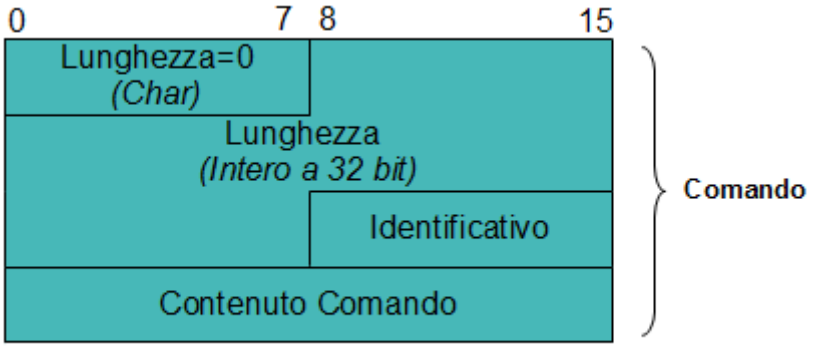

Figura 4.2: Formato dei comandi con lunghezza estesa in TraCI: il campo originario *Lunghezza* è impostato a 0, i successivi 32 bit sono occupati da un intero senza segno.

Il server invia ad ogni comando una risposta che rispetta il formato esposto alle Figure 4.1 e 4.2. In tal caso il campo *Identificativo* rappresenta il comando a cui si sta rispondendo. La parte iniziale del campo *Contenuto Comando* descrive l'esito, detto anche *stato*, della richiesta effettuata dal client dove il primo byte può assumere i seguenti valori:

- `0x00`: in caso di successo;
- `0xFF`: in caso di fallimento;
- `0x01`: in caso il comando richiesto non sia implementato.

In aggiunta allo stato, SUMO può aggiungere una stringa che descriva, ad esempio, le motivazioni di una richiesta fallita.

4.1.2 Tipi di dato

Tutti i messaggi in TraCI si compongono di uno *stream* di tipi di dato elementari e composti. La Tabella 4.1 mostra i tipi di dato supportati dal protocollo ed il loro formato.

Tabella 4.1: Tipi di dato supportati da TraCI

Tipo di dato	Dimensione	ID	Descrizione
`ubyte`	8 bit	`0x07`	Numeri interi (da 0 a 255).
`byte`	8 bit	`0x08`	Numeri interi (da -128 a +127).
`integer`	32 bit	`0x09`	Numeri interi (da -2^{31} a $2^{31}-1$).
`float`	32 bit	`0x0A`	Numeri in formato floating point IEEE754 (uso deprecato da SUMO 0.13.0).
`double`	64 bit	`0x0B`	Numeri in formato floating point IEEE754.

Continua nella prossima pagina

Tabella 4.1: continua dalla pagina precedente

Tipo di dato	Dimensione	ID	Descrizione
`string`	variabile	`0x0C`	32 bit iniziali che indicano la lunghezza della stringa seguiti da testo codificato in caratteri ASCII a 8 bit.
`stringList`	variabile	`0x0E`	32 bit che indicano il numero (n) di stringhe appartenenti alla lista seguiti da n stringhe.
`compound object`	variabile	`0x0F`	Oggetti composti, la struttura interna dipende dall'oggetto rappresentato.

La comprensione dei tipi di dato sopra esposti e di come essi vengono utilizzati dal protocollo di scambio dati è di fondamentale importanza al fine di interpretare correttamente le informazioni trasmesse tramite TraCI. Per quanto riguarda la descrizione degli oggetti composti si rimanda alla documentazione ufficiale in [SUMO].

4.2 Comandi in TraCI

Fulcro del controllo della simulazione attuato da TraCI sono i comandi. In questa sezione verranno descritti in particolare lo scopo ed il formato delle varie tipologie di comando. Essi possono essere classificati in 3 categorie distinte:

- *generali*;
- *recupero* informazioni;
- *modifica*.

I comandi *generali* non utilizzano alcun identificativo come parametro input, di conseguenza non vi è interazione diretta con gli oggetti presenti nella simulazione. Tra i comandi più importanti vi sono `Simulation Step` e `Close`, il cui scopo è, rispettivamente, l'avanzamento della simulazione e la chiusura della connessione TCP nel momento in cui il client lo ritenga opportuno. In questa categoria vi sono poi una serie di comandi legati allo scenario riprodotto il cui scopo è ricevere informazioni globali riguardanti l'ambiente oppure il calcolo della distanza tra due punti della mappa.

Nella seconda categoria, *comandi di recupero*, vi sono tutti i comandi il cui scopo è informare il client riguardo ad aspetti particolari della simulazione. È necessaria la specifica di un identificativo dell'oggetto di cui si vogliono ottenere informazioni, come ad esempio una corsia, un percorso, un veicolo, una strada e così via. Il formato di questi comandi e della successiva risposta di SUMO è descritto rispettivamente dalle Figure 4.3 e 4.4.

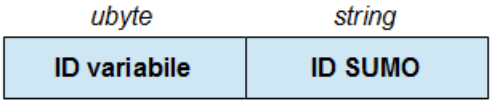

Figura 4.3: Formato dei comandi di recupero in TraCI. L'ID variabile è un valore dipendente dal comando inviato. L'ID SUMO invece fa riferimento all'identificativo dichiarato nei file `.xml` input della simulazione.

Figura 4.4: Formato della risposta ad un comando di recupero in TraCI. L'ID variabile e l'ID SUMO ripetono i valori definiti dal comando. Il campo `<RETURN_TYPE>` contiene la specifica dell'identificativo del tipo di dato (si veda la Tabella 4.1).

Ultima categoria di comandi è quella relativa alla *modifica* di elementi presenti nella simulazione come, ad esempio, corsie, veicoli, strade e semafori. Questi comandi consentono l'alterazione dell'esecuzione in corso d'opera. Il formato di questi comandi è descritto in Figura 4.5; a differenza dei comandi di *recupero* SUMO risponde comunicando il solo stato della richiesta senza aggiungere ulteriori informazioni.

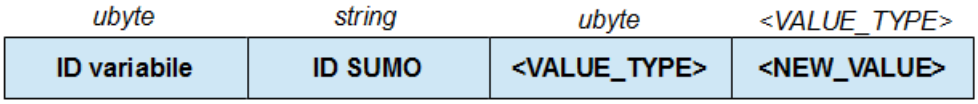

Figura 4.5: Formato dei comandi di modifica in TraCI. Come per i comandi di *recupero* sono presenti l'ID della variabile e l'ID dell'elemento che si intende modificare. Successivamente viene descritto il tipo di dato e il nuovo valore da assegnare.

Considerata la quantità di comandi previsti da TraCI, per un elenco completo si rimanda alla documentazione ufficiale di [SUMO].

4.3 Implementazione in ANSI C

Attualmente SUMO fornisce due implementazioni dell'interfaccia TraCI: una utilizzante il linguaggio di programmazione Python, ed un'altra utilizzante Java. Questa seconda implementazione non dispone tuttavia di una documentazione ufficiale e non è quindi stata considerata per le valutazioni proposte a fine capitolo.
Python è un linguaggio di programmazione estremamente versatile che fa della flessibilità e della portabilità il suo punto di forza. Essendo però un linguaggio interpretato presenta delle carenze per quel che riguarda la velocità di esecuzione. È dunque ragionevole supporre che un'implementazione di TraCI in linguaggio compilato possa offrire prestazioni migliori di una in un linguaggio interpretato. Per questo motivo si è scelto di cercare di migliorare le performance di TraCI realizzando un'implementazione dell'interfaccia

in ANSI C. Tutto il codice prodotto viene rilasciato con licenza pubblica generica GPL (*General Public License*). L'implementazione si compone di una serie di header file e dei corrispondenti file di definizione. Tra i più importanti:

- **`init.h/c`**: definiscono ed implementano le funzioni di inizializzazione della connessione, invio e ricezione dati, composizione dei messaggi;
- **`DT.h/c`**: definiscono i tipi di dato astratti e le funzioni per poterli utilizzare, implementano le funzioni di lettura e scrittura buffer e le funzioni di controllo delle risposte;
- **`simulation.h/c`, `vehicle.h/c`**: contengono la definizione dei comandi relativi agli aspetti della simulazione e dei veicoli secondo i moduli descritti in [SUMO].

Verranno ora descritte le strutture dati realizzate, la fase di scambio informazioni tra client e server, le procedure di composizione dei messaggi e le principali caratteristiche dell'implementazione realizzata.

4.3.1 Strutture dati

Per poter comprendere correttamente le fasi successivamente descritte è importante specificare le strutture dati realizzate in questa implementazione. I comandi inviati dal client verso il server devono essere accodati al fine di confrontare ordinatamente le risposte provenienti dal server. A tale scopo sono state realizzate le seguenti strutture dati, definite all'interno del file `DT.h`:

```
struct message
{
       unsigned char command;
       struct message *next;
};
```

```
struct msg_queue
{
        struct message *head;
        struct message *tail;
};
```

Queste strutture dati rispettano il tipico formato di liste e code. Ogni elemento di lista di tipo `message` è quindi composto dalla variabile `command` rappresentante l'identificativo del comando e da un puntatore all'elemento successivo della lista. La coda `msg_queue` mantiene il riferimento agli elementi di inizio e fine della lista allo scopo di poter realizzare le classiche operazioni di pop e push. All'interno di `DT.h` è stato inoltre definito un altro tipo di dato, *list*, più generico ed utilizzato in vari frangenti:

```
typedef struct list
{
        unsigned char *content;
        struct list *next;
} list;
```

dove `content` è un array di unsigned char allocato dinamicamente in base alle esigenze dell'elemento di lista che si intende realizzare. Ogni comando è caratterizzato da una struttura contenente tutte le variabili inerenti a quel comando. Ad esempio, la struttura che descrive il comando di recupero informazioni (definito in 4.2) relativo alle variabili di simulazione è così implementata:

```
typedef struct st_simulation
{
        int timeStep;
        int loadedVehiclesNumber;
        list loadedVehiclesIDS;
        int departedVehiclesNumber;
```

```
        list departedVehiclesIDS;
        int arrivedVehiclesNumber;
        list arrivedVehiclesIDS;
        int minExpectedVehicles;
        int teleportStartingVehNumber;
        list teleportStartingVehIDS;
        int teleportEndingVehNumber;
        list teleportEndingVehIDS;
        int delta_t;
} st_simulation;
```

Ogni comando è caratterizzato da un proprio identificativo, così come le variabili che lo compongono. Considerando `st_simulation` l'identificativo del comando di recupero variabile da questa struttura è `0xAB` mentre la variabile, ad esempio, `int timeStep` è caratterizzata dall'ID `0x70`. Tutti gli identificativi sono contenuti all'interno dell'header `constants.h` sotto forma di costanti.

4.3.2 Inizializzazione della connessione

Come già illustrato è necessario che il server sia in ascolto su una determinata porta e, successivamente, è compito del client inizializzare la connessione su tale porta. Per fare in modo che l'implementazione realizzata fosse utilizzabile sia su piattaforme Linux che su sistemi Windows è stato necessario includere librerie specifiche per i differenti ambienti. Nel caso di Windows sono state utilizzate le API (*Application Programming Interface*) Winsock tramite l'inclusione degli *header* `WinSock2.h` e `WS2tcpip.h` il cui scopo è riprodurre la struttura delle socket Unix. Per quanto riguarda Linux sono state utilizzate le API BSD sockets (*Berkeley sockets*) definite dagli *header*: `sys/types.h`, `sys/socket.h`, `netinet/in.h`, `netinet/tcp.h`, `arpa/inet.h`, `netdb.h`. Tramite l'inclusione di questi file, così come per le Winsock, è possibile realizzare processi in grado di comunicare in rete.

L'inizializzazione della connessione avviene tramite una chiamata alla funzione `init()`. Scopo di `init()` è realizzare la connessione tra client e server sulla base dell'indirizzo IP dichiarato e della porta su cui è in ascolto SUMO (di default tali parametri sono l'indirizzo locale `127.0.0.1` e la porta `8813`). Nel caso in cui non sia possibile stabilire una connessione, la funzione `init()` interrompe l'esecuzione dell'applicazione client comunicando il motivo di tale interruzione.

4.3.3 Invio dati

L'invio dei comandi da parte del client avviene tramite chiamate alla funzione `sendExact()`. Questa funzione determina innanzitutto la quantità di dati presente all'interno del buffer in uscita. Ciò avviene tramite l'utilizzo di una variabile intera (`buf_pos`) che mantiene, durante ogni ciclo di scambio dati, l'indice dell'ultima posizione del buffer occupata. Una volta determinata la dimensione del buffer è possibile inviare i dati al server tramite una chiamata alla funzione `send`, facente parte delle API sopra citate, caratterizzata dalla seguente signature:

```
ssize_t send(int socket,const void *buffer,size_t length,int flags)
```

dove `length` corrisponde alla dimensione del `buffer` precedentemente calcolata. Il parametro `flags` consente di modificare alcuni aspetti riguardanti il comportamento delle socket; tuttavia, sia in fase di invio che in fase di ricezione, tale variabile è sempre impostata a 0, ciò corrisponde alla specifica del comportamento di default. Il buffer in uscita è un array statico di unsigned char. La decisione di allocare staticamente la memoria relativa a `buffer_out` è dipesa dalla valutazione delle ridotte dimensioni dei messaggi inviati verso il server. Una volta terminato l'invio dei dati, il client esegue una chiamata alla funzione `recvExact()` che si occupa della ricezione dati.

4.3.4 Ricezione dati

Scopo della funzione `recvExact()` è offrire al client la possibilità di ricevere messaggi di risposta da parte di SUMO. A differenza di quanto affermato per il buffer di uscita, non potendo conoscere a priori la dimensione del flusso di dati proveniente dal server, è necessario allocare dinamicamente il buffer di ingresso. Tale scelta è dipesa sia dall'esigenza di gestire nel modo più efficace possibile la memoria allocata, sia, e soprattutto, per evitare possibili condizioni di buffer overflow. Per attuare questa allocazione dinamica è necessario considerare la struttura dei messaggi in TraCI (descritta in 4.1.1). Sapendo che i primi 4 byte di un messaggio sono destinati alla specifica della lunghezza totale del messaggio stesso, la funzione `recvExact()` alloca innanzitutto lo spazio necessario al buffer in ingresso, `buffer_in`, per ricevere questi primi byte. Questa ricezione avviene tramite la chiamata alla funzione `recv`, contenuta nelle API precedentemente citate, descritta dalla seguente signature:

```
ssize_t recv(int socket,const void *buffer,size_t length,int flags)
```

dove `length`, in questo caso, corrisponde alla dimensione di un intero ovvero ai 4 byte che specificano la lunghezza totale del messaggio. Una volta ottenuta questa informazione è possibile allocare la restante parte di `buffer_in` e successivamente ricevere la quantità di dati, corrispondente alla lunghezza totale ricevuta dalla prima `recv`, esclusi i primi 4 byte del flusso già inviati dal server. Una volta terminata la ricezione dei dati provenienti dal server, `recvExact()` restituisce il controllo a `sendExact()` che completa questo scambio controllando le risposte ai vari comandi ovvero il contenuto di `buffer_in`.

4.3.5 Controllo delle risposte

Come affermato in 4.1.1, TraCI invia ad ogni comando una risposta contenente l'esito della richiesta effettuata dal client. Possiamo definire la prima

parte di questa risposta come *"prefisso"*. Questo *prefisso* rispetta la struttura descritta dalle Figure 4.1 e 4.2 e si compone di almeno 7 byte così distribuiti:

- **1 byte**: lunghezza del prefisso;
- **1 byte**: ID del comando a cui si sta rispondendo;
- **1 byte**: esito della richiesta;
- **4 byte**: intero che indica la lunghezza n dell'eventuale stringa di descrizione;
- ***n* byte**: eventuale stringa di descrizione.

La stringa è facoltativa e, come descritto in 4.1.1, può essere aggiunta da SUMO in casi particolari quali il fallimento di una determinata richiesta. La funzione `checkCommand` scorre la coda dei comandi presente nel client e per ogni comando invoca la funzione `checkPrefix`. Scopo di tale funzione è leggere il *prefisso* e comunicare all'utente eventuali discrepanze tra l'ID del comando a cui si sta rispondendo ed il corrispettivo presente nella coda dei comandi. Una volta terminata anche la fase di controllo delle risposte, l'esecuzione ritorna alla funzione `sendExact()` che, infine, libera la memoria occupata dalla coda dei comandi e dal buffer in uscita.

4.3.6 Composizione dei messaggi

Nel momento in cui il client invoca un determinato comando, TraCI deve preparare il messaggio da inviare al server secondo il formato espresso nelle Figure 4.1 e 4.2. A seconda della tipologia di comando, per quanto illustrato alla sezione 4.2, la composizione del messaggio dovrà essere differente. Nel caso di comandi *generali*, come ad esempio quelli realizzati dalle funzioni `simulationStep()` e `close()`, la composizione avviene internamente alle funzioni. Ciò è dovuto alla differente struttura caratterizzante questa tipologia di messaggi. Queste funzioni innanzitutto posizionano la richiesta in fondo alla coda dei comandi tramite una chiamata alla funzione `queue_push_msg(unsigned char command, struct msg_queue`

`*message_queue)`; successivamente le informazioni che compongono il contenuto del comando vengono posizionate nel buffer di uscita (`buffer_out`). Per quanto riguarda invece i comandi di *recupero* e di *modifica*, aventi entrambi un formato standard, è stata realizzata la funzione `beginMessage` caratterizzata dalla seguente signature:

```
void beginMessage(char cmdID, char varID, char *objID, int length)
```

dove `cmdID` rappresenta l'identificativo del comando che si intende impartire, `varID` rappresenta l'identificativo della variabile interessata dal comando (a scopo esemplificativo si veda 4.3.1), `objID` l'eventuale riferimento ad un oggetto componente la simulazione e `length` la lunghezza delle informazioni aggiuntive dipendenti dalla tipologia di comando. La prima operazione compiuta da `beginMessage` è l'accodamento tramite `queue_push_msg` , successivamente viene determinata la lunghezza totale del comando ed in base ad essa viene definito il formato da utilizzare secondo quanto espresso nelle Figure 4.1 e 4.2.

4.3.7 Realizzazione di un'applicazione

Per poter realizzare applicazioni che utilizzano l'implementazione sopra descritta è necessario includere il file `init.h` e gli header relativi ai moduli di comandi che si vogliono utilizzare (ad esempio, `simulation.h` e `vehicle.h`). Compito del client è inizializzare la connessione tramite la funzione `init()`; una volta stabilita la connessione è possibile impartire a SUMO qualsiasi comando implementato. Nel momento opportuno è necessario, ai fini dell'avanzamento della simulazione, la chiamata alla funzione `simulationStep()`. Tipicamente la struttura di un'applicazione client TraCI è caratterizzata dalla presenza di un ciclo dove ogni iterazione rappresenta uno step di simulazione. Infine compito del client è la terminazione della connessione tramite la funzione `close()`.

4.4 Valutazione sperimentale

La valutazione dell'implementazione di TraCI appena descritta si è basata sulla realizzazione di un'applicazione in grado di riprodurre le opzioni di rerouting a probabilità descritte al capitolo 3. I tempi di computazione registrati sono poi stati confrontati con i tempi ottenuti mediante la realizzazione di uno script equivalente che utilizza l'implementazione in Python di TraCI. Infine, un ulteriore confronto è stato effettuato tramite la realizzazione di simulazioni aventi l'opzione `adaptation-interval` pari a 0 (si veda 3.4.1). L'applicazione in TraCI/C ed il corrispettivo script TraCI/Python sono stati definiti riproducendo il comportamento dell'opzione di simulazione `--device.rerouting.probability` descritta in 3.1. In presenza di questa opzione il simulatore applica la probabilità di reinstradamento nel momento in cui un determinato veicolo prenda parte alla simulazione. Questo comportamento è stato riprodotto rispettando la stessa struttura algoritmica con il fine di valutare le prestazioni di esperimenti realizzanti le stesse dinamiche di simulazione.
I risultati sottoesposti sono stati raccolti in simulazioni effettuate sugli stessi scenari definiti al capitolo 3 con probabilità di reinstradamento incrementale al 10%. Il calcolatore utilizzato per i test è un comune notebook equipaggiato di processore Intel® Core™ i5-2450M da 2,50Ghz, 6GB RAM e sistema operativo Windows 7. Ogni tempo di computazione registrato è il risultato di 10 misurazioni differenti per cui vengono presentate media e varianza. Analizzando i risultati raccolti ed i grafici sotto esposti è evidente come l'implementazione di TraCI in ANSI C offra prestazioni generalmente migliori rispetto alla corrispettiva implementazione Python. È inoltre possibile constatare che, in condizioni di rerouting, l'interfaccia realizzata offra prestazioni migliori del simulatore stesso. In assenza di reinstradamento le performance di TraCI/C sono in linea con quelle del simulatore (grafici alle Figure 4.8 e 4.11). Il miglioramento delle prestazioni diviene dunque tangibile nel momento in cui il simulatore si trovi a gestire opzioni di reinstradamento; l'alleggerimento della computazione, dovuto all'impartizione da parte di Tra-

CI dei soli comandi necessari, riesce quindi a compensare il trasferimento dei dati da client e server.

Scenario Los Angeles

		SUMO		Python		ANSI C	
N. Veicoli	**% Rerouting**	E_t	σ_t^2	E_t	σ_t^2	E_t	σ_t^2
50	0	5,154	0,036	6,026	0,051	5,349	0,031
	10	5,381	0,038	6,253	0,058	5,512	0,034
	20	5,614	0,044	6,221	0,057	5,577	0,036
	30	5,632	0,054	6,291	0,056	5,664	0,035
	40	5,738	0,046	6,363	0,062	5,882	0,033
	50	5,929	0,052	6,527	0,064	5,969	0,039
	60	6,009	0,056	6,522	0,075	5,927	0,041
	70	6,105	0,054	6,751	0,071	5,875	0,049
	80	6,287	0,062	6,742	0,067	6,061	0,058
	90	6,352	0,060	7,017	0,072	6,249	0,061
	100	6,397	0,064	7,082	0,069	6,370	0,065
500	0	12,991	0,466	13,981	0,487	13,346	0,473
	10	14,163	0,478	14,840	0,512	14,290	0,481
	20	16,523	0,517	15,801	0,519	14,950	0,501
	30	16,792	0,528	16,826	0,542	16,130	0,511
	40	17,995	0,576	17,649	0,585	16,830	0,559
	50	19,790	0,583	18,336	0,629	17,587	0,547
	60	19,966	0,601	19,480	0,676	18,877	0,538
	70	20,874	0,592	20,430	0,654	19,744	0,569
	80	21,544	0,611	21,674	0,679	21,006	0,587
	90	23,870	0,603	22,446	0,658	21,549	0,604
	100	24,918	0,621	23,502	0,647	22,867	0,598
1000	0	24,860	1,704	26,861	1,798	25,819	1,746
	10	27,612	1,802	28,528	1,829	27,712	1,777
	20	30,627	1,787	30,803	1,923	29,548	1,812
	30	34,352	1,839	32,735	1,876	31,527	1,801
	40	36,955	1,874	34,712	1,891	33,278	1,796
	50	37,876	1,903	36,715	1,946	35,323	1,821
	60	40,337	1,857	38,388	1,978	37,404	1,867
	70	42,405	1,879	40,037	1,938	39,075	1,879
	80	44,422	1,934	41,966	1,987	41,379	1,903
	90	47,104	1,928	44,357	1,973	43,177	1,898

Continua nella prossima pagina

Tabella 4.2: continua dalla pagina precedente

		SUMO		**Python**		**ANSI C**	
N. Veicoli	**% Rerouting**	E_t	σ_t^2	E_t	σ_t^2	E_t	σ_t^2
	100	50,355	1,912	46,338	1,975	45,009	1,911
5000	0	72,818	3,109	75,220	3,312	70,694	3,211
	10	86,170	3,178	87,197	3,298	84,993	3,197
	20	97,727	3,165	97,623	3,357	93,183	3,193
	30	110,779	3,221	108,711	3,304	104,898	3,206
	40	123,588	3,256	117,797	3,442	114,576	3,210
	50	139,544	3,245	127,615	3,411	123,432	3,195
	60	148,020	3,295	134,271	3,487	129,587	3,229
	70	159,975	3,345	144,692	3,512	139,878	3,298
	80	169,059	3,484	155,238	3,301	149,568	3,304
	90	188,105	3,515	163,621	3,376	150,002	3,365
	100	195,299	3,526	175,833	3,458	165,930	3,378

Tabella 4.2: Media E_t e varianza σ_t^2 dei tempi di esecuzione (in secondi) registrati nello scenario di Los Angeles.

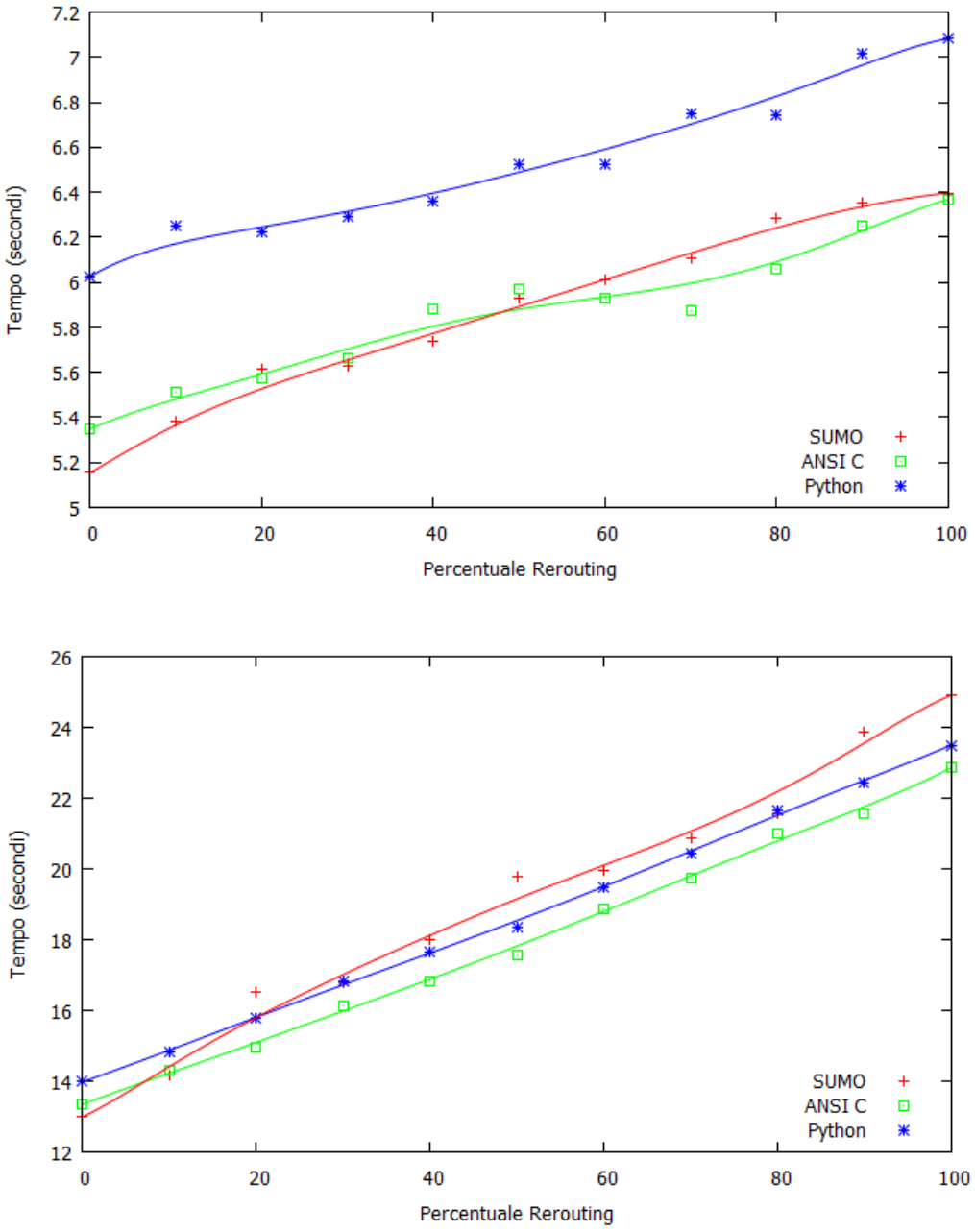

Figura 4.6: Grafici dei tempi di simulazione registrati nello scenario di Los Angeles con 50 e 500 veicoli partecipanti.

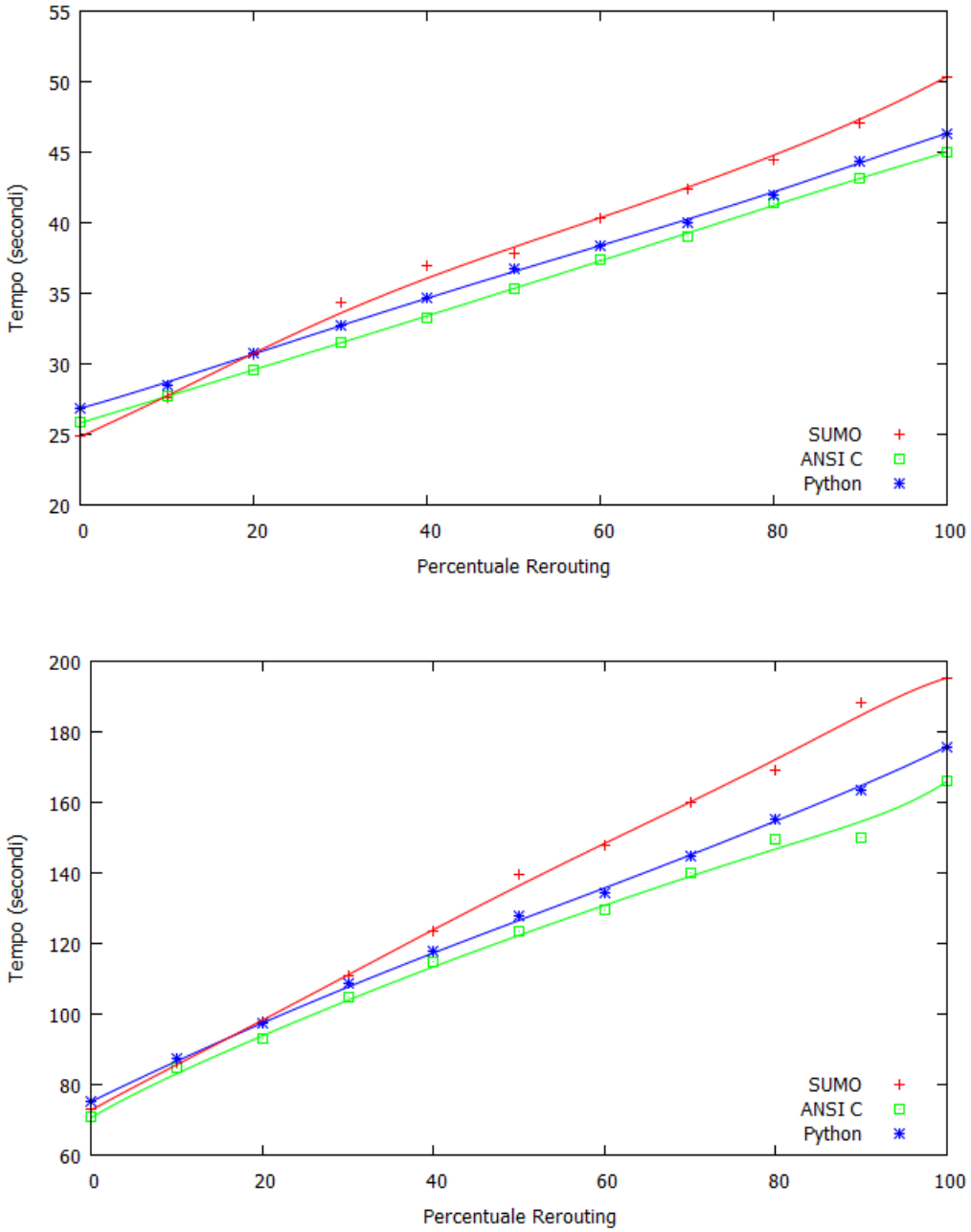

Figura 4.7: Grafici dei tempi di simulazione registrati nello scenario di Los Angeles con 1000 e 5000 veicoli partecipanti.

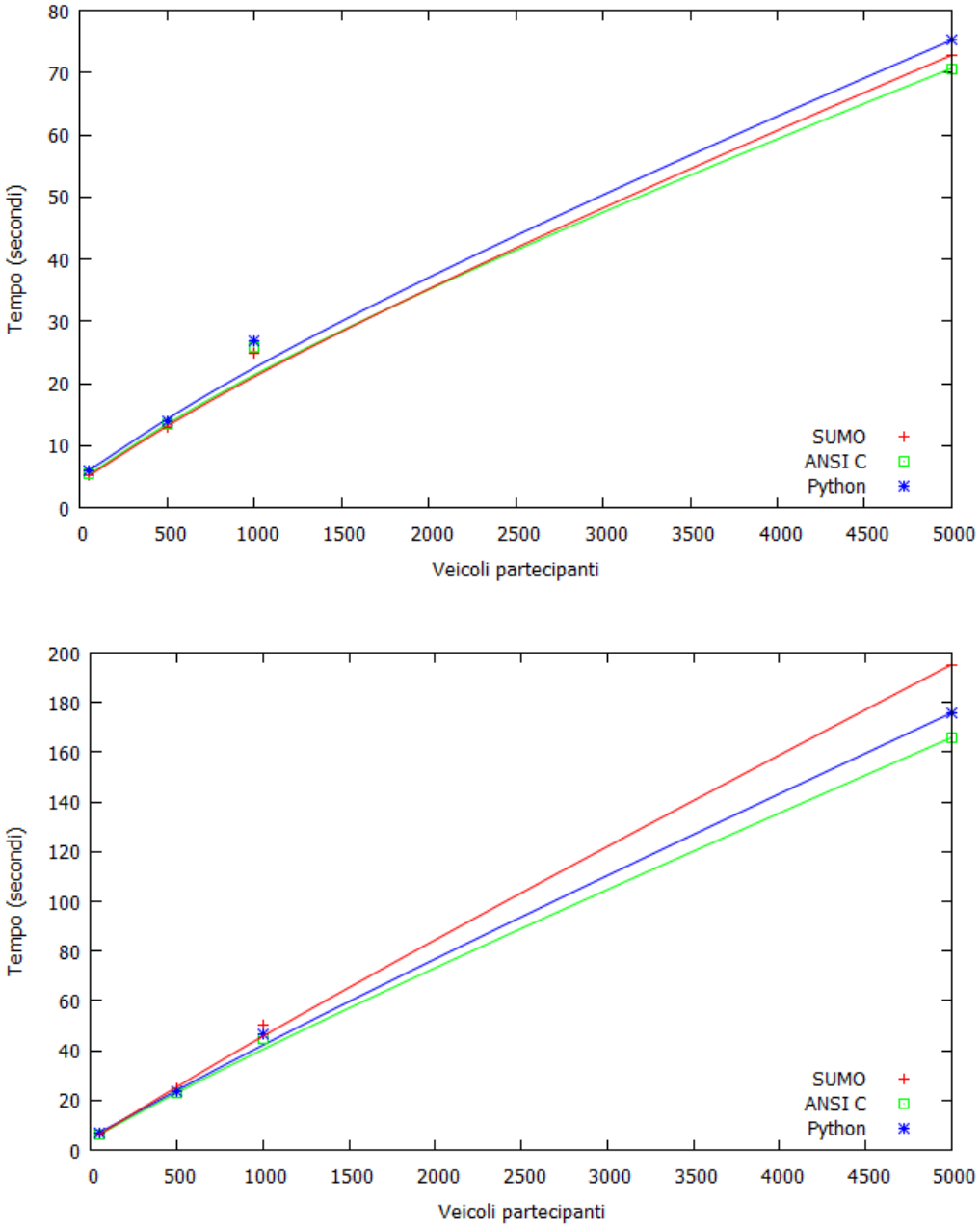

Figura 4.8: Grafici dei tempi nello scenario di Los Angeles con percentuale di reroute fissa allo 0% ed al 100% e numero di veicoli crescente.

Scenario New York

		SUMO		Python		ANSI C	
N. Veicoli	**% Rerouting**	E_t	σ_t^2	E_t	σ_t^2	E_t	σ_t^2
50	0	11,703	0,073	12,263	0,085	11,764	0,076
	10	12,237	0,074	12,488	0,082	11,915	0,071
	20	12,522	0,082	12,542	0,087	11,921	0,074
	30	12,557	0,075	12,606	0,089	11,992	0,079
	40	12,794	0,069	12,745	0,086	12,191	0,081
	50	13,038	0,073	12,895	0,085	12,428	0,073
	60	13,113	0,075	13,278	0,091	12,469	0,072
	70	13,216	0,083	13,299	0,088	12,577	0,080
	80	13,239	0,081	13,385	0,09	12,823	0,079
	90	13,407	0,082	13,44	0,093	12,962	0,082
	100	13,539	0,084	13,622	0,089	13,171	0,081
500	0	25,896	0,643	27,335	0,709	26,069	0,648
	10	27,465	0,665	27,555	0,724	26,911	0,641
	20	29,998	0,711	28,816	0,713	28,322	0,662
	30	32,786	0,697	30,428	0,758	29,505	0,698
	40	33,395	0,734	31,428	0,779	30,873	0,719
	50	36,61	0,792	32,349	0,743	31,937	0,712
	60	37,491	0,775	33,775	0,785	32,947	0,756
	70	38,842	0,762	34,986	0,809	34,091	0,768
	80	40,853	0,798	36,487	0,796	35,428	0,787
	90	42,279	0,723	37,671	0,823	37,136	0,772
	100	44,146	0,781	39,19	0,817	37,544	0,783
1000	0	38,412	1,545	42,271	2,104	41,231	1,988
	10	45,654	1,587	45,039	2,045	43,665	1,964
	20	48,12	1,589	48,253	1,991	45,939	1,942
	30	53,362	1,578	50,402	2,164	47,825	2,008
	40	59,167	1,601	53,083	2,111	51,098	2,067
	50	60,327	1,637	56,597	2,049	53,755	2,011
	60	63,862	1,609	57,789	2,193	56,321	2,114
	70	67,944	1,605	60,356	2,205	58,819	2,098
	80	71,286	1,654	63,477	2,249	61,711	2,125
	90	73,707	1,673	67,086	2,235	64,611	2,189

Continua nella prossima pagina

Tabella 4.3: continua dalla pagina precedente

		SUMO		**Python**		**ANSI C**	
N. Veicoli	**% Rerouting**	E_t	σ_t^2	E_t	σ_t^2	E_t	σ_t^2
	100	77,851	1,698	68,882	2,32	67,527	2,201
5000	0	154,263	4,236	158,989	4,404	153,000	4,014
	10	169,442	4,304	168,930	4,442	165,653	4,098
	20	183,007	4,434	179,095	4,439	173,692	4,176
	30	197,590	4,768	195,675	4,512	178,968	4,278
	40	215,492	4,645	204,321	4,853	199,873	4,465
	50	224,389	4,773	221,059	4,901	216,766	4,659
	60	247,052	4,839	239,679	4,878	229,453	4,601
	70	258,138	4,983	254,171	5,019	242,998	4,669
	80	276,027	5,019	264,477	5,114	257,086	4,785
	90	296,272	4,987	278,235	4,981	268,101	4,897
	100	306,243	5,103	296,726	5,201	276,154	4,953

Tabella 4.3: Media E_t e varianza σ_t^2 dei tempi di esecuzione (in secondi) registrati nello scenario di New York.

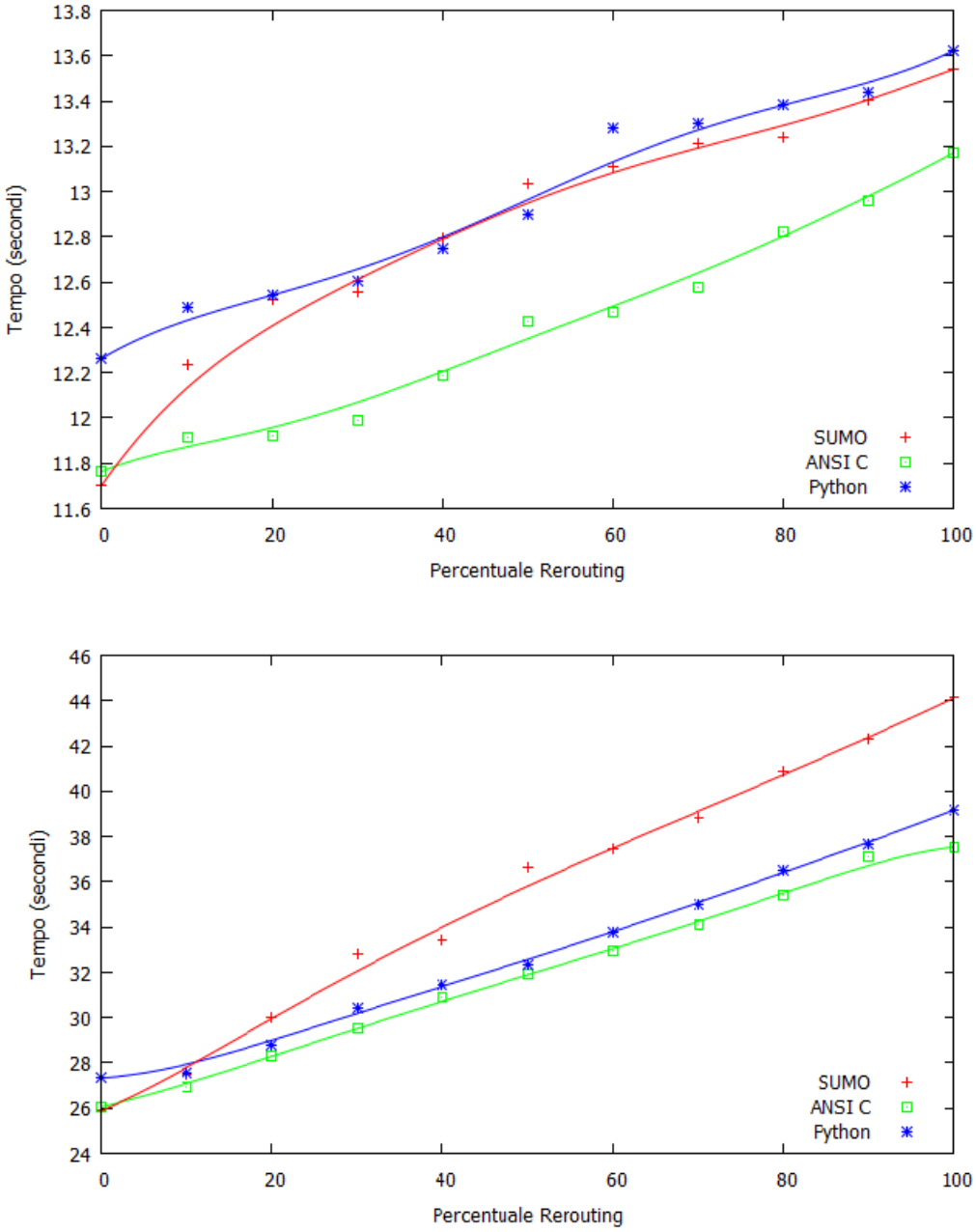

Figura 4.9: Grafici dei tempi di simulazione registrati nello scenario di New York con 50 e 500 veicoli partecipanti.

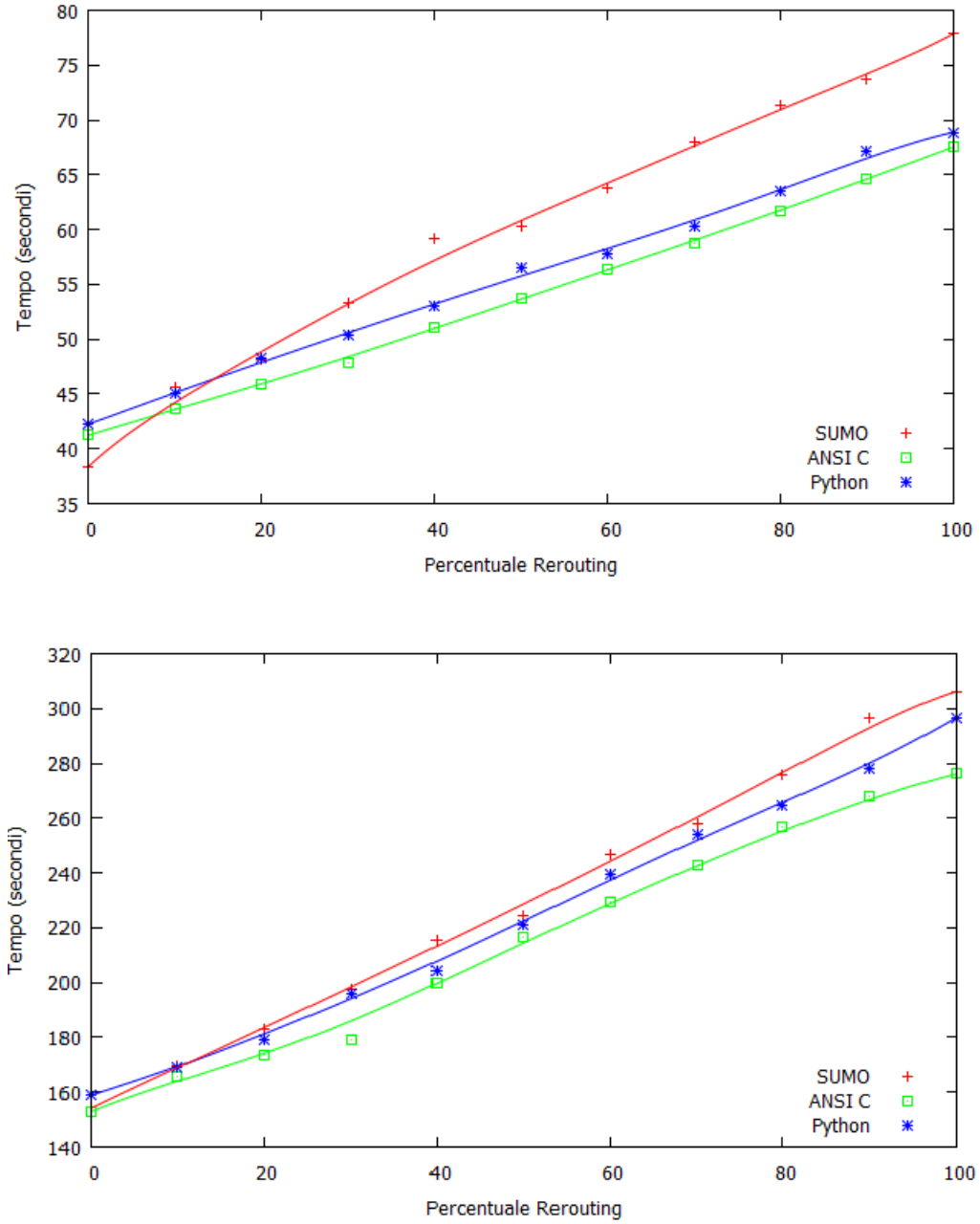

Figura 4.10: Grafici dei tempi di simulazione registrati nello scenario di New York con 1000 veicoli partecipanti.

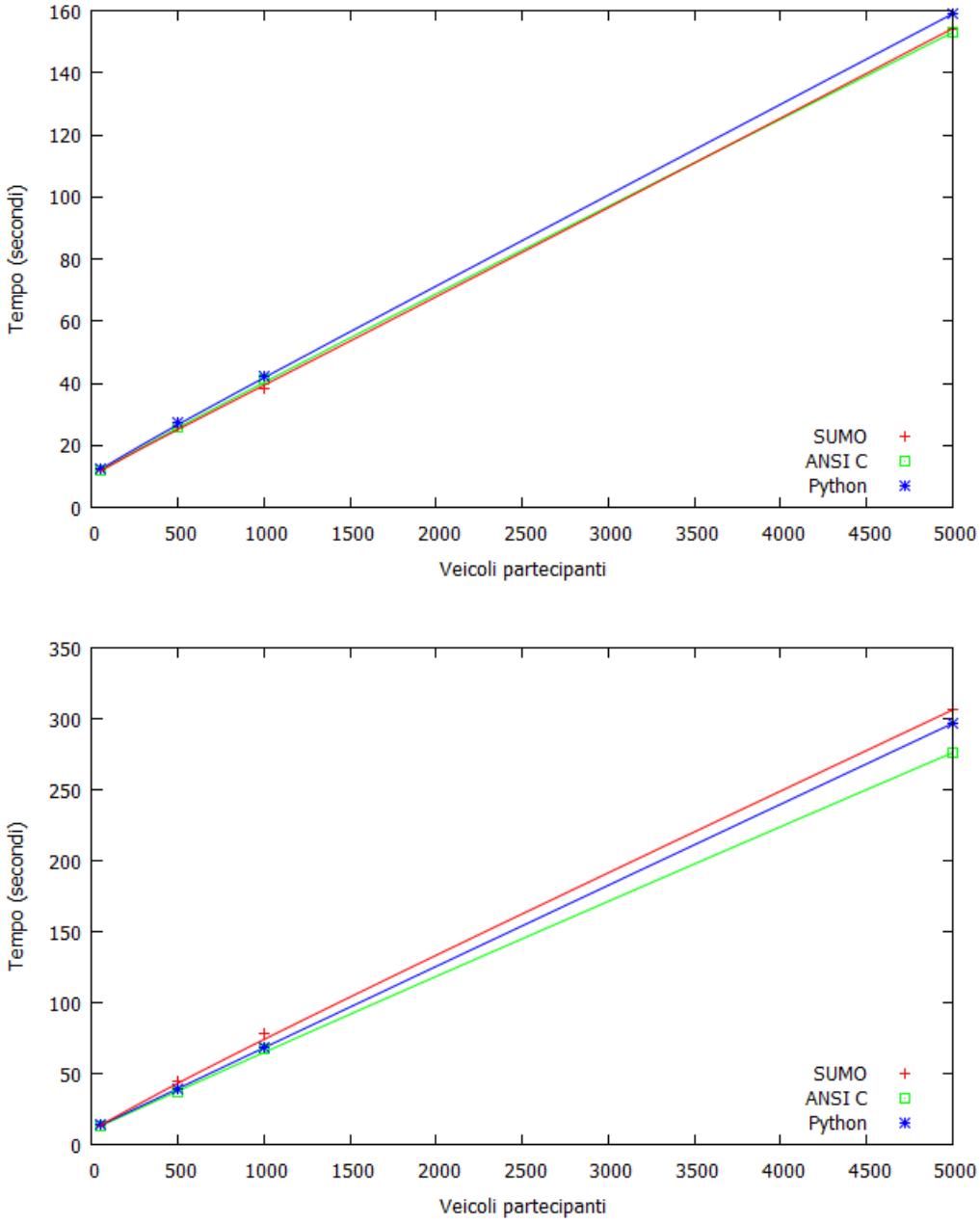

Figura 4.11: Grafici dei tempi nello scenario di New York con percentuale di reroute fissa allo 0% ed al 100% e numero di veicoli crescente.

Conclusioni

La mobilità veicolare è un elemento che contraddistingue la società moderna e la vita di tutti i giorni sotto molteplici aspetti: determina le nostre abitudini, caratterizza positivamente o negativamente le nostre città e, se opportunamente gestita, migliora la qualità di vita sia da un punto di vista della sicurezza stradale, sia da un punto di vista ambientale. Nonostante la sua evidente importanza, la mobilità veicolare non ha partecipato attivamente negli anni passati ad una vera e propria evoluzione informatica e tecnologica. Le reti veicolari ad-hoc rappresentano un significativo cambiamento nell'interpretazione della mobilità quotidiana. Questa complessa tipologia di rete porta con se una vera e propria rivoluzione sociale nell'interpretazione di un qualsiasi spostamento veicolare, sia esso l'abituale tragitto casa-lavoro oppure un lungo viaggio continentale. Affinchè tale tecnologia possa funzionare è necessario che i veicoli possano comunicare tra loro senza la necessità di un'infrastruttura preesistente. Ciò comporta una rivoluzione non solo sociale ma soprattutto tecnologica e scientifica. In questo contesto ogni veicolo deve essere opportunamente interfacciato per poter inviare e ricevere informazioni utili agli altri partecipanti alla rete. Questo implica quantomeno un ovvio adeguamento dei veicoli esistenti e delle politiche di produzione di nuovi veicoli. Si tratta certamente di un'operazione di vastissime proporzioni che potrà essere attuata solamente nel momento in cui lo sviluppo di applicazioni e protocolli per reti veicolari ad-hoc abbia raggiunto un livello adeguato e, soprattutto, consenta una comunicazione tra un veicolo e l'altro in tempi utili. I problemi principali derivano dal dinamismo intrinseco di queste reti e

dal relativo e costante cambiamento topologico. Questi fattori determinano rilevanti difficoltà nel routing dei pacchetti che attraversano la rete. Visti i costi e la difficoltà nel realizzare esperimenti sul campo, è necessario valutare alternative che consentano quantomeno un'approssimazione della realtà. In questo studio è stato dunque analizzato MoViT, strumento sviluppato ad UCLA (*Università della California, Los Angeles*) che simula la mobilità e virtualizza i veicoli componenti la rete. In questo modo è possibile sperimentare tutte le problematiche legate allo sviluppo di applicazioni e protocolli veicolari. Punto focale di questo studio è stata la componente inerente la simulazione all'interno di MoViT, ovvero il generatore di mobilità SUMO. SUMO è stato sottoposto a valutazione e, pur essendosi rivelato un ottimo simulatore, sono stati evidenziati e descritti alcuni difetti. Questa analisi ha portato alla definizione di uno strumento che potesse offrire un'alternativa al controllo della simulazione in fase di esecuzione e che potesse migliorare le prestazioni del simulatore, e la conseguente produzione di dati di mobilità per MoViT, in determinati contesti. Tale strumento potrà inoltre essere ampliato con l'inclusione di altri moduli previsti dal simulatore, tralasciati per la valutazione offerta in questo studio. Infine, con lo scopo di migliorare ulteriormente le prestazioni di SUMO e di TraCI, si potrebbero in futuro valutare alternative basate su API di memoria condivisa che sostituiscano il protocollo TCP attualmente utilizzato.

Bibliografia

[1] *ITU Statistics 2012: Key statistical highlights* - June 2012.

[2] James F. Kurose, Keith W. Ross. *Computer Networks: A Top-Down Approach Featuring the Internet* 2008 pp. 453-516.

[3] Imrich Chlamtac, Marco Conti, Jennifer J.-N. Liu. *Mobile ad hoc networking: imperatives and challenges* 2003.

[4] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. *The broadcast storm problem in a mobile ad hoc network.* In Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking (MobiCom '99). ACM, New York, NY, USA, 151-162.

[5] Theodore L. Willke, Patcharinee Tientrakool, Nicholas F. Maxemchuk. *A survey of inter-vehicle communication protocols and their applications.* 2009.

[6] F.K. Karnadi, Zhi Hai Mo, Kun-chan Lan, *Rapid Generation of Realistic Mobility Models for VANET* Wireless Communications and Networking Conference 2007, pp.2506-2511, 11-15 March 2007

[7] Kun-chan Lan, Chien-Ming Chou, *Realistic mobility models for Vehicular Ad hoc Network (VANET) simulations* ITS Telecommunications, 2008. pp.362-366, 24-24 Oct. 2008

[8] Bojin Liu, Khorashadi B, Haining Du, Ghosal D., Chen-nee Chuah, Zhang M. *VGSim: An integrated networking and microscopic vehicular mobility simulation platform* Communications Magazine, IEEE , vol.47, no.5, pp.134-141, May 2009

[9] Michel Ferreira, Hugo Conceição, Ricardo Fernandes, Ozan K. Tonguz *Stereoscopic aerial photography: an alternative to model-based urban mobility approaches.* In Proceedings of the sixth ACM international workshop on VehiculAr InterNETworking (VANET '09). ACM, New York, NY, USA, pp. 53-62

[10] J. Banks, J. Carson, B. Nelson, D. Nicol. *Discrete-Event System Simulation.*2003 pp. 3

[11] J. Haerri, F. Filali, C. Bonnet, Marco Fiore. *VanetMobiSim: generating realistic mobility patterns for VANETs.* In Proceedings of the 3rd international workshop on Vehicular ad hoc networks (VANET '06). ACM, New York, NY, USA, pp. 96-97

[12] Eugenio Giordano, Lara Codecà, Brian Geffon, Giulio Grassi, Giovanni Pau, and Mario Gerla. 2012. *MoViT: the mobile network virtualized testbed.* In Proceedings of the ninth ACM international workshop on Vehicular inter-networking, systems, and applications (VANET '12). ACM, New York, NY, USA, 3-12.

[13] Krajzewicz D., Hertkorn, G. Rössel C., Wagner, P. *Sumo (simulation of urban mobility).* In Proceedings of the 4th Middle East Symposium on Simulation and Modelling. pp. 183-187, 2002

[14] Rakotoarivelo T., Jourjon G., Ott M., Seskar I. *OMF: a control and management framework for networking testbed.* 2009

[15] Benvenuti C. *Understanding Linux network internals* O'Reilly Media, Inc, 2005.

[16] Karger D., Lehman E., Leighton T., Panigrahy R., Levine M., Lewin D.*Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web.* In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97). ACM, New York, NY, USA, pp. 654-663, 1997

[17] Wegener A., Piórkowski M., Raya M., Hellbrück H., Fischer S., Hubaux JP. *TraCI: An interface for Coupling Road Traffic and Network Simulators.* In Proceedings of the 11th communications and networking simulation symposium. pp. 155-163, 2008

[18] Rappaport T. *Wireless Communications* Prentice Hall PTR Upper Saddle River, NJ, 2002

[19] Giordano E., Frank R., Pau G., Gerla M. *CORNER: a Step Towards Realistic Simulations for VANET.* In Proceeding of the 7th ACM International Workshop on Vehicular Inter-NETworking (VANET 2010), 2010

[20] Bianchi G. *Performance analysis of the IEEE 802.11 distributed coordination function.* IEEE Journal on selected areas in communications.vol.18, no.3 pp. 535-547, 2000

[21] Zhou J., Ji Z., Bagrodia R. *TWINE: A hybrid emulation testbed for wireless networks and applications*, 2006

[22] Zhou J., Ji Z., Varshney M., Xu Z., Yang Y., Marina M., Bagrodia R. *WHYNET: A Hybrid Testbed for Large-Scale, Heterogeneous and Adaptive Wireless Networks.* In proceedings of the 1st international workshop on Wireless network testbeds, experimental evaluation & characterization. pp. 111-112, 2006

[23] Beuran R., Nakata J., Okada T., Lan Tien Nguyen, Yasuo Tan, Shinoda Y. *A Multi-Purpose Wireless Network Emulator: QOMET.* Advanced

Information Networking and Applications - Workshops, 2008. pp.223-228, 2008

[24] Hibler M., Ricci R., Stoller L., Duerig J., Guruprasad S., Stack T., Webb K., Lepreau,J. *Large-scale virtualization in the Emulab network testbed.* USENIX 2008 Annual Technical Conference on Annual Technical Conference, pp. 113-128, 2008

[25] Macker J.P., Chao W., Weston J.W. *A low-cost, IP-based mobile network emulator (MNE)* MILCOM '03. 2003 IEEE , vol.1, no., pp. 481-486, 2003

[26] Behrisch M., Bieker L., Erdmann J., Krajzewicz D. *SUMO - Simulation of Urban MObility. An Overview.* SIMUL 2011: The Third International Conference on Advances in System Simulation. pp. 55-60, 2011

[27] Krauß S. *Microscopic Modelling of Traffic Flow: Investigation of Collision Free Vehicle Dynamics.* DLR (Hauptabteilung Mobilität und Systemtechnik), 1998

[28] . Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., Stein C. *Introduction to Algorithms (Second ed.)* Section 24.3: Dijkstra's algorithm. pp. 595–601, 2001

[WIKI] `http://it.wikipedia.org/wiki/Simulazione`
[OMF] `http://omf.mytestbed.net`
[NTP] `http://www.ntp.org/`
[SUMO] `http://sumo.sourceforge.net/`
[OSM] `http://www.openstreetmap.org/`
[VISUM] `http://www.ptvamerica.com/software/ptv-vision/visum/`
[VISSIM] `http://www.ptvamerica.com/software/ptv-vision/vissim/`
[OPENDRIVE] `http://www.opendrive.org/index.htm`
[MATSIM] `http://www.matsim.org/`
[ARCVIEW] `http://www.esri.com/software/arcgis/arcview`
[JOSM] `http://josm.openstreetmap.de/`

[GEOFABRIK] `http://www.geofabrik.de/`

[VS2010] `http://www.microsoft.com/visualstudio/`

NOTA: tutte le citazioni da testi in lingua straniera sono state effettuate traducendo o riassumendo nel modo semanticamente più fedele possibile.