Università di Pisa

FACOLTÀ DI INGEGNERIA
Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea Specialistica

Generazione software di traffico Ethernet ad alta velocità: Ostinato incontra netmap

Relatori
Prof. Luigi Rizzo
Ing. Giuseppe Lettieri

Candidato
Giorgio Buffa

Anno Accademico 2012–2013

Sommario

In questa tesi viene affrontato il problema della generazione *software* di traffico su reti Ethernet ad alta velocità (10 Gbps). In particolare, si estenderà uno specifico programma, chiamato “Ostinato” [27], per utilizzare il *framework* “netmap” [19; 20], con l'obiettivo di aumentare il numero massimo di pacchetti inviabili al secondo e di migliorare l'accuratezza di tale processo trasmissivo.

Si fornirà inoltre una valutazione sperimentale delle proprietà metrologiche dell'applicativo (numero massimo di pacchetti trasmissibili al secondo, accuratezza nella generazione sintetica del traffico, ecc.), non reperibile altresì in letteratura.

L'analisi metrologica svolta inizialmente (sulla versione ufficiale di Ostinato) ha evidenziato una scarsa accuratezza, specie su sistemi operativi *non* Linux. Nell'introdurre il supporto nativo a netmap, quindi, alcune parti parti fondamentali del programma sono state riviste, compreso il motore di trasmissione dei pacchetti.

Dopo la ristrutturazione del codice, i dati sperimentali collezionati hanno mostrato la capacità del generatore di produrre pacchetti con perfetta precisione e accuratezza fino al punto di saturazione della piattaforma (il cui valore dipende ovviamente dalle caratteristiche dell'elaboratore, dal sottosistema di rete adottato e dalle dimensioni dei pacchetti Ethernet trasmessi).

Con netmap, Ostinato risulta in grado di produrre 10 Gbps di traffico (e oltre), anche nel caso peggiore di pacchetti Ethernet di dimensione minima, proponendosi come strumento indispensabile per realizzare esperimenti e misurazioni su reti ad alta velocità, in quanto unico nella sua capacità di coniugare prestazioni elevate, precisione, accuratezza, flessibilità e semplicità di utilizzo.

Parole chiave: Ostinato, netmap, generatore di traffico, 10 Gbit Ethernet, *line rate*, *commodity hardware*.

Indice

Sommario iii

Indice v

Elenco delle figure vii

Elenco delle tabelle ix

Elenco dei listati xi

1 Introduzione **1**
- 1.1 I generatori di traffico . . . 1
- 1.2 L'eterogeneo mondo dei generatori di traffico *software* . . . 3
  - 1.2.1 Alcuni esempi significativi . . . 4
- 1.3 Generazione *software* di traffico su reti ad alta velocità . . . 8
  - 1.3.1 Il ruolo di netmap . . . 9
- 1.4 Panoramica sullo stato dell'arte . . . 10
  - 1.4.1 Adozione di Ostinato nell'industria e nella ricerca . . . 11
- 1.5 Obiettivi e organizzazione di questa tesi . . . 12

2 Ostinato **15**
- 2.1 Caratteristiche e funzionalità . . . 15
  - 2.1.1 Caso d'uso: generazione di un flusso UDP costante . . . 16
- 2.2 Architettura *software* di Ostinato . . . 22
  - 2.2.1 Le modalità di interazione fra *client* e *server* . . . 23
  - 2.2.2 La struttura interna del *server* . . . 27
  - 2.2.3 La gestione delle porte sul *server* . . . 27
  - 2.2.4 Il modello definito da `AbstractPort` per la lista dei pacchetti da trasmettere . . . 29
  - 2.2.5 La trasmissione dei pacchetti secondo `PcapPort` . . . 34
  - 2.2.6 La gestione delle statistiche . . . 35

3 Adattamento con netmap **37**
- 3.1 Motivazioni e considerazioni progettuali . . . 37
- 3.2 Impatto a livello d'architettura . . . 39
- 3.3 Gestione delle marcature temporali . . . 40
- 3.4 La lista dei pacchetti da trasmettere . . . 41

3.5 L'algoritmo di trasmissione dei pacchetti 44
3.6 Controllo esclusivo della porta 47
3.7 Mettendo tutto insieme 47
3.8 Revisione della classe `PcapPort` 48

4 Valutazione delle prestazioni 51
4.1 Metodologia 51
4.1.1 Scenari e configurazioni 52
4.1.2 Variabili e metriche 54
4.1.3 Registrazione delle statistiche 54
4.2 Implementazione ufficiale di Ostinato 55
4.3 Dopo la revisione di strutture dati e algoritmi 59
4.4 Utilizzando netmap 61

5 Conclusioni e sviluppi futuri 65

Bibliografia 67

Elenco delle figure

2.1 La finestra principale di Ostinato (componente *client*) 17
2.2 Esempio flusso UDP, lo scenario per il caso d'uso 17
2.3 Esempio flusso UDP, il nuovo *stream* per la porta selezionata . . . 18
2.4 Esempio flusso UDP, scheda `Protocol Selection` 19
2.5 Esempio flusso UDP, scheda `Protocol Data` 20
2.6 Esempio flusso UDP, scheda `Stream Control` 21
2.7 Esempio flusso UDP, scheda `Packet View` 22
2.8 Esempio flusso UDP, trasmissione in corso 23
2.9 Il meccanismo di comunicazione fra i componenti di Ostinato . . . 24
2.10 La struttura interna di `drone` 27
2.11 La gerarchia di classi per la gestione delle porte in `drone` 28
2.12 Esempio modello lista, configurazione di esempio 32
2.13 Esempio modello lista, decomposizione del profilo atteso 33
2.14 Esempio modello lista, rappresentazione secondo `AbstractPort` . . 33
2.15 Esempio modello lista, rappresentazione secondo `PcapPort` 34

3.1 La nuova classe `NetmapPort` nella gerarchia di `AbstractPort` . . . 39
3.2 Esempio modello lista, rappresentazione secondo `PacketList` . . . 42
3.3 La struttura interna di `NetmapPort` 47

4.1 Configurazione di base per gli esperimenti 52
4.2 Ostinato-libpcap, profilo pacchetti trasmessi per `fx8350` (Linux, Windows, FreeBSD) 56
4.3 Ostinato-libpcap, profilo pacchetti trasmessi per `mbp9,2` (OS X) . . 57
4.4 Ostinato-libpcap, profilo pacchetti trasmessi per `h102` (Linux) . . . 58
4.5 Ostinato-libpcap (rev.), profilo pacchetti trasmessi per `fx8350` (Linux, FreeBSD) 60
4.6 Ostinato-libpcap (rev.), profilo pacchetti trasmessi per `mbp9,2` (OS X) 60
4.7 Ostinato-libpcap (rev.), profilo pacchetti trasmessi per `h102` (Linux) 61
4.8 Ostinato-netmap, capacità trasmissiva al variare della dimensione massima dei *burst* per `h102` (Linux) 62
4.9 Ostinato-netmap, profilo pacchetti trasmessi per `h102` (Linux) . . . 62
4.10 Ostinato-netmap, distribuzione delle dimensioni dei *burst* al variare della velocità trasmissiva per `h102` (Linux) 63

Elenco delle tabelle

1.1 Confronto fra alcuni generatori di traffico stocastici (*open-source*) . 5

4.1 Caratteristiche degli elaboratori impiegati per gli esperimenti . . . 53
4.2 Ostinato-libpcap, confronto fra i vari sistemi (max pps generati) . . 57
4.3 Ostinato-libpcap (rev.), confronto fra i vari sistemi (max pps generati) 59

Elenco dei listati

2.1 L'interfaccia `OstService` 24
2.2 I Protocol Buffers che descrivono una porta 25
2.3 I Protocol Buffers che descrivono statistiche e stato d'una porta . . 26
2.4 L'interfaccia per popolare la lista dei pacchetti sulla porta 30

3.1 L'interfaccia per popolare la `PacketList` 42
3.2 L'interfaccia per iterare sulla `PacketList` 43
3.3 L'algoritmo di trasmissione dei pacchetti usato da `NetmapPort` . . 44
3.4 Il nuovo algoritmo di trasmissione dei pacchetti per `PcapPort` . . . 48

4.1 Estensione dell'interfaccia `OstService` per il *logging* 55
4.2 L'interfaccia per il *logging* esposta dalla classe `AbstractPort` . . . 55

Capitolo 1

Introduzione

Nell'ambito delle reti di calcolatori, il comportamento e le prestazioni dei protocolli di comunicazione, delle applicazioni e (in parte) dei dispositivi stessi vengono studiati secondo approcci che possono essere classificati come: analitici, simulativi o sperimentali.

Fra questi, l'approccio sperimentale, basato su misurazioni attive o passive, sta diventando sempre più comune e utilizzato. Il motivo è principalmente legato alla crescente complessità delle reti, che si sviluppa lungo varie direttrici (topologie, tecnologie esistenti, volumi di traffico, applicazioni supportate, ecc.), rendendo sempre più difficile la loro comprensione e modellazione. Lo stesso metodo basato sulla riproduzione di registrazioni di traffico reale, oltre a essere soggetto alle leggi sulla protezione dei dati personali, risulta piuttosto limitato quando è necessario riprodurre scenari particolari (ad esempio: attacchi, oppure condizioni critiche o poco comuni).

In questa introduzione si parlerà dei generatori di traffico: cosa sono, a cosa servono e perché sono utili. La discussione sarà incentrata sui generatori *software*, illustrandone l'importanza e presentando le difficoltà legate alla loro realizzazione, con particolare riferimento al loro utilizzo su reti ad alta velocità. Verrà inoltre fornita una panoramica sullo stato dell'arte in tale settore, concludendo con l'esposizione degli obiettivi e dell'organizzazione di questo lavoro.

1.1 I generatori di traffico

Sulle moderne reti di calcolatori i messaggi scambiati fra due applicazioni in comunicazione sono trasportati in forma pacchettizzata: l'informazione da trasmettere viene suddivisa in pacchetti di dimensioni variabili, che sono inviati individualmente attraverso la rete. Per ognuno di questi pacchetti, il contenuto informativo vero e proprio è preceduto da una intestazione, che contiene tutta l'informazione necessaria affinché il pacchetto stesso possa essere identificato e inoltrato fino alla sua destinazione finale. La struttura di ogni pacchetto dipende, oltre che dalle specifiche applicazioni, anche dalle tecnologie e dai protocolli di rete adottati per realizzare tale comunicazione [7].

All'interno di questo scenario, i generatori di traffico agiscono come strumenti di misura: realizzati mediante piattaforme *hardware* o *software*, sono utilizzati per produrre e trasmettere pacchetti sulla rete in maniera controllata, ovvero rispettando le caratteristiche e il profilo definiti dall'utilizzatore per tale traffico: protocolli di rete impiegati, valori dei campi di controllo, contenuto informativo, numero di pacchetti inviati al secondo, come varia nel tempo la dimensione dei pacchetti e la velocità trasmissiva, ecc.

I flussi di traffico da generare e le loro caratteristiche possono essere descritti con modalità differenti: specificando un modello stocastico, fornendo una registrazione di traffico reale da riprodurre, oppure definendo liste di istruzioni da eseguire [34].

Alcuni di questi strumenti si occupano della sola generazione di traffico, altri (usati in coppia) sono in grado di effettuare misurazioni attive sul funzionamento della rete, registrando, ad esempio, la banda disponibile lungo il collegamento utilizzato, la probabilità di perdita dei pacchetti, l'ammontare del ritardo sperimentato dagli stessi o come esso varia (*jitter*).

Poter disporre di simili strumenti per la generazione sintetica del traffico risulta fondamentale in vari settori. Essi sono infatti usati nella pratica quotidiana da ricercatori, ingegneri e tecnici di rete, per i compiti più disparati: misurare le prestazioni o le caratteristiche dei sistemi (siano essi collegamenti, apparati, protocolli, sistemi operativi o programmi); verificarne il funzionamento sotto sforzo; validare le implementazioni (ad esempio, la capacità di riconoscimento e l'efficacia di *firewall*, IDS[1] e IPS[2]); valutare la bontà di nuove proposte che riguardano architetture e protocolli di rete, o dei meccanismi per garantire la qualità del servizio; per l'identificazione e la risoluzione di problemi; ecc. [35]

Requisiti importanti per un generatore di traffico sono la precisione e l'accuratezza: la *precisione* è correlata alla qualità e alla stabilità del sistema e permette di ottenere valori o misurazioni successive simili, se non uguali; l'*accuratezza*, d'altra parte, determina di quanto si discosta il valore prodotto da quello richiesto [16].

Nel trattare i generatori di traffico, il primo aspetto da prendere in considerazione riguarda la modalità di realizzazione (*hardware* o *software*), in quanto essa ha molteplici ripercussioni su costi, prestazioni e funzionalità che ne derivano.

I generatori *hardware* sono sistemi dedicati alla singola applicazione, che si basano spesso su componenti *hardware* specifici (come schede FPGA, o processori di rete) per raggiungere le prestazioni richieste entro determinati intervalli di confidenza sui valori imposti (numero di pacchetti generati al secondo, varietà di traffico, ecc.).

I generatori *software*, di contro, sono programmi eseguiti su un calcolatore *general purpose*, in spazio utente (come una qualsiasi applicazione), o a livello di nucleo di sistema operativo (il cosiddetto *kernel*).

Se, da un lato, i generatori *hardware* sono precisi, accurati e capaci di raggiungere prestazioni elevate (ovvero, numero di pacchetti generati al secondo), dall'altro si rivelano soluzioni anche molto costose, specie quando supportano

[1] *Intrusion Detection System*; in italiano, sistema di rilevamento delle intrusioni.

[2] *Intrusion Prevention System*; in italiano, sistema di prevenzione delle intrusioni.

protocolli o tecnologie di rete di recente concezione, e questo rappresenta spesso una barriera significativa per la loro adozione. Inoltre, offrono un ventaglio di opzioni piuttosto limitato in termini di funzionalità, protocolli supportati ed estensibilità (la possibilità di introdurre il supporto per nuovi protocolli o nuove funzioni).

Nella pratica, quindi, si utilizzano in prevalenza generatori *software* (che sono in genere sviluppati da università, centri di ricerca, o gruppi di lavoro autonomi per rispondere a esigenze più o meno specifiche): sono soluzioni economiche, spesso gratuite, se non addirittura aperte (*open-source*, liberamente modificabili). Il loro funzionamento dipende però dalla potenza del calcolatore sottostante, dal sistema operativo adottato e dal carico di lavoro e di traffico a cui il sistema stesso è sottoposto [16], risultando spesso meno accurati e prestanti rispetto alla loro controparte *hardware*.

Difatti, nell'ambito della ricerca sulle reti, la maggior parte delle volte vengono impiegate piattaforme *software* non tanto per la loro economicità, quanto soprattutto per la loro flessibilità potendo, ad esempio, dispiegare un numero elevato di nodi generatori di traffico, per riprodurre scenari distribuiti; modificare il codice sorgente per scopi di ricerca specifici; effettuare esperimenti operando su e con implementazioni reali di sistemi operativi e protocolli di rete [2].

1.2 L'eterogeneo mondo dei generatori di traffico *software*

È difficile fornire una tassonomia dei generatori *software* poiché, nel corso degli anni, ne sono stati sviluppati davvero tanti. Ognuno offre un insieme specifico di funzionalità ed è più o meno adatto per studiare determinati problemi; inoltre, considerato che molti di essi utilizzano approcci differenti per uno stesso scopo, caratteristiche e prestazioni possono variare sensibilmente. I criteri per la loro organizzazione, quindi, sono molteplici.

Per fare degli esempi, esistono generatori *software* che lavorano in spazio utente, altri come moduli del sistema operativo (*kernel*) e altri ancora che si basano su *hardware* specializzato; certi generatori richiedono privilegi amministrativi, mentre altri no; la trasmissione del traffico può avvenire usando l'interfaccia *socket*, utilizzando librerie e meccanismi alternativi, o forgiando direttamente i pacchetti da trasmettere; ecc.

Botta et al. [2] fornisce una classificazione basata sul livello di astrazione a cui questi lavorano, distinguendo fra le seguenti categorie.

- **Generatori di livello applicativo**: riproducono il profilo di traffico caratteristico di una determinata applicazione di rete.

- **Generatori di flusso**: si limitano a replicare le caratteristiche di determinati flussi di traffico (ad esempio, la loro durata, o il numero di pacchetti e byte trasferiti).

- **Generatori di pacchetti**: consentono di specificare la dimensione di ogni pacchetto inviato e il tempo che intercorre fra ognuno di essi, in accordo alle distribuzioni statistiche scelte per queste due variabili. Si tratta della categoria più popolata.
- **Generatori multilivello**: tengono in considerazione le interazioni fra i molteplici livelli nella pila dei protocolli (utenti, sessioni, connessioni, caratteristiche della rete). Sono usati solo raramente per condurre ricerche sperimentali, essendo complessi e spesso non di pubblico dominio.

Un criterio ortogonale, basato su considerazioni di tipo sistemistico, è descritto in Schroeder et al. [25].

- **Generatori ad anello aperto**: il loro funzionamento è indipendente dalle osservazioni sullo stato della rete rilevate durante il processo di trasmissione.
- **Generatori ad anello chiuso**: il loro funzionamento cambia dinamicamente sulla base di queste osservazioni, le quali determinano la variazione dei parametri che descrivono il traffico da generare. I cambiamenti possono riguardare la distribuzione statistica delle dimensioni dei pacchetti e dei tempi fra un pacchetto e il successivo, o il loro contenuto.
- **Generatori ad anello parzialmente aperto**: sono sistemi ibridi rispetto ai precedenti due.

1.2.1 Alcuni esempi significativi

Sul Web esistono diverse liste (più o meno aggiornate) che elencano i vari generatori di traffico *software* esistenti, come ad esempio [4; 26; 33]. Un elenco abbastanza completo, aggiornato a Luglio 2012, è reperibile in Papadopoulos [15]; tale opera presenta inoltre un'istantanea, in termini quantitativi (considerando le citazioni in articoli e brevetti), dei generatori più utilizzati nell'ambito della ricerca e l'andamento della loro popolarità nel tempo.

In questa sottosezione saranno presentati alcuni generatori *software*, scelti in base alla loro popolarità o perché particolarmente rappresentativi di una determinata categoria.

Iperf

Iperf[3] è sicuramente il generatore di traffico più conosciuto e utilizzato. È stato inizialmente sviluppato dal gruppo di lavoro "Distributed Applications Support Team" (DAST) presso il "National Laboratory for Applied Network Research" (NLANR) per stimare la massima velocità di trasmissione raggiungibile su un collegamento utilizzando i protocolli TCP e UDP.

[3] Iperf, http://iperf.fr

Tabella 1.1: Confronto fra alcuni generatori di traffico stocastici (*open-source*)

	iperf	**D-ITG**	**pkt-gen**	**Ostinato**
Primo rilascio	Feb/2000	Set/2003	Nov/2011	Giu/2010
Versione corrente	3.0.1 (Gen/2014)	2.8.1 (Lug/2013)	N/A (Gen/2014)	0.5.1 (Lug/2012)
Ambito	Misurazione banda	Generazione stocastica	Generazione alta velocità	Generazione di pacchetti
Sistema operativo	Linux, FreeBSD, Mac OS X[1]	Linux, FreeBSD, Mac OS X, Windows, Montavista Linux, OpenWRT, SnapGear	Linux, FreeBSD	Linux, FreeBSD, Mac OS X, Windows
Livello	Spazio utente	Spazio utente	Spazio utente	Spazio utente
Privilegi	Utente	Utente	Amministratore	Amministratore
Interfaccia	Console, GUI[2]	Console, GUI[2]	Console	GUI
Protocolli di link	✗	✗	✗	Ethernet II, 802.3, 802.3 LLC, 802.3 SNAP, 802.1Q, 802.1ad
Protocolli di rete	IPv4, IPv6	IPv4, IPv6	IPv4	IPv4, IPv6, 6over4, 4over6, 4over4, 6over6, ARP
Proto. di trasporto	TCP, UDP	TCP, UDP, ICMPv4, ICMPv6, SCTP, DCCP	UDP	TCP, UDP, ICMP, IGMP, MLD
Proto. applicativi	✗	DNS, Telnet, VoIP (più codec), Counter Strike, Quake 3	✗	✗
Multicast	✓	✗	✗	✗
Broadcast	✗	✗	✗	✗
Profili di traffico	Costante	Costante, Uniforme, Esponenziale, Pareto, Cauchy, Normale, Poisson, Gamma, Weibull, On/Off, Traccia `pcap`	Costante	Costante, Traccia `pcap`
Metriche misurate	Throughput, jitter, perdite, one-way delay, ritrasmissioni TCP, utilizzazione CPU	Throughput, jitter, perdite, one-way delay e round-trip time	Pacchetti, Pps	Pacchetti, Pps, Byte, Bps trasmessi e ricevuti; errori in ricezione
Qualsiasi ricevitore	✗	✗	✓	✓
Distribuito	✓	✓	✗	✓
Log file	✗	✓ (binario)	✗	✗
Sviluppo	`C++`, `socket`	`C`, `socket`	`C`, `netmap`	`C++`, `libpcap`
Altre note	`iperf3` non compatibile con v2	—	Può tramsmettere verso più indirizzi MAC e IPv4	Protocolli configurabili a piacimento

[1] `iperf2` supporta anche Windows.
[2] GUI sviluppata da terze parti.

Iperf è basato sul modello *client/server* ed è progettato per lavorare sia con IPv4, sia con IPv6. Multipiattaforma e semplice da usare, permette di regolare vari parametri dei protocolli TCP/UDP, del processo di trasmissione (consente anche trasmissioni multiple in parallelo) e fornisce un resoconto su varie metriche: banda disponibile lungo il collegamento, quantità di dati trasferiti, ritardi, *jitter* e percentuali di perdita dei pacchetti.

Questo strumento è basato su linea di comando, ma esiste anche una interfaccia grafica, realizzata in Java e chiamata Jperf, che fa da intermediario con tale interfaccia a linea di comando.

La versione 3 è una implementazione completamente nuova di iperf, sempre *open-source* ma non retro-compatibile con `iperf2`, realizzata da zero con l'obiettivo di avere una base di codice più piccola e semplice, le cui funzionalità possono essere rese disponibili anche sotto forma di libreria (utilizzabili quindi in altri programmi). `Iperf3` introduce inoltre un certo numero di caratteristiche, disponibili in altri strumenti come `nuttcp` e `netperf`, ma che non sono presenti in `iperf2`.

D-ITG

Un generatore stocastico particolarmente versatile e interessante, nonché discretamente impiegato, è D-ITG[4], sviluppato dal gruppo di ricerca "Traffic" (parte dell'unità "COMICS"), presso l'Università "Federico II" di Napoli.

D-ITG è uno strumento *open-source*, distribuito e multipiattaforma, basato sul modello *client/server*.

Supporta la generazione di traffico su reti IPv4/IPv6 (permette anche di impostare i valori per i campi `TOS` e `TTL`) ed è capace di generare traffico a livello di rete, a livello di trasporto e a livello applicativo, replicando i processi stocastici per le dimensioni dei pacchetti e le distanze temporali fra pacchetti successivi (*inter-departure time*) secondo una varietà di modelli statistici selezionabili dall'utente.

Le metriche misurabili con D-ITG sono: ritardo (*one-way delay* e *round-trip time*), *jitter*, percentuali di perdita, *throughput*.

Lo strumento è basato su linea di comando, ma esiste anche una interfaccia grafica "non ufficiale", che interagisce direttamente con l'interfaccia a riga di comando.

D-ITG si compone di quattro elementi che possono operare in maniera distribuita (di seguito descritti), ognuno dei quali realizza una specifica funzione.

`ITGSend` agisce come *client* ed è il componente trasmettitore del traffico; può generare un singolo flusso di traffico, o più flussi di traffico contemporaneamente (in quest'ultimo caso le loro caratteristiche vanno specificate mediante un *file* di configurazione). `ITGRecv` invece agisce da *server*, ricevendo i flussi di traffico dai vari trasmettitori. Entrambe queste entità generano dei messaggi di *log*, che possono essere memorizzati localmente o remotamente su file (in formato binario) usando `ITGLog`, il componente che funge appunto da *server* per i *log*. Una volta

[4]Distributed Internet Traffic Generator (D-ITG), http://traffic.comics.unina.it/software/ITG/

concluso l'esperimento, è possibile decodificare e analizzare i risultati ottenuti utilizzando `ITGDec`.

`pkt-gen`

`pkt-gen`[5] è un generatore semplice ma estremamente performante, forse ancora poco conosciuto in quanto piuttosto recente e di settore, sviluppato presso l'Università di Pisa all'interno del progetto netmap — da non confondere quindi con il generatore `pktgen`[6], incluso nel *kernel* linux; né con `Pktgen-DPDK`[7], basato sul *framework* Intel DPDK.

Basato appunto su netmap, permette di generare flussi UDP (verso singoli indirizzi o insiemi di indirizzi) con velocità nell'ordine delle decine di milioni di pacchetti al secondo. È quindi uno strumento davvero utile nell'ambito della generazione di traffico e della sperimentazione su reti 10 Gbit Ethernet.

Ostinato

Questo lavoro di tesi è basato su Ostinato[8], un generatore di traffico *open-source* sviluppato di recente, basato sulla libreria `pcap`. Multi-piattaforma, distribuito ed estremamente flessibile, si configura attraverso una interfaccia grafica e supporta in modo versatile una moltitudine di protocolli, permettendo di generare flussi di traffico aventi caratteristiche differenti.

Le caratteristiche e l'organizzazione interna di Ostinato saranno trattate in forma estesa nel Capitolo 2; nel frattempo, la Tabella 1.1 fornisce un utile quadro di sintesi e di confronto dei programmi presentati finora, tutti appartenenti alla classe dei generatori basati su modelli stocastici.

`Packetdrill`

`Packetdrill`[9] è uno strumento di *scripting open-source* appartenente alla classe di generatori basati su liste di istruzioni da eseguire, sviluppato da Google per validare le implementazioni dei protocolli di rete.

Lo strumento opera in *real-time* con l'immagine del sistema operativo sotto esame, controllando che si verifichi una certa sequenza attesa di eventi, per pacchetti e chiamate di sistema. Funziona su Linux, FreeBSD, OpenBSD e NetBSD; inoltre, è portabile su tutti i sistemi operativi conformi allo standard POSIX che supportano la libreria `pcap` (per la cattura e la trasmissione di pacchetti).

Nello specifico, permette di verificare la correttezza e le prestazioni dell'intera pila di protocolli TCP/UDP/IPv4/IPv6, dalle chiamate di sistema fino all'interfaccia con le schede di rete, attraverso la definizione di *script* che sono eseguiti in maniera automatica, precisa e riproducibile.

---

[5] `pkt-gen` (netmap), http://info.iet.unipi.it/~luigi/netmap/

[6] The linux packet generator (`pktgen`), http://www.linuxfoundation.org/collaborate/workgroups/networking/pktgen

[7] `Pktgen-DPDK`, https://github.com/Pktgen/Pktgen-DPDK

[8] http://code.google.com/p/ostinato/

[9] `packetdrill`, http://code.google.com/p/packetdrill/

Gli *script* sono definiti utilizzando un linguaggio la cui sintassi è assimilabile a quelle dei ben noti strumenti `tcpdump` e `strace`, composto da quattro tipi di istruzioni: pacchetti, chiamate di sistema, comandi *shell* (che consentono la configurazione e la verifica dello stato del sistema) e *script* Python.

`Packetdrill` è quindi utile durante l'implementazione di nuove funzionalità, consentendo un modello di sviluppo basato sui *test*; per realizzare *test* di regressione sulle prestazioni; per l'identificazione, la riproduzione e la risoluzione di problemi.

Tcpreplay

Tcpreplay[10] è una ben nota *suite* di strumenti *open-source*, originariamente sviluppati da Aaron Turner e ora mantenuti da AppNeta, che consentono di utilizzare registrazioni (in formato `pcap`) di traffico reale per testare dispositivi e applicazioni di rete[11]. Questi strumenti permettono di classificare il traffico registrato, modificare i valori dei vari campi nelle intestazioni dei pacchetti catturati (a livello *data-link*, di rete e di trasporto) e quindi ritrasmettere il traffico così modificato sulla rete.

1.3 Generazione *software* di traffico su reti ad alta velocità

L'elaborazione dei pacchetti e le comunicazioni con la rete coinvolgono diversi sottosistemi *hardware* e *software*. In base allo specifico contesto operativo, le caratteristiche di questi sottosistemi hanno una incidenza più o meno rilevante sulle prestazioni (si traducono in costi di esercizio) [23].

Per quanto concerne le risorse fisiche, i costi riguardano l'occupazione del processore, della memoria e dei *bus* di sistema (fra processore e memoria, e con le periferiche di I/O).

A livello *software*, poi, è necessario distinguere fra due fattori di costo: quelli legati al sistema operativo e quelli legati all'applicazione considerata. La prima categoria tiene conto delle risorse consumate per spostare i pacchetti dalla rete alle applicazioni, e viceversa; essa include, ad esempio, i costi associati all'elaborazione delle interruzioni, alle chiamate di sistema e alla copia dei dati fra lo spazio utente e il *kernel*. La seconda categoria, invece, tiene conto dei costi legati alle specifiche elaborazioni realizzate dall'applicazione; ad esempio, marcatura temporale, classificazione dei pacchetti, ecc. sono tutti costi applicativi.

In molti sistemi, uno dei fattori (limiti *hardware*, costi di sistema, costi applicativi) predomina sull'altro, nascondendo di fatto informazioni importanti sul comportamento e sui colli di bottiglia presenti nel sistema sotto esame. Ne consegue che è spesso difficile identificare, isolare e comprendere il contributo dei singoli elementi sulle prestazioni complessive del sistema.

---

[10] http://tcpreplay.synfin.net

[11] Gli strumenti che fanno parte della *suite* sono: `tcpprep`, `tcprewrite`, `tcpreplay`, `tcpliveplay`, `tcpreplay-edit`, `tcpbridge` e `tcpcapinfo`.

Le reti ad alta velocità pongono sfide impegnative per la generazione *software* del traffico. Su un collegamento 10 Gbit Ethernet, infatti, possono transitare fino a 14,88 milioni di pacchetti al secondo (Mpps): uno ogni 67,2 nanosecondi (all'incirca ogni 200 cicli di *clock*). I generatori di traffico devono quindi raggiungere notevoli velocità trasmissive; purtroppo, senza accorgimenti particolari, o *hardware* dedicato, i sistemi operativi convenzionali e i programmi in esecuzione su di essi non sono in grado di gestire un volume di traffico così elevato.

Nel caso dei generatori *software*, le difficoltà derivano dal fatto che il sottosistema di rete dei sistemi operativi *general-purpose* è progettato per essere flessibile e modulare, in modo da supportare le numerose tecnologie e i vari protocolli di rete esistenti ed essere in grado di accogliere le innovazioni future con relativa semplicità. All'interno del *kernel*, ogni pacchetto trattato passa attraverso diversi livelli di elaborazione; in casi specifici, alcuni di questi stadi sono in realtà inutili e potrebbero essere evitati. La generalità si traduce quindi in tempi di elaborazione maggiori e ciò è in contrasto con gli stringenti vincoli prestazionali imposti dalle reti ad alta velocità.

Inoltre, i generatori di traffico sono strumenti di misura; oltre al volume di traffico producibile, è importante che la trasmissione dei pacchetti sia un processo preciso e accurato, ovvero rispetti i valori e le proprietà statistiche attese (velocità imposte e distribuzione dei tempi relativi fra l'invio di un pacchetto e il successivo). Nel caso dei generatori *software*, tali proprietà possono essere compromesse da fattori esterni e interni all'applicativo stesso [2].

I fattori *esterni* sono legati alla gestione delle interruzioni e alla presenza di altri processi in esecuzione sul medesimo calcolatore, che si contendono le risorse *hardware* disponibili (processore, memoria e adattatori di rete); causano ritardi nell'elaborazione variabili e impredicibili, che dipendono sia dal carico di lavoro a cui è sottoposto l'elaboratore, sia dal volume di traffico gestito.

Altre interferenze (ritardi) sono dovute a fattori *interni*, ovvero alla gestione delle attività complementari alla trasmissione dei dati: la loro preparazione e marcatura temporale (che richiede l'interazione con il sistema operativo); la registrazione (*logging*) delle operazioni effettuate o delle statistiche e delle metriche di funzionamento; ecc.

Per effettuare sperimentazioni e valutazioni su reti ad alta velocità è sì importante poter disporre di un generatore di traffico *software*, ma realizzarne uno che sia al tempo stesso flessibile, preciso, accurato e capace di inviare milioni di pacchetti al secondo (Mpps) non è perciò semplice. È necessario adottare opportuni accorgimenti, sia a livello di sistema operativo (utilizzando appositi *framework*, specificamente progettati per le reti ad alta velocità), sia a livello applicativo (impiegando opportuni accorgimenti progettuali; ad esempio, il parallelismo e i meccanismi di attesa attiva).

1.3.1 Il ruolo di netmap

In questo lavoro di tesi la generazione *software* di traffico ad alta velocità è stata realizzata modificando Ostinato in modo da farlo funzionare con netmap.

Il *framework* netmap [19; 20] è una soluzione alternativa all'inefficiente sottosistema di rete tradizionale, sviluppata per gestire volumi elevati di traffico su sistemi operativi convenzionali (FreeBSD e Linux) senza che sia necessario ricorrere a *hardware* specializzato o estremamente prestante.

Esso adotta delle tecniche progettuali che risolvono o riducono l'impatto di alcune costose operazioni (in termini temporali, ovvero tempo di esecuzione) presenti nel sottosistema di rete tradizionale e normalmente eseguite per ogni pacchetto, fra cui l'allocazione dinamica di memoria, le chiamate di sistema e la copia dei dati.

A livello architetturale, netmap è un modulo del nucleo del sistema operativo usabile in alternativa all'inefficiente sottosistema di rete tradizionale: ne mantiene parte dell'interfaccia esposta alle applicazioni e si fonda su un'area di memoria condivisa fra lo spazio utente e il *kernel* (garantendo comunque la protezione delle aree di memoria del *kernel* e dei registri delle schede di rete).

1.4 Panoramica sullo stato dell'arte

Negli ultimi anni l'architettura degli elaboratori sta evolvendo aumentando il grado di parallelismo dei sistemi; si pensi ad esempio ai processori *multi-core* e alle schede di rete multi-coda (con più code per la ricezione e la trasmissione dei pacchetti). Per sfruttare appieno le possibilità offerte da questo nuovo paradigma nell'ambito delle reti ad alta velocità, è necessario riprogettare i sistemi operativi e le applicazioni. Le applicazioni devono essere organizzate in più flussi di esecuzione indipendenti (*thread*) e devono saper preservare la località dei dati. I sistemi operativi (e in particolare il sottosistema di rete) devono permettere la trasmissione e la ricezione simultanea del traffico, offrendo interfacce *software* che consentano alle applicazioni di sfruttare le caratteristiche *hardware* delle moderne schede di rete [23].

I meccanismi convenzionali offerti dai sistemi operativi *general-purpose* non sono quindi sufficienti per la trasmissione/ricezione del traffico su reti ad alta velocità. Nell'ambito delle reti di calcolatori, la letteratura recente è perciò ricca di tecniche e proposte per realizzare sottosistemi di rete ad alte prestazioni in sistemi operativi *general-purpose* basati su elaboratori convenzionali (*commodity hardware*). Un buon quadro di sintesi a riguardo si trova in Rizzo [19].

Lo specifico problema dello sviluppo di generatori *software* per reti ad alta velocità è però ancora poco trattato, come ammoniscono Rizzo et al. [23] e Bonelli et al. [1].

In particolare, Rizzo et al. [23] offre una panoramica delle tecniche usate per realizzare sistemi veloci di trasmissione/cattura pacchetti su calcolatori convenzionali, descrivendo le obsolescenze progettuali esistenti nei sistemi operativi *general-purpose* e soffermandosi su due nuovi *framework*, netmap e PF_RING DNA, in grado di sfruttare le moderne schede di rete per raggiungere e superare i 10 Gbps di traffico elaborato anche su *hardware* non particolarmente recente.

Bonelli et al. [1] presenta un generatore di traffico estensibile e fortemente parallelo, basato su un nuovo tipo di *socket* per Linux (sviluppato dagli autori

e chiamato `PF_DIRECT`) che, utilizzando una coda *memory-mapped* e un insieme pre-allocato di `sk_buf`, si interfaccia direttamente con la scheda di rete, senza richiedere modifiche ai relativi *device driver*. Il codice del generatore (e del modulo *kernel* che implementa tale *socket*) non è però pubblicamente disponibile e i risultati esposti lasciano spazio a discreti margini di miglioramento; ciò fa supporre che tale strumento sia un prodotto ancora piuttosto giovane, non adatto a un reale utilizzo. In teoria questo generatore offre la possibilità di implementare in modo semplice nuovi modelli di traffico, grazie alla sua architettura modulare basata su *plug-in* che nasconde il motore *multi-thread* sottostante: ai modelli è semplicemente richiesto di generare i pacchetti, specificandone la dimensione e l'*inter-departure time* con il pacchetto precedente. In pratica, però, i modelli devono anche creare l'intero contenuto del pacchetto: il sistema non sembra ancora offrire alcuna *facility* per la gestione dei protocolli standard, come il TCP o l'UDP.

Attraverso due casi di studio (OpenvSwitch, e Click in spazio utente), Rizzo et al. [24] illustra in termini quantitativi i benefici che si possono ottenere a livello prestazionale utilizzando un sottosistema di rete più efficiente (netmap, in questo caso) di quello tradizionale. Di contro, l'articolo mostra anche che spesso non è sufficiente intervenire solo sul sistema operativo: limiti alla capacità di elaborazione possono risiedere nell'applicazione stessa (ovvero, sono legati ai particolari algoritmi o strutture dati utilizzati), magari mascherati dalle inefficienze del sistema di I/O sottostante.

Botta et al. [2] fornisce interessanti elementi riguardo le prestazioni e il livello di accuratezza di alcuni popolari generatori di traffico *software*: D-ITG, TG, MGEN, RUDE/CRUDE. Inoltre, l'articolo descrive alcuni concetti chiave relativi all'influenza dello *scheduler* del sistema operativo sull'accuratezza del processo di trasmissione, suggerendo alcune tecniche progettuali per aumentare sia l'accuratezza, sia le prestazioni raggiungibili; ad esempio, utilizzare le attese attive per regolare l'intervallo temporale fra pacchetti successivi, piuttosto che far sospendere il trasmettitore mediante chiamate di sistema come `select()` o `poll()`.

Infine, Molnár et al. [11] elenca e classifica le metriche usate nella recente letteratura per validare il funzionamento dei generatori di traffico *software*, facendo presente come non c'è in realtà un consenso, nell'ambito della ricerca scientifica, su come misurare le prestazioni di questi strumenti e quali aspetti considerare (relativamente alle caratteristiche del traffico prodotto). Questo rende difficile valutare e confrontare in modo semplice i risultati reperibili in letteratura sui diversi generatori di traffico esistenti.

1.4.1 Adozione di Ostinato nell'industria e nella ricerca

Ostinato è stato sviluppato di recente — la prima versione (v0.1) risale a Giugno 2010. Ciò nonostante, alcuni prodotti commerciali e delle ricerche già sfruttano le possibilità offerte da questo programma: in particolare, la grande flessibilità nella definizione dei flussi di traffico da trasmettere, configurabili anche attraverso un'interfaccia grafica intuitiva.

All'esposizione *Interop*[12] di New York (Ottobre 2010), ad esempio, *Napatech*[13] ha mostrato una soluzione a basso costo per la generazione di traffico a 20 Gbps (aggregati) basata su Ostinato, *hardware* convenzionale e adattatori di rete *Napatech* NT20E2 [12]. Un approccio simile è adottato da *NextComputing*[14], che commercializza generatori di traffico economici (chiamati *Continuum TGEN* [13]) basati su architettura PC, adattatori *Napatech* 1–40 Gbit Ethernet e Ostinato, capaci di trasmettere fino a 20 Gbps (aggregati) e, contemporaneamente, catturare il traffico fino a 10 Gbps.

In ambito accademico, l'adozione di Ostinato sembra essere cauta. Da una ricerca effettuata da chi scrive su Google Scholar[15] negli ultimi giorni di Dicembre 2013, sono emersi nove lavori (fra articoli, tesi di laurea e *proceedings* di conferenze) che utilizzano tale generatore di traffico, ad esempio per analisi prestazionali, o per valutare il funzionamento di sistemi di sicurezza. Ostinato viene inoltre citato in almeno altri undici lavori.

Volendo individuare quelle che sono le principali mancanze di Ostinato, ovvero le cause che più ne limitano o rallentano l'adozione, è possibile addurre le seguenti argomentazioni.

- L'assenza di studi sperimentali per la valutazione delle proprietà metrologiche del programma.
- L'impossibilità di utilizzare modelli statistici per definire il profilo di traffico da generare (è supportata solo la trasmissione di *stream* a velocità costante).
- La mancanza di una interfaccia a riga di comando e/o di *scripting*. Tale funzionalità è comunque prevista nella *roadmap* del progetto e verrà implementata nel prossimo futuro.

1.5 Obiettivi e organizzazione di questa tesi

Il contributo di questo lavoro di tesi, incentrato su Ostinato e la generazione di traffico su reti Ethernet ad alta velocità, si può descrivere in maniera sintetica come qui di seguito indicato.

- Introduzione del supporto nativo a netmap.
- Miglioramento dell'accuratezza e della velocità nella generazione dei pacchetti, mediante interventi mirati su alcune parti fondamentali del codice (lato *server*).

[12] http://www.interop.com

[13] *Napatech*: importante produttore di schede di rete multi-porta "intelligenti", per reti Ethernet ad alta velocità (1–40 Gbps), http://www.napatech.com

[14] *NextComputing*: azienda specializzata nella costruzione di *computer* trasportabili con prestazioni estreme, http://www.nextcomputing.com

[15] http://scholar.google.it

- Valutazione metrologica sperimentale dell'applicativo (anche attraverso l'introduzione di un sistema per il *logging* delle metriche e delle statistiche di trasmissione).

Al momento, infatti, non esiste in letteratura alcuna valutazione sperimentale delle proprietà metrologiche dell'applicativo (numero massimo di pacchetti trasmettibili al secondo, accuratezza del suo processo di generazione dei pacchetti, ecc.). In questa tesi si colmerà perciò tale mancanza.

Si è reso inoltre evidente, nello svolgimento del lavoro, che Ostinato rappresenta uno di quei casi menzionati in precedenza (cfr. sezione 1.4) per cui non è sufficiente l'utilizzo di un sottosistema di rete efficiente per ottenere di riflesso buone prestazioni (alte velocità, accuratezza), ma è necessario intervenire anche sul motore di trasmissione dei pacchetti. Quindi, oltre a realizzare l'adattamento con netmap (utilizzandone l'interfaccia di programmazione nativa), la parte di codice che gestisce in Ostinato la trasmissione dei pacchetti è stata riscritta, sia in termini di strutture dati, sia di algoritmi; il nuovo motore di trasmissione così sviluppato risulta essere, al contempo, più semplice, preciso e accurato.

I miglioramenti derivano per lo più dall'adozione di una strategia di recupero dei ritardi, che consente di evitare, o ammortizzare su più pacchetti (dipende dai casi), alcuni costi fissi associati alle chiamate di sistema (ad esempio, quelle legate alla temporizzazione) andando a trasmettere immediatamente quei pacchetti che a un dato istante già dovevano essere stati inviati. Ciò avviene in maniera dinamica e adattativa, in modo da rispettare il profilo di traffico richiesto dall'utente.

Il resto della tesi è strutturato in questo modo: il Capitolo 2 illustra le caratteristiche e l'architettura *software* di Ostinato, la cui conoscenza risulta propedeutica alla descrizione del lavoro svolto, argomento oggetto del Capitolo 3; successivamente, il Capitolo 4 presenta e discute i risultati ottenuti, mentre il Capitolo 5 fornisce le conclusioni e offre una panoramica sui possibili sviluppi futuri.

Capitolo 2

Ostinato

Ostinato [27] è un *software* di rete *open-source* e multi-piattaforma che permette di confezionare pacchetti e generare traffico in maniera semplice e flessibile, attraverso una interfaccia grafica.

In questo capitolo verranno presentate le caratteristiche, le modalità di funzionamento (anche attraverso la descrizione di un caso concreto) e l'architettura *software* di Ostinato. Si potrà quindi comprendere quali sono le funzionalità offerte dall'applicativo e conoscerne l'organizzazione interna. Quest'ultimo aspetto sarà utile in seguito, quando verrà esposto il lavoro di adattamento con netmap (vedi Capitolo 3), ma anche in generale: il codice sorgente è infatti poco commentato, e la documentazione disponibile in rete scarna.

2.1 Caratteristiche e funzionalità

Ostinato è stato brevemente introdotto nella sottosezione 1.2.1 (vedi in particolare la Tabella 1.1). Questo programma si caratterizza per semplicità di utilizzo (in quanto si configura attraverso una interfaccia grafica[1]) e versatilità.

Ostinato implementa infatti i protocolli di rete standard più comuni nell'ambito delle reti Ethernet e dello *stack* TCP/IP[2], permettendo di combinare tali protocolli a piacimento e di configurare/variare qualsiasi campo di tali protocolli, anche per singolo pacchetto.

[1] Resta il fatto che il supporto di una interfaccia basata su riga di comando, o di *scripting*, è forse la funzionalità più richiesta e attesa, al momento, dalla comunità di Ostinato [14]. Interfacce di questo tipo, infatti, sono comuni nell'ambito delle reti, permettendo di automatizzare una serie di compiti (fra cui la configurazione dei flussi di traffico da trasmettere, per l'analisi e la verifica del funzionamento e/o delle prestazioni di un sistema), oltre a rendere lo strumento stesso più scalabile e flessibile.

Sulla *mailing-list* del progetto è circolata di recente una *patch* [17] che introduce il supporto a Python, rendendo di fatto meno opprimente questa pesante mancanza. Tale contributo si è rivelato prezioso nell'ambito di questo lavoro di tesi proprio per automatizzare la gran parte dei *test* prestazionali effettuati sull'applicativo.

[2] Nuovi protocolli possono essere introdotti con relativa semplicità, mediante l'apposito *framework*.

Inoltre, permette di catturare su *file* `pcap` [30] il traffico trasmesso e ricevuto su una o più interfacce di rete, per una successiva ritrasmissione o analisi — quest'ultima da effettuarsi con altri strumenti, ad esempio Wireshark [32].

Ostinato supporta i maggiori sistemi operativi (fra cui Linux, BSD, Mac OS X e Windows)[3], utilizza `libpcap/WinPcap` [18; 31], e si basa su una architettura *client/server* nella quale più *server* possono essere controllati da un e un solo *client* (tali componenti possono essere in esecuzione su elaboratori differenti, magari con sistemi operativi diversi, o sullo stesso elaboratore).

Ogni *server* gestisce le interfacce di rete della macchina su cui è in esecuzione, mentre attraverso il *client* è possibile configurare tipologia, flussi e profili di traffico da trasmettere, regolando il processo di trasmissione/cattura su ogni scheda di rete dei *server* controllati.

Il traffico generato dalla singola interfaccia di rete di un *server* può essere costituito da uno o più *stream*, che possono essere trasmessi in maniera sequenziale o in parallelo, ognuno con una propria costituzione (protocolli di rete impiegati, valori per i campi di controllo, contenuto informativo, ecc.) e inviato a velocità differenti (in termini di pacchetti al secondo; o di *burst* al secondo e numero di pacchetti per *burst*).

> Ostinato utilizza il termine *stream*, che va inteso come *attività da realizzare*, o anche *gruppo di pacchetti* da trasmettere. Un flusso di traffico, secondo l'accezione *5-Tuple*[4], può essere infatti descritto mediante uno o più *stream*.

2.1.1 Caso d'uso: generazione di un flusso UDP costante

Questa sezione fornisce una panoramica dell'interfaccia, dell'utilizzo e delle possibilità offerte da Ostinato, descrivendo un caso concreto: la generazione di un flusso UDP a velocità costante, costituito da pacchetti Ethernet di dimensione minima. La procedura di configurazione verrà spiegata passo passo, mostrando e commentando le funzioni e i parametri più significativi per il caso presentato. Per una trattazione approfondita sulla configurazione di Ostinato e sul significato delle varie opzioni si rimanda alla guida utente del programma, reperibile *online*[5].

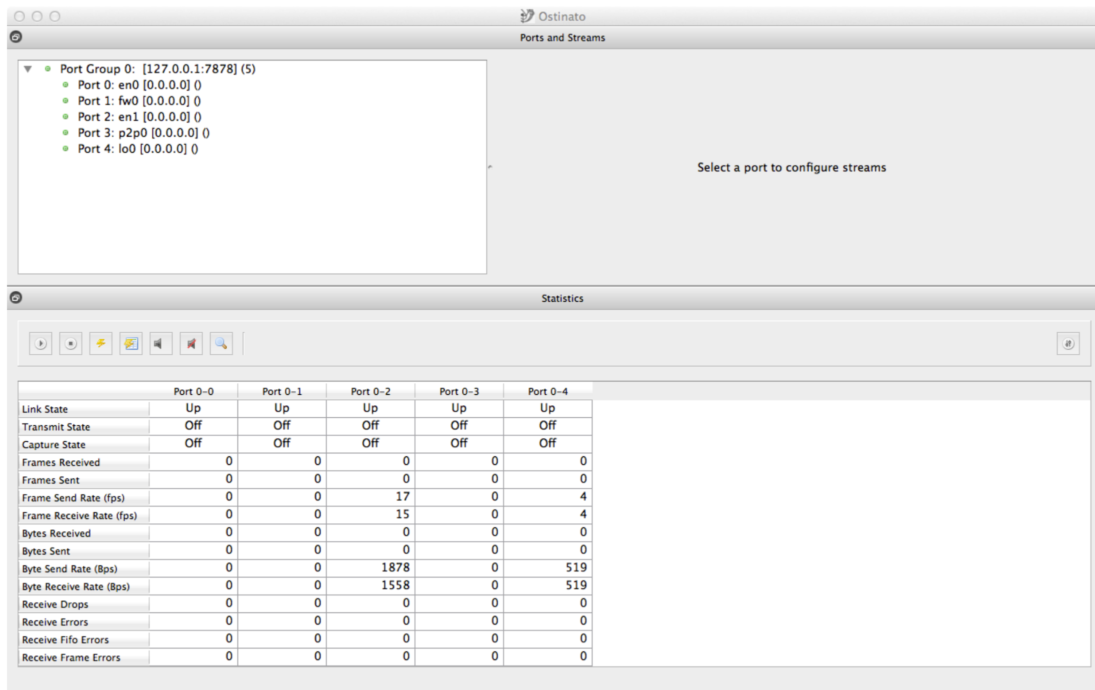
La Figura 2.1 mostra come si presenta l'interfaccia grafica di Ostinato (o meglio, della sua componente *client*). L'area di lavoro è divisa in tre sezioni principali.

- Server connessi e adattatori di rete disponibili; in alto a sinistra.
- Elenco e configurazione degli *stream* per l'adattatore di rete selezionato; in alto a destra.

[3]Per eseguire Ostinato, sono richiesti i privilegi di amministratore.

[4]Indirizzo IP sorgente, indirizzo IP destinazione, numero di porta sorgente, numero di porta destinazione e protocollo in uso.

[5]Guida utente di Ostinato, http://code.google.com/p/ostinato/wiki/UserGuide



	Port 0-0	Port 0-1	Port 0-2	Port 0-3	Port 0-4
Link State	Up	Up	Up	Up	Up
Transmit State	Off	Off	Off	Off	Off
Capture State	Off	Off	Off	Off	Off
Frames Received	0	0	0	0	0
Frames Sent	0	0	0	0	0
Frame Send Rate (fps)	0	0	17	0	4
Frame Receive Rate (fps)	0	0	15	0	4
Bytes Received	0	0	0	0	0
Bytes Sent	0	0	0	0	0
Byte Send Rate (Bps)	0	0	1878	0	519
Byte Receive Rate (Bps)	0	0	1558	0	519
Receive Drops	0	0	0	0	0
Receive Errors	0	0	0	0	0
Receive Fifo Errors	0	0	0	0	0
Receive Frame Errors	0	0	0	0	0

Figura 2.1: La finestra principale di Ostinato (componente *client*)

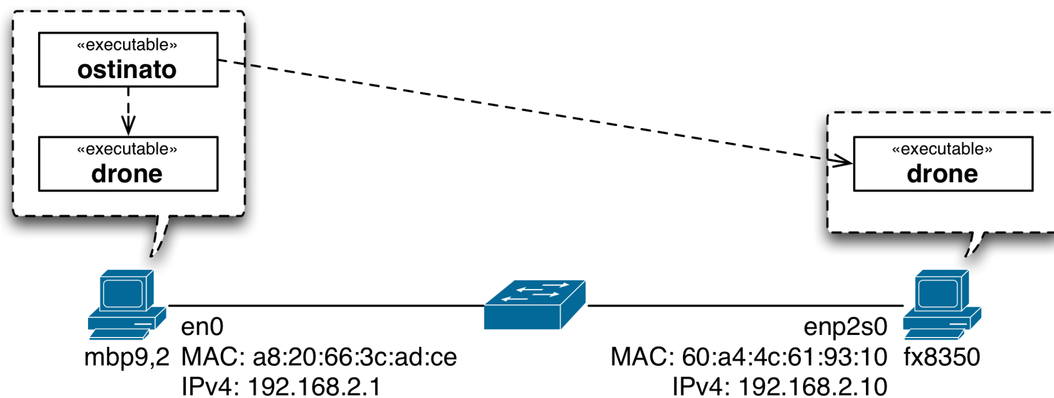


Figura 2.2: Esempio flusso UDP, lo scenario per il caso d'uso

- Controllo, stato e statistiche degli adattatori di rete disponibili; nella parte inferiore.

Attraverso l'interfaccia grafica è possibile configurare, controllare e monitorare le schede di rete su ogni *server* a cui ci si collega; i *server* sono indicati con la dicitura `Port Group`. La modalità di funzionamento predefinita consiste nel gestire l'elaboratore locale (quello su cui si avvia l'interfaccia grafica), rappresentato dall'indirizzo `127.0.0.1:7878`. Ci si può connettere ad altri *server* mediante `File>New Port Group`.

Lo scenario di esempio è riportato in Figura 2.2: il traffico sarà originato dall'adattatore `enp2s0` del calcolatore remoto (su cui è in esecuzione solo la com-

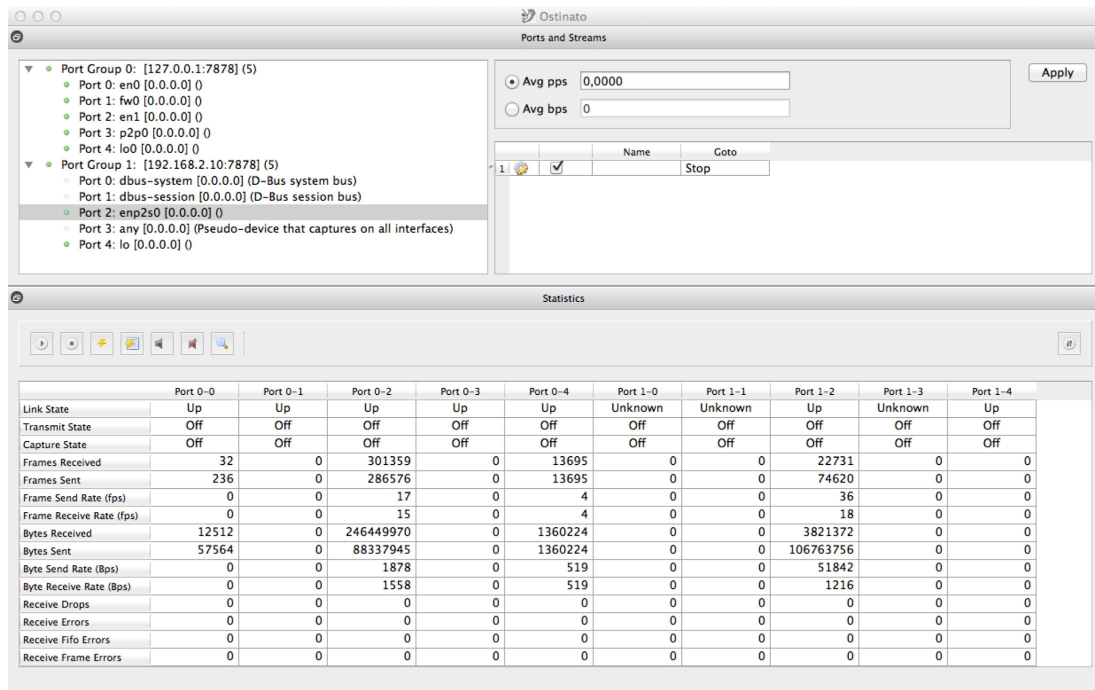


Figura 2.3: Esempio flusso UDP, il nuovo *stream* per la porta selezionata

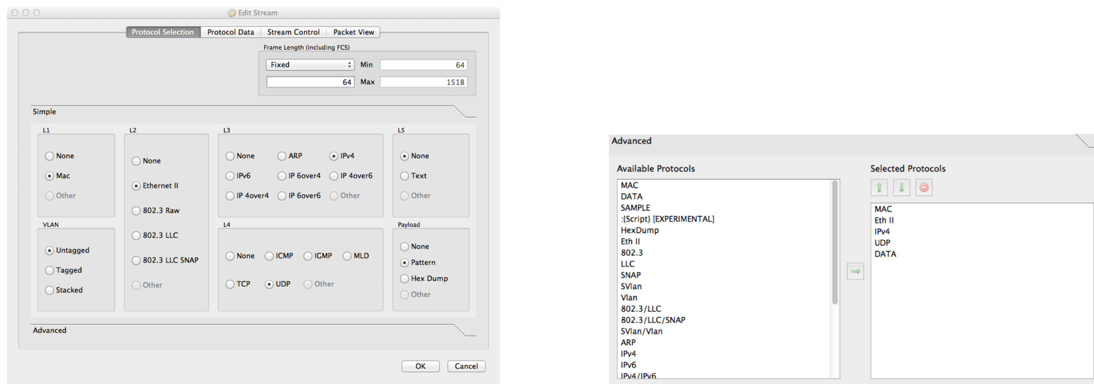
ponente *server* di Ostinato) verso l'adattatore `en0` del calcolatore locale (sul quale sono in esecuzione entrambe le componenti *client* e *server* di Ostinato)[6].

Per procedere alla configurazione del flusso (o dei flussi) di traffico da generare è necessario selezionare l'adattatore d'interesse, ovvero quello che sarà la sorgente di tale traffico, quindi scegliere `File>New Stream`.

> Quando si configurano più *stream* è possibile decidere se trasmetterli in sequenza o in parallelo.
>
> Nella modalità di trasmissione *sequenziale*, gli *stream* sono inviati uno dopo l'altro: si inviano tutti i pacchetti del primo *stream*, quindi si passa al secondo, e così via, rispettando le velocità imposte per ognuno di essi.
>
> Nella seconda modalità, chiamata *interleaved*, gli *stream* sono trasmessi insieme, rispettando i rapporti di velocità relativa; questa modalità è chiamata anche "continua", in quanto non è possibile specificare il numero di pacchetti/*burst* che costituiscono ogni singolo *stream*, ma solo le loro velocità. La modalità di trasmissione si sceglie mediante `File>Port Configuration`; la modalità sequenziale è quella predefinita.

A seguito di ciò, all'elenco degli *stream* configurati sulla scheda di rete selezionata viene aggiunta una entrata, come mostrato in Figura 2.3. Ogni *stream* può

[6]La componente *server* di Ostinato prende il nome di `drone`, mentre la componente *client* che implementa l'interfaccia grafica è chiamata `ostinato` (con l'iniziale minuscola).



(a) Simple Mode (b) Advanced Mode

Figura 2.4: Esempio flusso UDP, scheda `Protocol Selection`

avere un nome associato, mentre la casella di selezione determina se tale *stream* è abilitato, cioè se verrà trasmesso o meno. La colonna `Goto`, invece, regola il comportamento del trasmettitore una volta completata la trasmissione di quello *stream*; le possibili opzioni sono: "fermati", "passa allo *stream* successivo" e "ricomincia dal primo".

A questo punto è possibile specificare le proprietà dello *stream* appena creato, selezionandolo e scegliendo `File>Edit Stream`. La finestra che si apre è costituita da varie schede che consentono di configurare quali protocolli impiegare (`Protocol Selection`), i valori dei campi di controllo e il contenuto informativo per questi protocolli (`Protocol Data`), il numero di pacchetti da trasmettere e la velocità con cui farlo (`Stream Control`), e di verificare la costituzione dei pacchetti così configurati (`Packet View`).

La scheda `Protocol Selection` (Figura 2.4) consente di configurare la dimensione dei pacchetti e i protocolli di rete da impiegare. La dimensione dei pacchetti può essere un valore fisso, o variabile fra due estremi (`Min` e `Max`) in modo crescente, decrescente, o casuale. In ogni caso, la dimensione dei pacchetti include il codice di controllo.

> Per creare pacchetti Ethernet di dimensione minima o massima usare i valori `64` e `1518`, rispettivamente.

I protocolli di rete possono essere selezionati attraverso una interfaccia semplificata (Figura 2.4a), utile nel caso si desideri una combinazione *standard* di protocolli (come nel caso d'uso qui trattato). L'interfaccia avanzata (Figura 2.4b) permette invece di definire combinazioni speciali o non *standard* (ad esempio: *tunneling*, l'utilizzo di tre *tag* VLAN o di *script* definiti dall'utente).

Mediante la scheda `Protocol Data` (Figura 2.5) si configurano i valori per i campi dei vari protocolli selezionati al passaggio precedente. Questa scheda è costituita da varie sezioni. Gli indirizzi MAC possono avere valori fissi, o variare in modo crescente/decrescente: `Count` specifica la cardinalità dell'insieme di questi indirizzi, mentre `Step` determina l'incremento fra un indirizzo e il successivo. Lo stesso dicasi per gli indirizzi IP, per i quali non è possibile specificare

(a) Media Access Protocol

(b) Ethernet II

(c) Internet Protocol v4

(d) User Datagram Protocol

(e) Payload data

Figura 2.5: Esempio flusso UDP, scheda `Protocol Data`

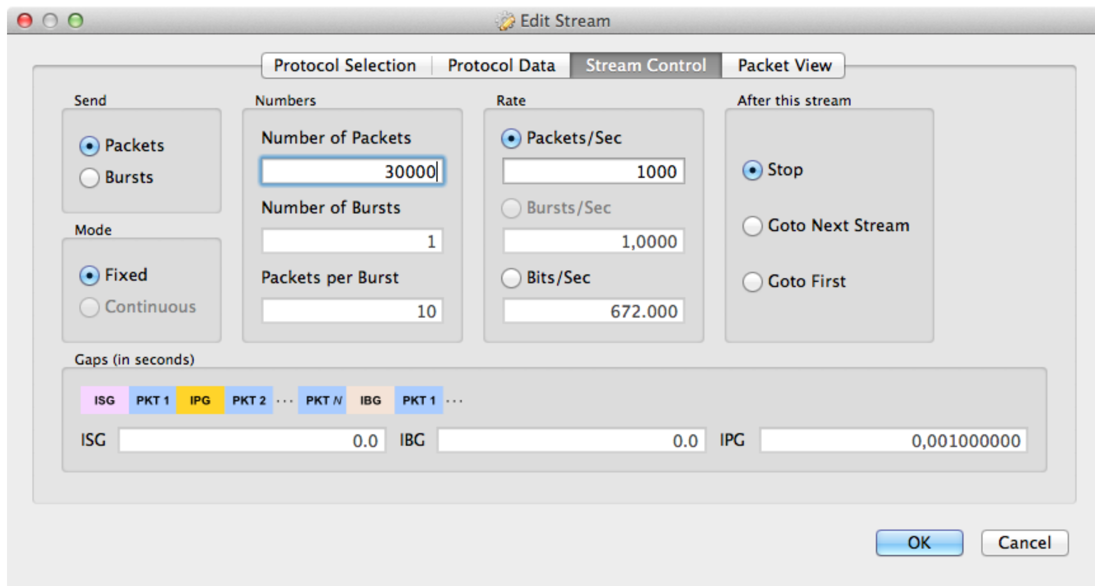


Figura 2.6: Esempio flusso UDP, scheda `Stream Control`

l'incremento, ma è disponibile la variazione casuale dell'indirizzo all'interno dell'intervallo indicato. Il *payload* può essere una parola (32 Byte) fissa, casuale, oppure modificata in modo crescente/decrescente per ogni pacchetto.

Attraverso la scheda `Stream Control` (Figura 2.6) è possibile specificare se inviare pacchetti o *burst* di pacchetti. Nel primo caso, va indicato il numero di pacchetti da trasmettere e la velocità a cui farlo espressa in termini di pacchetti al secondo, o bit al secondo. Nel secondo caso, vanno definiti il numero di *burst* da trasmettere e il numero di pacchetti che costituiscono ogni *burst*; la velocità di trasmissione può essere specificata in *burst* al secondo, o in bit al secondo (tutti i pacchetti di un *burst* sono spediti uno di seguito l'altro, senza pause nel mezzo).

> Non è possibile indicare esplicitamente la durata dello *stream* in termini temporali. Nel caso di modalità trasmissiva continua, inoltre, si può configurare solo la velocità trasmissiva (e non il numero di pacchetti da trasmettere).
>
> Per inviare uno *stream* alla massima velocità possibile, usare il valore `0 pps`.

La scheda `Packet View` (Figura 2.7) permette di visualizzare la struttura del primo pacchetto dello *stream*, e il suo corrispettivo in codice esadecimale.

> Se lo *stream* è costituito da più pacchetti con campi variabili, questi non possono essere ispezionati (ovvero, è visualizzabile solo il primo).

Completata la configurazione dello *stream*, questa va confermata premendo il bottone `Apply` in alto a destra. Le informazioni di configurazione vengono inviate al *server*, ed è quindi possibile procedere alla trasmissione.

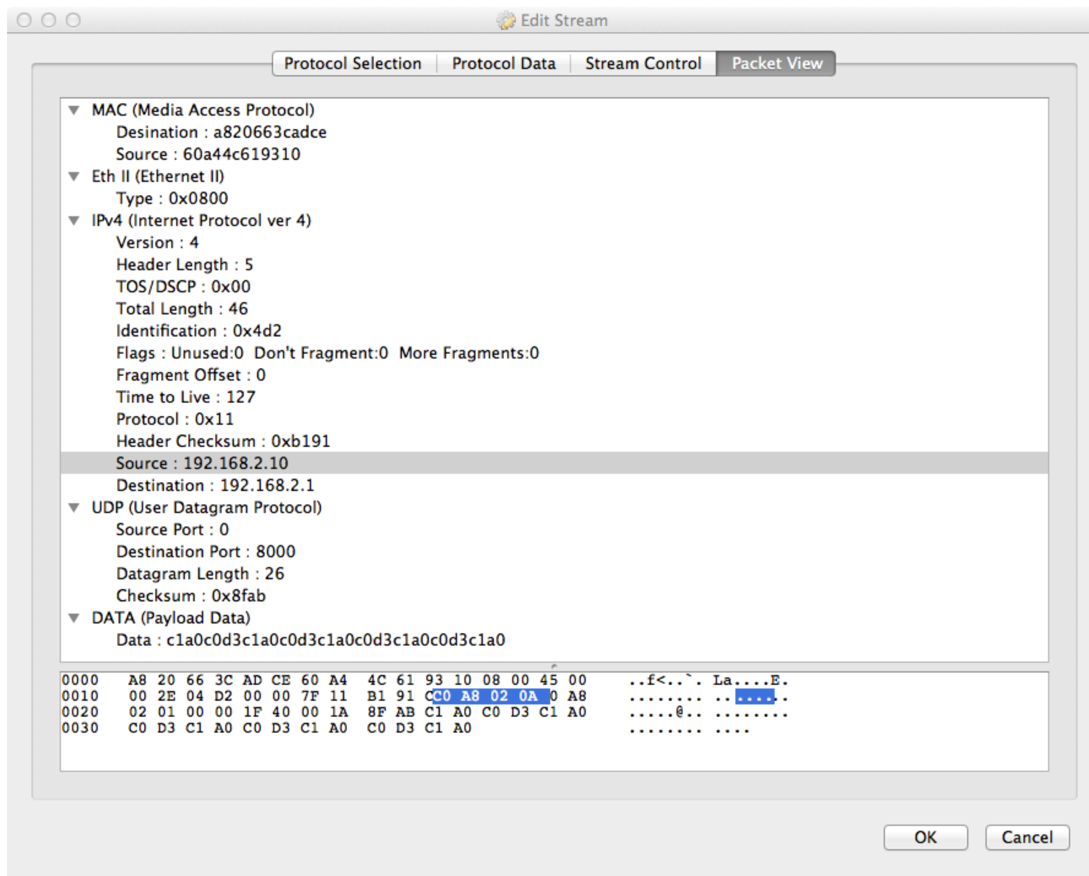


Figura 2.7: Esempio flusso UDP, scheda `Packet View`

> È possibile salvare l'elenco degli *stream* definiti su una scheda di rete, selezionandola e scegliendo `File>Save Streams`, in modo da riutilizzarla, in un secondo momento o su un differente adattatore, tramite la funzione `File>Load Streams`. Quest'ultima funzione permette di scegliere se aggiungere gli *stream* che si stanno caricando in coda a quelli già presenti sulla scheda, oppure al posto di quelli esistenti.

Per dare l'avvio alla trasmissione, è necessario selezionare, nella finestra delle statistiche in basso, la colonna corrispondente alla scheda di rete sorgente (facendo *click* sulla riga di intestazione), quindi premere il bottone `Start Transmit`: il primo della lista (Figura 2.8). Gli altri pulsanti permettono di interrompere la trasmissione, azzerare le statistiche, iniziare/interrompere la cattura dei pacchetti sulle interfacce selezionate, o visualizzare i pacchetti catturati (quest'ultima funzione richiede Wireshark).

2.2 Architettura *software* di Ostinato

L'architettura di un sistema ne descrive la struttura organizzativa (in termini di componenti costitutivi, interfacce, relazioni fra tali componenti e con l'ambiente esterno, vincoli esistenti, ecc.), il suo comportamento, e i principi alla base del suo

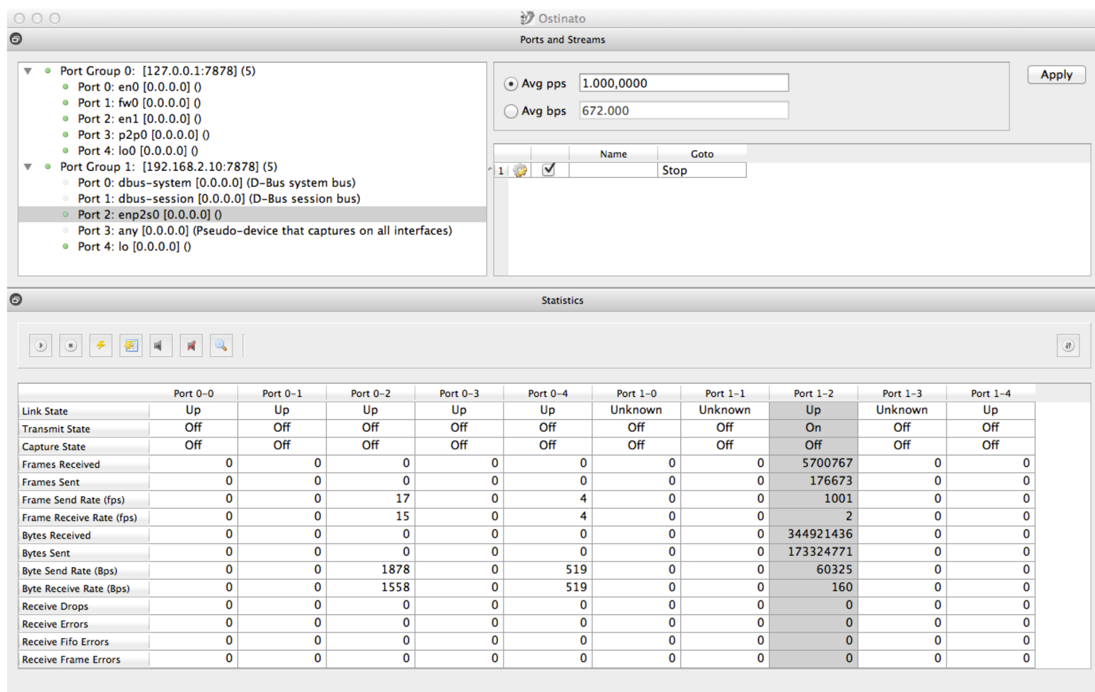


Figura 2.8: Esempio flusso UDP, trasmissione in corso

progetto e della sua evoluzione. In un sistema *software*, le parti che interagiscono attraverso interfacce comprendono classi, componenti e sottosistemi[7].

Questa sezione presenterà gli elementi dell'architettura *software* di Ostinato che sono rilevanti ai fini del lavoro svolto nell'ambito di questa tesi. In particolare, ci si focalizzerà sulla struttura della componente *server*.

Ostinato è scritto in linguaggio `C++` [29]; si basa sulle librerie Qt[8] e utilizza il meccanismo dei Protocol Buffers[9] per serializzare i dati nelle chiamate di procedura remota e per il salvataggio degli *stream* su *file*.

2.2.1 Le modalità di interazione fra *client* e *server*

Il programma è strutturato secondo una architettura *client/server*. Le due entità, che sono due eseguibili distinti, comunicano mediante *socket* utilizzando il protocollo TCP; il formato dei messaggi scambiati e le chiamate di procedura remota sono definite utilizzando i Protocol Buffers[10] (vedi Figura 2.9).

---

[7]IEEE Std 1471–2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE, September 2000. URL http://standards.ieee.org/findstds/standard/1471-2000.html.
Len Bass, Paul Clements and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley Professional, 3th edition, September 2012. ISBN 978–0–321–81573–6.

[8]Qt Project, http://qt-project.org

[9]Google Protocol Buffers, http://code.google.com/p/protobuf/

[10]Tali definizioni sono contenute nel *file* `common/protocol.proto`.

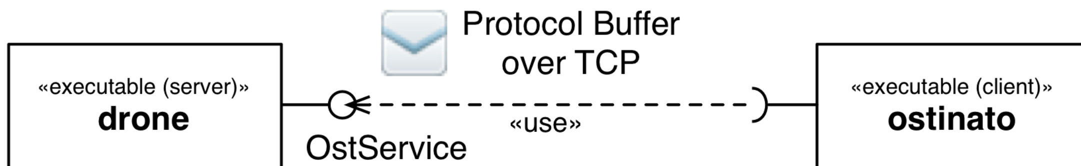


Figura 2.9: Il meccanismo di comunicazione fra i componenti di Ostinato

La componente *server*, chiamata `drone`, gestisce le interfacce di rete (d'ora in avanti denominate "porte" per non far confusione con il concetto di "interfaccia *software*") dell'elaboratore su cui è in esecuzione, secondo le indicazioni del *client*.

La versione corrente di Ostinato (v0.5.1) prevede che ogni *server* può essere connesso, al più, a un solo *client*, mentre un singolo *client* può controllare più *server*.

Ostinato dispone di un *client* basato su interfaccia grafica (realizzata con le librerie Qt), che prende il nome di `ostinato` (con l'iniziale minuscola). Di recente, un utente ha pubblicato sulla *mailing list* del progetto una *patch* (non ufficiale) che implementa una interfaccia di *scripting* basata su Python [17].

Entrando nello specifico, i servizi messi a disposizione da `drone` sono realizzati da una classe `MyService` che implementa l'interfaccia `OstService`. Tale interfaccia, riportata nel Listato 2.1, permette di svolgere le seguenti funzioni.

- Recuperare le informazioni sulle porte disponibili, la loro configurazione e il loro stato (inclusi gli *stream* definiti).
- Configurare le porte, relativamente al controllo esclusivo e alla modalità trasmissiva.
- Configurare gli *stream* su ogni porta.
- Controllare la trasmissione dei pacchetti su ogni porta.
- Controllare la cattura dei pacchetti trasmessi e ricevuti su ogni porta, e recuperare i dati catturati.
- Controllare le statistiche su ogni porta.

Listato 2.1: L'interfaccia `OstService`

```
service OstService {
    rpc getPortIdList(Void) returns (PortIdList);
    rpc getPortConfig(PortIdList) returns (PortConfigList);
    rpc modifyPort(PortConfigList) returns (Ack);

    rpc getStreamIdList(PortId) returns (StreamIdList);
    rpc getStreamConfig(StreamIdList) returns (StreamConfigList);
    rpc addStream(StreamIdList) returns (Ack);
    rpc deleteStream(StreamIdList) returns (Ack);
    rpc modifyStream(StreamConfigList) returns (Ack);
```

```
    rpc startTx(PortIdList) returns (Ack);
    rpc stopTx(PortIdList) returns (Ack);

    rpc startCapture(PortIdList) returns (Ack);
    rpc stopCapture(PortIdList) returns (Ack);
    rpc getCaptureBuffer(PortId) returns (CaptureBuffer);

    rpc getStats(PortIdList) returns (PortStatsList);
    rpc clearStats(PortIdList) returns (Ack);
}
```

Ogni porta è identificata da un numero non negativo, e ad essa possono essere associati un nome, una descrizione e delle note testuali (Listato 2.2). Lo stato dell'interfaccia, se abilitata o meno, dipende dalla configurazione amministrativa; la modalità con cui gli *stream* sono trasmessi, in sequenza o in parallelo, è invece regolata dall'utente attraverso il *client* (nell'interfaccia grafica, mediante `File>Port Configuration`). È possibile marcare una porta ad uso esclusivo (`File>Exclusive Port Control`), in modo da evitare che il sistema operativo la usi per inviare pacchetti non generati da Ostinato; questa funzione, utile per creare un ambiente di *test* controllato, è sperimentale e al momento disponibile solo su Windows.

Listato 2.2: I Protocol Buffers che descrivono una porta

```
enum TransmitMode {
    kSequentialTransmit = 0;
    kInterleavedTransmit = 1;
}

message PortId {
    required uint32 id = 1;
}

message PortIdList {
    repeated PortId port_id = 1;
}

message Port {
    required PortId port_id = 1;
    optional string name = 2;
    optional string description = 3;
    optional string notes = 4;
    optional bool is_enabled = 5;
    optional bool is_exclusive_control = 6;
    optional TransmitMode transmit_mode = 7 [default = ↩
       kSequentialTransmit];
}
```

```
message PortConfigList {
    repeated Port port = 1;
}
```

Per ogni porta, le statistiche registrate riguardano: il numero di pacchetti, pacchetti al secondo, Byte, Byte al secondo trasmessi e ricevuti; il numero degli errori riscontrati in ricezione; inoltre, si tiene traccia dello stato del collegamento e dello stato di funzionamento riguardo la trasmissione e la cattura (Listato 2.3).

Listato 2.3: I Protocol Buffers che descrivono statistiche e stato d'una porta

```
enum LinkState {
    LinkStateUnknown = 0;
    LinkStateDown = 1;
    LinkStateUp = 2;
}
message PortState {
    optional LinkState link_state = 1 [default = LinkStateUnknown];
    optional bool is_transmit_on = 2 [default = false];
    optional bool is_capture_on = 3 [default = false];
}

message PortStats {
    required PortId port_id = 1;
    optional PortState state = 2;

    optional uint64 rx_pkts = 11;
    optional uint64 rx_bytes = 12;
    optional uint64 rx_pkts_nic = 13;
    optional uint64 rx_bytes_nic = 14;
    optional uint64 rx_pps = 15;
    optional uint64 rx_bps = 16;

    optional uint64 tx_pkts = 21;
    optional uint64 tx_bytes = 22;
    optional uint64 tx_pkts_nic = 23;
    optional uint64 tx_bytes_nic = 24;
    optional uint64 tx_pps = 25;
    optional uint64 tx_bps = 26;

    optional uint64 rx_drops = 100;
    optional uint64 rx_errors = 101;
    optional uint64 rx_fifo_errors = 102;
    optional uint64 rx_frame_errors = 103;
}

message PortStatsList {
    repeated PortStats port_stats = 1;
}
```

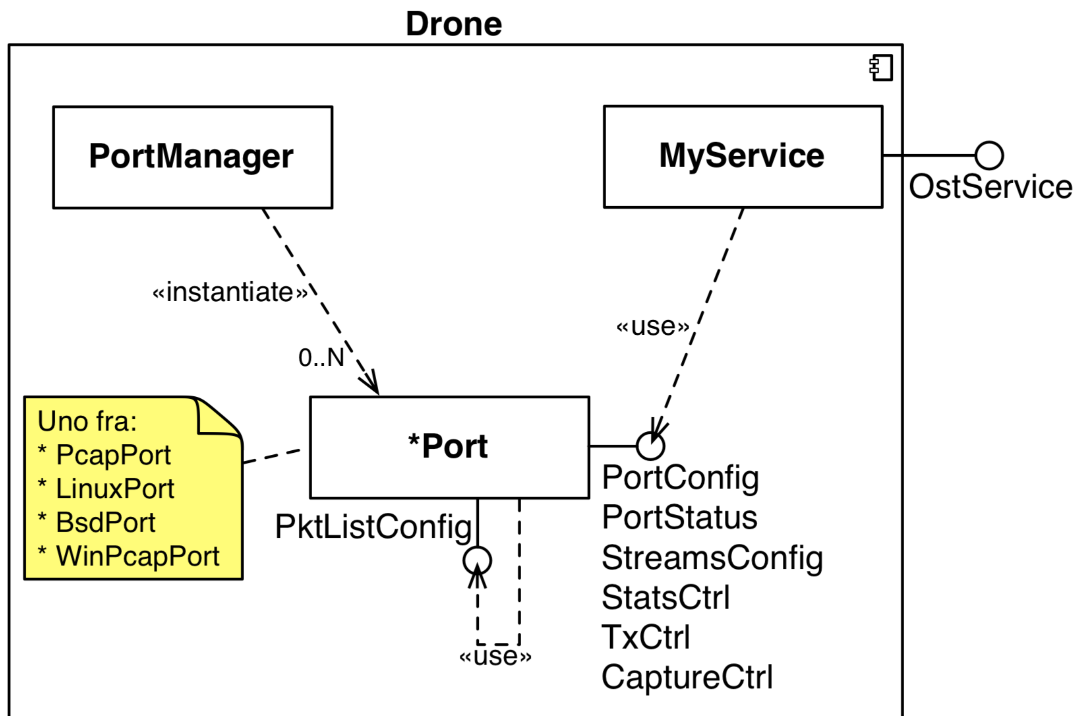


Figura 2.10: La struttura interna di `drone`

2.2.2 La struttura interna del *server*

Gli elementi fondamentali di `drone` e le loro relazioni sono rappresentati in Figura 2.10.

La classe `MyService` svolge il ruolo di *gateway*, ovvero gestisce tutte le comunicazioni da e verso il *client*. Essa implementa l'interfaccia `OstService`, presentata nella sezione precedente (sottosezione 2.2.1)

La classe `PortManager` identifica, all'avvio di `drone`, tutte le porte presenti sull'elaboratore locale e istanzia un oggetto `Port` per ognuna di esse (responsabile della sua gestione). Il `PortManager` scopre le porte disponibili usando la libreria `pcap`: in particolare, la funzione `pcap_findalldevs()`.

Se la configurazione del sistema cambia, ad esempio viene aggiunta una nuova scheda di rete USB o ThunderBolt, questa non verrà identificata fino al successivo avvio dell'eseguibile.

Per quanto riguarda la gestione delle porte, questa sarà trattata in dettaglio nella sezione successiva (sottosezione 2.2.3).

2.2.3 La gestione delle porte sul *server*

In `drone`, le porte sono rappresentate da una gerarchia di classi, illustrata in Figura 2.11. All'avvio, il `PortManager` istanzia un oggetto per ogni porta identificata sull'elaboratore locale. La classe a cui appartiene l'oggetto istanziato dipende dal sistema operativo in uso.

- Su Linux, vengono istanziati oggetti di classe `LinuxPort`.

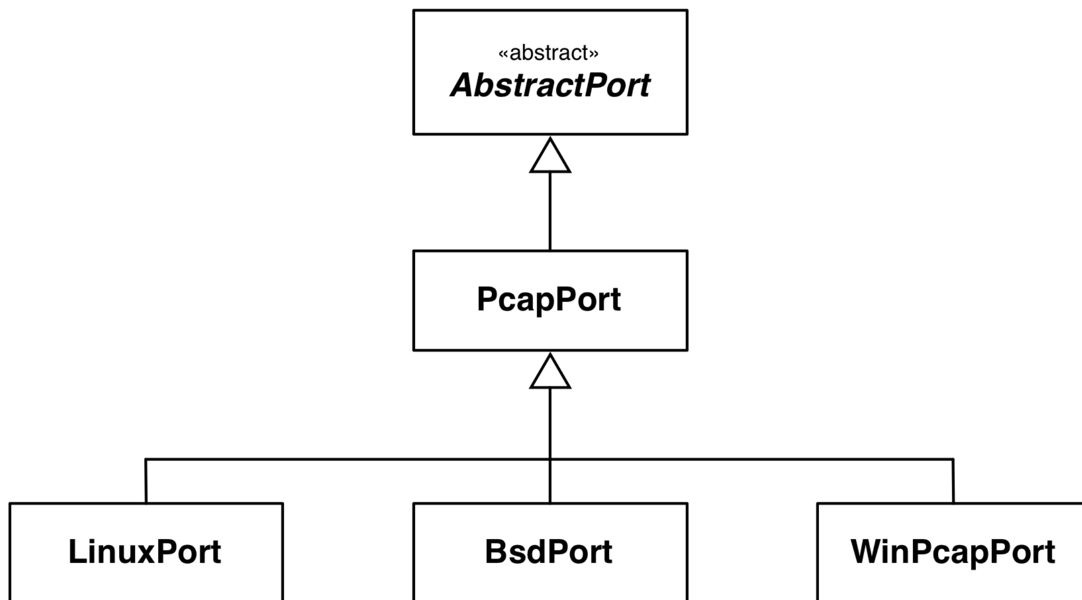


Figura 2.11: La gerarchia di classi per la gestione delle porte in `drone`

- Su FreeBSD e Mac OS X, vengono istanziati oggetti di classe `BsdPort`.
- Su Windows, vengono istanziati oggetti di classe `WinPcapPort`.
- In tutti gli altri casi, vengono istanziati oggetti di classe `PcapPort`.

La classe astratta `AbstractPort`, base della gerarchia, specifica le interfacce usabili dal *gateway* `MyService` per soddisfare le richieste di servizio provenienti dal *client*. Inoltre, implementa alcune importanti funzionalità di base per la gestione degli *stream*: in particolare, fornisce gli algoritmi che traducono gli *stream* configurati su una porta nella lista di pacchetti da trasmettere.

Più nel dettaglio, le interfacce esposte da `AbstractPort` permettono quanto di seguito descritto.

- Configurare e recuperare i dati identificativi e la modalità trasmissiva della porta.
- Recuperare lo stato operativo del collegamento.
- Verificare che la porta sia usabile.
- Configurare gli *stream* per la porta, e recuperare la configurazione corrente. Ogni modifica riguardante gli *stream* associati a una porta comporta la rielaborazione della lista di pacchetti derivata da tali *stream*.
- Controllare le statistiche: recupera dati, resetta. I valori delle statistiche restituiti sono relativi all'ultimo *reset*.
- Popolare la lista dei pacchetti da trasmettere, derivata dagli *stream* associati a una porta.

- Controllare la trasmissione dei pacchetti: inizia, ferma, interroga sullo stato.
- Controllare la cattura dei pacchetti: inizia, ferma, interroga sullo stato, recupera i dati — i dati catturati vanno strutturati nel formato specificato dalla libreria `pcap` [30], così da essere interpretabili da Wireshark.
- Configurare e verificare il controllo esclusivo della porta.

Ogni classe derivata da `AbstractPort`, dovendo implementare le funzioni virtuali pure da essa specificate, ha le seguenti responsabilità.

- Definire e implementare le strutture dati che memorizzano la lista dei pacchetti da trasmettere, da popolare secondo l'interfaccia specificata da `AbstractPort`.
- Trasmettere su richiesta i pacchetti di questa lista, definendo e implementando un opportuno motore di trasmissione per gli stessi.
- Catturare su richiesta i pacchetti, salvandoli su *file* `pcap`.
- Calcolare e collezionare le statistiche dei pacchetti trasmessi e ricevuti.
- Definire una modalità per il controllo esclusivo della porta.

La classe `PcapPort` realizza i primi 3 punti di questo elenco, lasciando alle sottoclassi l'implementazione degli specifici meccanismi per l'aggiornamento delle statistiche e il controllo esclusivo della porta, che dipendono dal sistema operativo in uso. Le classi `LinuxPort`, `BsdPort` e `WinPcapPort` sono quindi specializzazioni della classe `PcapPort` per ogni piattaforma supportata da Ostinato.

Per svolgere i suoi compiti, `PcapPort` utilizza la libreria `pcap` (o il suo equivalente `WinPcap` in ambiente Windows), come il suo nome suggerisce.

2.2.4 Il modello definito da `AbstractPort` per la lista dei pacchetti da trasmettere

La classe astratta `AbstractPort` costruisce la lista dei pacchetti da trasmettere durante la fase di configurazione, cioè prima che la trasmissione abbia inizio: preparare i pacchetti in anticipo evita infatti interferenze e ripercussioni negative sulle prestazioni durante il processo di trasmissione vero e proprio. La gestione di questa lista è sotto la responsabilità delle classi concrete derivate, sia per quanto concerne la definizione delle strutture dati da utilizzare, sia rispetto la gestione della memoria associata. `AbstractPort` fornisce gli algoritmi per popolare la lista dei pacchetti da trasmettere e specifica l'interfaccia che le classi derivate devono implementare per usare tali algoritmi[11].

[11] La classe `PcapPort` e le sue specializzazioni mostrate in Figura 2.11 utilizzano tale soluzione. Qualora una classe concreta non volesse adottarla, questa dovrebbe offrire una implementazione alternativa delle funzioni usate dal *gateway* `MyService` per configurare gli *stream* di una porta, oltre a definire dei nuovi algoritmi di traduzione per determinare la lista dei pacchetti da trasmettere a partire da tali *stream*.

Gli algoritmi[12] forniti da `AbstractPort` traducono gli *stream* configurati su una porta nella corrispondente lista di pacchetti da trasmettere, nella quale ogni pacchetto ha una specifica marcatura temporale che consente di determinare l'istante di trasmissione atteso per lo stesso (o meglio, l'intertempo con il pacchetto precedente).

Le classi derivate che vogliono utilizzare questi algoritmi, devono implementare le funzioni virtuali da essi invocate per popolare la lista dei pacchetti da trasmettere (la cui definizione e gestione è, come detto, opaca ad `AbstractPort`). L'interfaccia in questione è riportata nel Listato 2.4 e di seguito spiegata: affinché possa essere compresa è necessario conoscere il modello definito da `AbstractPort` per la lista dei pacchetti da trasmettere.

Listato 2.4: L'interfaccia per popolare la lista dei pacchetti sulla porta

```
virtual void clearPacketList();
virtual void setPacketListLoopMode(bool loop, quint64 secDelay, ↩
    quint64 nsecDelay);
virtual bool appendToPacketList(long sec, long nsec, const uchar ↩
    *packet, int length)
virtual void loopNextPacketSet(qint64 size, qint64 repeats, long ↩
    repeatDelaySec, long repeatDelayNsec);
```

`clearPacketList()` Azzera il contenuto della lista dei pacchetti da trasmettere.

`setPacketListLoopMode()` Se l'argomento `loop` assume valore `vero`, l'intero contenuto della lista deve essere trasmesso in maniera ciclica, con un certo ritardo fra ogni ripetizione; il valore di questo ritardo è specificato dagli altri argomenti: `secDelay` indica il numero dei secondi e `nsecDelay` il numero di nanosecondi. Altrimenti, la lista va trasmessa una volta sola e i valori di `secDelay` e `nsecDelay` vanno ignorati.

`appendToPacketList()` Inserisce un pacchetto in coda alla lista. Il contenuto del pacchetto si trova nell'area di memoria puntata da `packet`, le cui dimensioni sono specificate dall'argomento `length`. Gli argomenti `sec` e `nsec` specificano la marcatura temporale del pacchetto, in termini di secondi e nanosecondi, rispettivamente. Il primo pacchetto aggiunto alla lista ha *timestamp* `0,0`.

`loopNextPacketSet()` I successivi `size` pacchetti devono essere trasmessi ciclicamente, per `repeats` volte, con un ritardo fra ogni ripetizione pari a `repeatDelaySec` secondi e `repeatDelayNsec` nanosecondi. Questa funzione è invocata *prima* di aggiungere alla lista tale insieme di pacchetti da trasmettere ciclicamente — vedi `appendToPacketList()`.

[12] L'algoritmo effettivamente impiegato dipende dalla modalità trasmissiva configurata per la porta, ovvero se gli *stream* sono da inviare in maniera sequenziale (uno dopo l'altro), o in parallelo.

> Il ciclo definito dalla funzione `setPacketListLoopMode()` è differente da quello definito dalla funzione `loopNextPacketSet()`: il primo riguarda l'intera lista dei pacchetti, mentre il secondo riguarda solo la parte associata a uno specifico *stream*.

Il modello adottato da `AbstractPort` per la lista dei pacchetti da trasmettere è concepito in modo da ridurre il tempo di costruzione della lista stessa e la quantità di memoria necessaria per contenerla.

Dato uno *stream* S, la lista dei pacchetti PL_S che ne deriva può essere rappresentata mediante due liste, PS_S e R_S. La prima contiene i pacchetti all'interno del periodo minimo[13] di S, se esiste; la seconda, i pacchetti da inviare dopo aver completato tutte le ripetizioni (*repeats*) di PS_S — il numero dei pacchetti in PL_S può infatti *non* essere un multiplo intero del numero dei pacchetti in PS_S.

$$S \implies PL_S, \quad PL_S \to \{PS_S, R_S, repeats\}$$

$$|PL_S| = repeats * |PS_S| + |R_S|$$

Ogni pacchetto P_i costituisce l'unità di trasmissione per la scheda di rete: esso contiene tutte le intestazioni dei protocolli utilizzati e il contenuto informativo[14]. Inoltre, ogni pacchetto ha dei metadati associati: la sua dimensione e una marcatura temporale.

La marcatura temporale (*timestamp*) applicata a ogni pacchetto verifica le seguenti proprietà.

- Il *timestamp* del primo pacchetto è 0.

$$timestamp(P_0) = 0$$

- I *timestamp* sono monotonici non decrescenti.

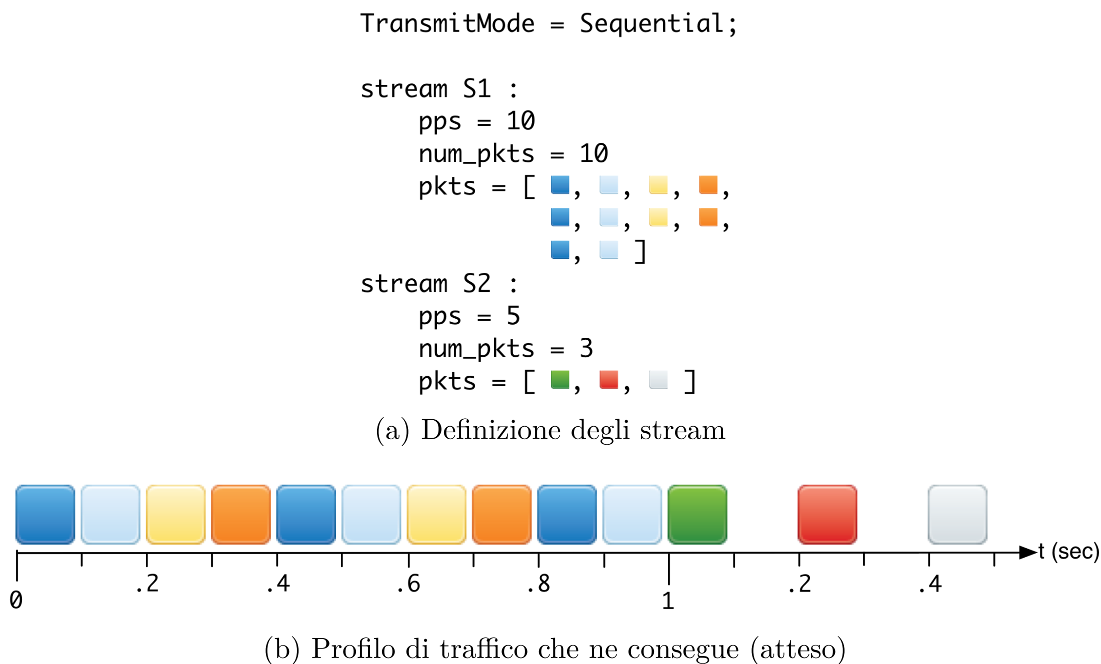
$$\forall P_i \in PacketList, \quad timestamp(P_i) \geq timestamp(P_{i-i})$$

- La differenza fra due *timestamp* successivi corrisponde all'intervallo di tempo fra l'invio dei due pacchetti corrispondenti (chiamato *inter-departure time*, IDT).

$$\forall P_i \in PacketList, \quad timestamp(P_i) - timestamp(P_{i-i}) = IDT(P_i)$$

[13]Per ogni *stream* attivo, `AbstractPort` verifica se lo *stream* è periodico, quindi determina l'insieme minimo di pacchetti che lo rappresentano. La lista dei pacchetti conterrà solo questo insieme minimo di pacchetti, e le indicazioni su quante volte trasmetterlo. Tale indicazione viene fornita tramite la funzione `loopNextPacketSet()`.

[14]In altre parole, la sua costituzione è quella finale, così come sarà inviato sul collegamento, fatto salvo per il codice di ridondanza ciclica (CRC) del protocollo Ethernet, tipicamente calcolato e aggiunto in coda dalla scheda di rete stessa.



(a) Definizione degli stream

(b) Profilo di traffico che ne consegue (atteso)

Figura 2.12: Esempio modello lista, configurazione di esempio

Le marcature temporali vanno intese in termini relativi, non assoluti: non rappresentano l'istante di trasmissione atteso per il pacchetto, ma vanno utilizzate per calcolare il tempo (la distanza temporale) che dovrebbe intercorrere fra un pacchetto e quello precedente per rispettare, in fase di trasmissione, la velocità di invio imposta.

> Nel caso in cui un gruppo di pacchetti abbia la stessa marcatura temporale, ovvero
>
> $$\forall P_i \in PacketGroup, \quad timestamp(P_i) = timestamp(P_{i-i})$$
>
> i pacchetti di tale gruppo sono trasmessi alla massima velocità possibile.

I *timestamp* associati ai pacchetti non tengono infatti conto dei cicli che possono esistere all'interno della lista dei pacchetti da trasmettere, o per l'intera lista stessa; per questo motivo i *timestamp* non rappresentano l'istante di trasmissione atteso — anche se sarebbe possibile determinarlo con dei semplici calcoli a partire dalle marcature temporali stesse e dalla conoscenza della struttura di questi cicli.

Esempio riepilogativo

Considerato quanto spiegato finora, concludere con un esempio riepilogativo è sicuramente d'aiuto. Si farà riferimento alla configurazione mostrata in Figura 2.12, che definisce due *stream* da trasmettere in modalità sequenziale, nella quale i pacchetti da inviare sono rappresentati con dei *box* colorati: a colori diversi corrisponde un contenuto differente.

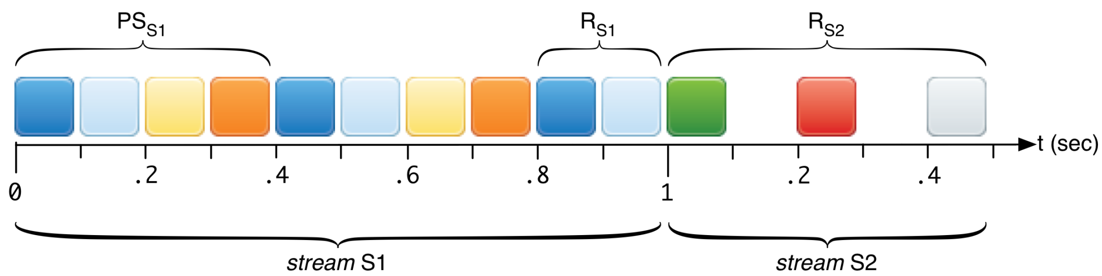


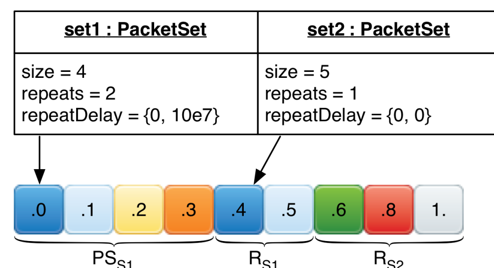
Figura 2.13: Esempio modello lista, decomposizione del profilo atteso

```
clearPacketList();

loopNextPacketSet(4, 2, 0, 100000000);
appendToPacketList(0, 000000000, ■, …);  } PS_S1
appendToPacketList(0, 100000000, ■, …);
appendToPacketList(0, 200000000, ■, …);
appendToPacketList(0, 300000000, ■, …);

appendToPacketList(0, 400000000, ■, …);  } R_S1
appendToPacketList(0, 500000000, ■, …);
appendToPacketList(0, 600000000, ■, …);  } R_S2
appendToPacketList(0, 800000000, ■, …);
appendToPacketList(1, 000000000, ■, …);
```

(a) Configurazione tramite l'interfaccia di `AbstractPort`



(b) Rappresentazione secondo il modello definito da `AbstractPort`

Figura 2.14: Esempio modello lista, rappresentazione secondo `AbstractPort`

Il primo *stream*, chiamato $S1$, è composto da 10 pacchetti da trasmettere alla velocità di 10 pacchetti al secondo e ha una struttura periodica. I primi quattro pacchetti ne costituiscono il periodo minimo, e fanno quindi parte della lista PS_{S1}, da trasmettere 2 volte; gli ultimi due pacchetti fanno invece parte dell'altra lista, R_{S1}.

Il secondo *stream*, $S2$, è formato da 3 soli pacchetti da trasmettere alla velocità di 5 pacchetti al secondo. Non avendo una struttura periodica, tutti i suoi pacchetti fanno parte della lista R_{S2}. La Figura 2.13 illustra il profilo di traffico atteso così decomposto.

La rappresentazione, secondo il modello definito da `AbstractPort`, della lista dei pacchetti da trasmettere derivata da tali *stream* è quindi quella mostrata in Figura 2.14.

In particolare, la Figura 2.14a riporta la sequenza di funzioni invocate da `AbstractPort` durante la traduzione degli *stream* nella corrispondente lista di pacchetti da trasmettere: la funzione `loopNextPacketSet()` è chiamata una sola volta, per impostare il numero di ripetizioni per la sequenza di pacchetti che formano PS_{S1}; gli altri pacchetti, in pratica, vengono considerati parte di un'unica sequenza, da trasmettere una sola volta.

In Figura 2.14b, le etichette applicate ai pacchetti (i *box* colorati) ne rappresentano le marcature temporali corrispondenti; `repeatDelay` è espresso con due valori: i secondi e i nanosecondi di attesa fra due ripetizioni della stessa sequenza.

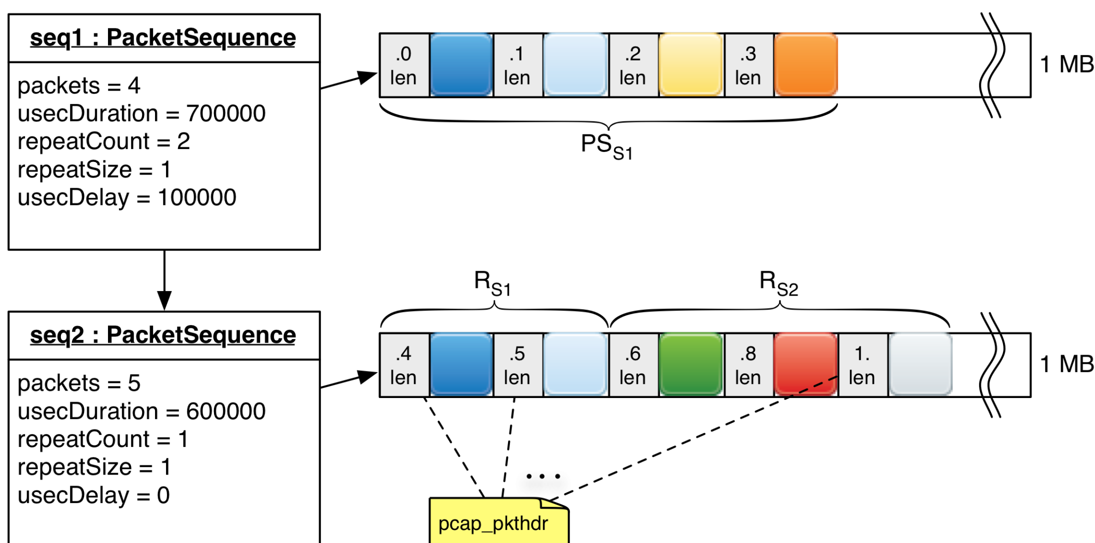


Figura 2.15: Esempio modello lista, rappresentazione secondo `PcapPort`

2.2.5 La trasmissione dei pacchetti secondo `PcapPort`

In questa sezione si parlerà solo della classe `PcapPort`, in quanto le sue specializzazioni (`LinuxPort`, `BsdPort` e `WinPcapPort`) non apportano modifiche per quanto riguarda la gestione e la trasmissione su richiesta dei pacchetti.

Ogni oggetto di classe `PcapPort` è responsabile di una scheda di rete e gestisce la propria lista di pacchetti da trasmettere. La struttura di questa lista è compatibile con il modello definito da `AbstractPort`, descritto nella sezione precedente (sottosezione 2.2.4).

In `PcapPort`, i pacchetti da trasmettere sono memorizzati in blocchi di memoria (*chunk*) di dimensione fissa (1 MByte ognuno), organizzati a lista. Ogni blocco, che prende il nome di `PacketSequence`, contiene un numero variabile di pacchetti; ognuno di questi pacchetti è preceduto dai relativi metadati (marcatura temporale e dimensione, in un formato specifico di `libpcap`, ovvero `struct pcap_pkthdr`).

L'insieme minimo di pacchetti per uno *stream* può essere distribuito su una o più `PacketSequence`: dipende dalla sua occupazione di memoria.

La lista dei pacchetti da trasmettere è quindi rappresentata da una lista di `PacketSequence` e delle strutture addizionali che determinano come si itera su ogni `PacketSequence` o su un loro gruppo (più `PacketSequence` contigue), in base al valore di `repeatSize` — ogni *chunk*, o una loro successione, può essere trasmesso più volte prima di passare a quello successivo. La Figura 2.15 ne dà una rappresentazione, con riferimento all'esempio introdotto nella sezione precedente.

Per ogni oggetto di classe `PcapPort`, un *thread* gestisce la trasmissione dei pacchetti su richiesta; questi sono inviati con la funzione `pcap_sendpacket()` della libreria `pcap`. Un *thread* distinto si occupa della cattura dei pacchetti e del loro salvataggio su *file*, sempre su richiesta.

Gli eventuali tempi di attesa prima della trasmissione di ogni pacchetto, necessari per per rispettare la velocità di invio richiesta, sono calcolati sulla base

delle marcature temporali dei pacchetti e l'orologio di sistema.

I *timestamp* sono usati in maniera relativa: per calcolare la distanza temporale che dovrebbe esistere rispetto al pacchetto precedente.

All'inizio della trasmissione di una `PacketSequence` il trasmettitore registra il valore dell'orologio di sistema; i *timestamp* sono quindi valutati relativamente a questo primo pacchetto della `PacketSequence` corrente. Confrontando il tempo trascorso dall'invio del pacchetto precedente con la distanza temporale prevista (calcolata dai *timestamp*), il trasmettitore determina il tempo di attesa (o l'ammontare del ritardo) per quel pacchetto.

Ne consegue che l'algoritmo implementato da `PcapPort` richiede due interrogazioni dell'orologio (ovvero, due chiamate di sistema per la sola temporizzazione) per ogni pacchetto, anche quando il trasmettitore è in ritardo.

2.2.6 La gestione delle statistiche

La classe `PcapPort` e le sue specializzazioni gestiscono le statistiche in modo leggermente diverso.

Ogni oggetto di classe `PcapPort` utilizza due *thread* per il monitoraggio delle statistiche: uno per ogni direzione del traffico (ricezione e trasmissione). Ognuno di questi *thread* utilizza la funzione `pcap_setdirection()` della libreria `pcap`, per indicare a quale direzione del traffico è interessato, quindi invoca incessantemente la funzione `pcap_next_ex()` per recuperare ogni singolo pacchetto transitato sulla scheda di rete in quella direzione, aggiornando le statistiche di conseguenza. Al momento, sono considerate solo le statistiche relative ai pacchetti e Byte ricevuti e trasmessi (solo i loro valori assoluti, non quelli al secondo).

Su Windows (`WinPcapPort`) è adottata una strategia simile, con la differenza che sono calcolati anche i valori al secondo del numero di pacchetti e Byte, trasmessi e ricevuti.

`BsdPort`, invece, utilizza un approccio diverso da quello spiegato finora: esiste un solo *thread* che si occupa del monitoraggio delle statistiche per tutte le porte di quel sistema, in entrambe le direzioni. Quest'unico *thread* interroga periodicamente (ogni secondo) la tabella di routing, utilizzando una `sysctl()` di tipo `CTL_NET.PF_ROUTE`, e utilizza le informazioni da essa estratte per aggiornare le statistiche.

Anche in `LinuxPort` esiste un unico *thread* che si occupa dell'aggiornamento delle statistiche per tutte le porte, in entrambe le direzioni. Ogni secondo, viene letto il contenuto di `/proc/dev/net`, oppure si utilizzano le statistiche di `netlink`, a seconda di quale sistema è disponibile.

Capitolo 3

Adattamento con netmap

Ostinato usa la libreria `pcap` per la trasmissione e la cattura dei pacchetti. Le prestazioni raggiungibili da questa libreria non sono però sufficienti se paragonate ai volumi di traffico che possono transitare sulle reti ad alta velocità (milioni di pacchetti al secondo). La generazione di traffico ad alta velocità richiede l'utilizzo di sistemi più efficienti. In questo lavoro è stato adottato il *framework* netmap.

È possibile seguire due strade per utilizzare netmap in Ostinato: una strada è l'emulazione della libreria `pcap`; l'altra, quella di utilizzare direttamente l'API[1] fornita da netmap.

Netmap mette infatti a disposizione una libreria, chiamata `libnetmap`, che espone la stessa interfaccia *software* della libreria `pcap` e ne traduce le chiamate nelle corrispondenti operazioni realizzate secondo l'API nativa di netmap. La libreria `libnetmap` va a sostituire la libreria `pcap`: per utilizzare questo approccio è necessario invocare il *linker* sul codice oggetto[2] di `drone` (la componente *server* di Ostinato) per creare un'unità eseguibile collegata a questa libreria di emulazione.

L'alternativa consiste appunto nell'estendere il codice sorgente di Ostinato in modo da usare l'API nativa di netmap direttamente: al posto della libreria `pcap`. In questo lavoro di tesi è stato seguito tale approccio.

Nel seguito del capitolo verranno presentate le motivazioni legate a questa scelta, l'impatto a livello d'architettura che ne consegue, e gli aspetti rilevanti in termini progettuali e implementativi. Si suppone che il lettore sia familiare con l'organizzazione interna di Ostinato, presentata nella sezione 2.2.

3.1 Motivazioni e considerazioni progettuali

In generale, dato un programma che utilizza la libreria `pcap`, l'emulazione di questa libreria può essere una strategia rapida e conveniente per valutare le prestazioni di tale applicativo quando usato su sottosistemi di rete più efficienti di quello tradizionale (per inviare/ricevere pacchetti attraverso una o più interfacce

[1] API, *Application Programming Interface*; in italiano, interfaccia di programmazione di un'applicazione.

[2] La traduzione del codice sorgente in linguaggio macchina.

di rete). Nel caso di Ostinato, ad esempio, questo approccio è stato tentato in Deri [3] con il solo scopo di stabilire la massima velocità trasmissiva raggiungibile utilizzando PF_RING DNA[3] (il lavoro non tiene conto delle questioni relative all'accuratezza del processo di generazione, fissata la velocità trasmissiva).

A livello pratico, però, possono poi emergere una serie di problematiche che rendono questo metodo infruttuoso, o insoddisfacente. Ostinato fa parte di questa casistica in quanto le sue prestazioni (in termini di pacchetti generabili al secondo e accuratezza di tale processo) sono comunque limitate da una serie di fattori, qui presentati.

Il modello adottato da `drone` prevede che i pacchetti da trasmettere siano preparati in anticipo (prima che la trasmissione abbia inizio), e questo è positivo perché esclude una possibile sorgente di interferenza o di rallentamento durante la trasmissione vera e propria. L'algoritmo di trasmissione implementato da `PcapPort` richiede però due interrogazioni dell'orologio (ovvero, due chiamate di sistema legate alla temporizzazione) per ogni pacchetto, anche quando il trasmettitore è in ritardo. Questo rappresenta un collo di bottiglia significativo, che va necessariamente rimosso.

Inoltre, i *timestamp* associati ai pacchetti sono calcolati con risoluzione del microsecondo, mentre il valore dell'orologio di sistema viene letto con risoluzioni differenti sui diversi sistemi operativi supportati da Ostinato.

- Su Windows, è usata la funzione `QueryPerformanceCounter()`, che offre la risoluzione del nanosecondo.
- Sugli altri sistemi, viene invocata la funzione `gettimeofday()`, che invece offre una risoluzione del microsecondo.

Per velocità trasmissive superiori al milione di pacchetti al secondo è necessario impiegare *timestamp* e *timer* con risoluzioni del nanosecondo.

Anche la tecnica di attesa utilizzata per rispettare la velocità trasmissiva richiesta cambia in base al sistema operativo.

- Su Windows e Linux viene adottata la tecnica delle attese attive.
- Su FreeBSD e Mac OS X, il *thread* trasmettitore si sospende invocando una `usleep()`.

L'attesa attiva è una tecnica fondamentale e imprescindibile per ottenere buone accuratezze nel momento in cui si voglio trasmettere più di 50/100 pacchetti al secondo.

Infine, in `PcapPort` il codice del motore di trasmissione è complicato dal dover navigare sulla lista dei pacchetti da trasmettere. Il *thread* di trasmissione deve conoscere l'organizzazione interna della lista; è lui stesso ad occuparsi, durante la trasmissione, della ricostruzione degli *stream* dati gli insiemi minimi dei pacchetti

[3] PF_RING DNA: una soluzione che affronta in modo alternativo gli stessi problemi risolti da netmap, http://www.ntop.org/products/pf_ring/dna/

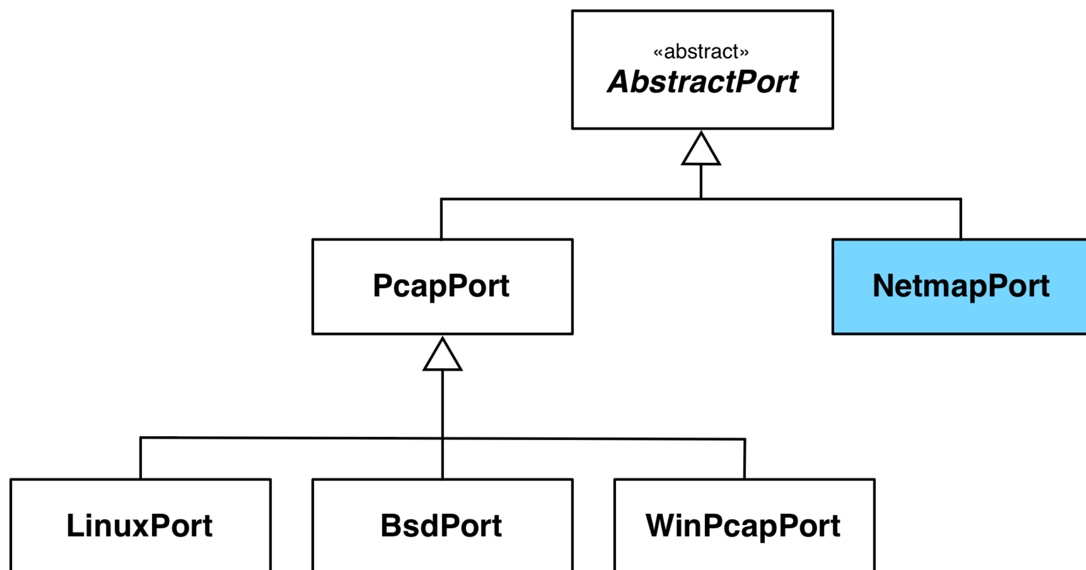


Figura 3.1: La nuova classe `NetmapPort` nella gerarchia di `AbstractPort`

che li rappresentano: tiene conto dell'esistenza di cicli all'interno della lista, o per la lista stessa, e maneggia i *timestamp* di conseguenza.

Quindi, nel caso di Ostinato l'emulazione della libreria `pcap` non costituisce, da sola, una strada percorribile per raggiungere gli obiettivi prefissati per questo lavoro di tesi: elevate velocità trasmissive, dell'ordine dei milioni di pacchetti per secondo, e la riproduzione accurata del profilo di traffico richiesto dall'utente. È infatti necessario intervenire anche sul motore di generazione dei pacchetti e le strutture dati da esso utilizzate.

Dovendo intervenire sul codice di Ostinato, diventa naturale introdurre il supporto nativo a netmap, in quando ne derivano ulteriori benefici in termini di versatilità: oltre alla libertà di adottare un algoritmo trasmissivo *ad-hoc*, non vengono compromesse funzionalità esistenti (potendo utilizzare netmap congiuntamente a `libpcap`), si può impiegare Ostinato anche su porte VALE [21; 22] ed è possibile sfruttare le caratteristiche *hardware* delle moderne schede di rete multi-coda.

3.2 Impatto a livello d'architettura

Per introdurre il supporto nativo all'API di netmap è necessario estendere il codice di Ostinato. Questa sezione descrive come e dove si collocano, a livello architetturale, le modifiche che è necessario apportare.

Per supportare un nuovo tipo di porta è sufficiente intervenire sul codice sorgente di `drone`[4], introducendo una nuova derivazione della classe `AbstractPort` e modificando il `PortManager` di conseguenza.

Questa nuova classe è stata chiamata `NetmapPort` (Figura 3.1), e viene usata per gestire le porte in modalità netmap, siano esse adattatori reali o interfac-

[4] Il codice di `drone` è reperibile nella cartella `server/` dei sorgenti di Ostinato.

ce VALE. Essa svolge funzioni analoghe a quelle espletate da `PcapPort`, utilizzando però l'API netmap (invece della libreria `pcap`) per tutte le operazioni di trasmissione e ricezione dei pacchetti sulla porta controllata.

Il `PortManager` identifica tutti gli adattatori di rete presenti sull'elaboratore locale e istanzia un oggetto per ognuno degli adattatori trovati, responsabile della sua gestione. Questo avviene all'avvio di `drone`, utilizzando la funzione `pcap_findalldevs()` della libreria `pcap`. Dato che le porte VALE sono create dinamicamente, è necessario indicare al `PortManager` quando e come crearle. Inoltre, per gli adattatori reali si richiede di poter stabilire se utilizzarli in modalità normale o in modalità netmap.

Il `PortManager` è stato quindi esteso e istruito per istanziare oggetti di classe `NetmapPort`. Le porte da utilizzare in modalità netmap sono elencate mediante una opzione dalla riga di comando. Un'altra opzione permette di stabilire la dimensione massima dei *burst* prodotti dalle porte in modalità netmap[5].

Come spiegato nella sottosezione 2.2.3, la classe `NetmapPort` ha le seguenti responsabilità.

- Definire e implementare le strutture dati che memorizzano la lista dei pacchetti da trasmettere, da popolare secondo l'interfaccia specificata da `AbstractPort`.
- Trasmettere su richiesta i pacchetti di questa lista, definendo e implementando un opportuno motore di trasmissione per gli stessi.
- Catturare su richiesta i pacchetti, salvandoli su *file* `pcap`.
- Calcolare e collezionare le statistiche dei pacchetti trasmessi e ricevuti.
- Definire una modalità per il controllo esclusivo della porta.

Ognuno di questi punti sarà affrontato e descritto nelle sezioni che seguono, dove si spiegherà anche come sono stati risolti i limiti di Ostinato discussi alla sezione precedente (sezione 3.1) — le parti relative alla gestione delle statistiche e al salvataggio su *file* `pcap` dei pacchetti non sono sviluppate con sezioni a sé stanti, bensì saranno trattate nella sezione 3.7.

3.3 Gestione delle marcature temporali

Per supportare velocità trasmissive dell'ordine dei milioni di pacchetti al secondo è necessario adottare *timestamp* e *timer* con risoluzione del nanosecondo; inoltre, il *thread* che si occupa dell'invio dei pacchetti dovrebbe impiegare le attese attive, piuttosto che la sospensione, quando necessita di rallentare per rispettare i limiti di velocità imposti.

La classe `TimeStamp` è un tipo di dato astratto sviluppato nell'ambito di questo lavoro di tesi proprio per incapsulare quelle operazioni su valori temporali

[5] Il lavoro di integrazione delle funzionalità relative a netmap con l'interfaccia grafica di Ostinato è lasciato come parte degli sviluppi futuri di questo lavoro (vedi Capitolo 5).

necessarie a un generatore di pacchetti per reti ad alta velocità. Gestisce *timestamp* e *timer* con risoluzione del nanosecondo e offre una interfaccia conveniente per realizzare operazioni aritmetiche, confronti e conversioni di formato.

I dati sono memorizzati come in una `struct timespec`, mentre l'orologio di sistema viene interrogato con metodi che dipendono dal sistema operativo, in modo da ottenere valori con risoluzione del nanosecondo.

- Su Windows, è usata la funzione `QueryPerformanceCounter()`.
- Su Linux e FreeBSD, viene invocata la funzione `clock_gettime()`.
- Su Mac OS X, che non dispone della funzione `clock_gettime()`, si utilizza la funzione `mach_absolute_time()`.

La classe mette anche a disposizione dei metodi statici per realizzare attese attive di una certa durata, o fino a un istante temporale stabilito.

3.4 La lista dei pacchetti da trasmettere

La struttura dati utilizzata da `NetmapPort` per gestire la lista dei pacchetti da trasmettere è compatibile con il modello definito da `AbstractPort` (descritto nella sottosezione 2.2.4) ed è stata progettata in modo da semplificare al massimo il codice del motore di trasmissione: tutta la complessità legata alla creazione e alla navigazione della lista dei pacchetti secondo tale struttura è infatti celata dietro la sua interfaccia. Come avviene per `PcapPort`, la lista dei pacchetti da trasmettere è popolata dagli algoritmi forniti da `AbstractPort`; a differenza di `PcapPort`, scorrere la lista in fase di trasmissione è semplice come incrementare un iteratore.

Tale ristrutturazione della lista dei pacchetti è rappresentata dalla classe `PacketList`. Essa gestisce un *buffer* lineare contenente i pacchetti da trasmettere, e delle strutture dati ausiliari che memorizzano i metadati associati ai pacchetti (dimensione, marcatura temporale, *inter-departure time* con il pacchetto precedente) e la definizione dei vari cicli su sequenze di questi pacchetti[6]. La Figura 3.2 mostra l'organizzazione secondo `PacketList` della lista dei pacchetti da trasmettere per l'esempio trattato nella sottosezione 2.2.4 (cfr. Figura 2.12).

La classe *template* `DataBuffer` implementa il *buffer* lineare contenente i pacchetti da trasmettere, costituito da un unico *chunk* di memoria (realizzato mediante un `std::vector` — i dati si trovano quindi in un'area contigua nello *heap*)[7].

[6] Il modello definito da `AbstractPort` per la lista dei pacchetti da trasmettere permette infatti di memorizzare solo quel sottoinsieme minimo di pacchetti che rappresenta l'intera lista. Questo è possibile perché gli *stream* configurati su una porta possono essere periodici: si pensi ad esempio a uno *stream* costituito da un certo numero di pacchetti, tutti uguali, inviati a velocità costante.

[7] Modificare l'implementazione per suddividere il *buffer* in più *chunk* è semplice. Esistono due vincoli principali: (i) ogni *chunk* deve essere costituito da un'area di memoria contigua, e (ii) un pacchetto non può essere memorizzato a cavallo di due *chunk*. Questi vincoli derivano dalla scelta progettuale di mantenere un *layout* di memoria compatibile con quello definito

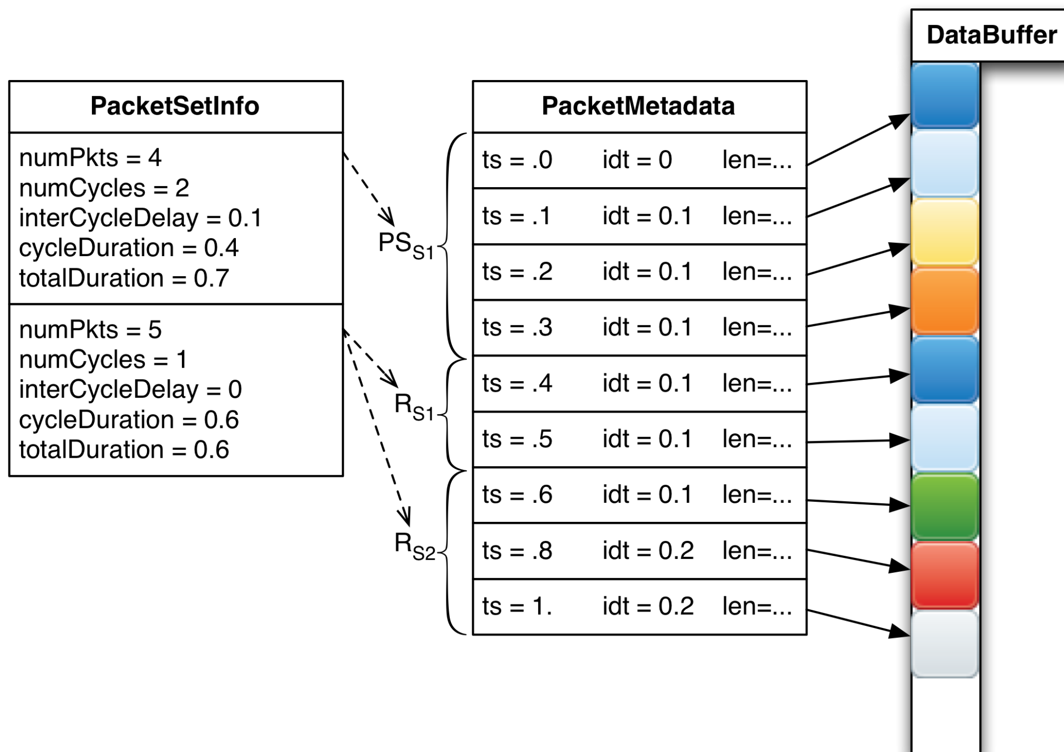


Figura 3.2: Esempio modello lista, rappresentazione secondo `PacketList`

La classe permette l'accesso diretto a ogni singolo elemento (pacchetto) nel buffer, mediante una funzione indice (`operator[]`) che restituisce un riferimento all'elemento indicato; il pacchetto può quindi essere maneggiato da una funzione `C` senza bisogno di copie di memoria — tale funzione può leggere il contenuto dell'area di memoria, e modificarla nel contenuto ma non nelle dimensioni.

L'interfaccia per popolare la `PacketList`, riportata nel Listato 3.1, è analoga a quella definita da `AbstractPort`. I pacchetti inseriti tramite la funzione `append()` vengono copiati nel `DataBuffer` interno all'oggetto di classe `PacketList` su cui la funzione è stata invocata.

Listato 3.1: L'interfaccia per popolare la `PacketList`

```
void reset();
void setLoopMode(bool doLoop = false, const TimeStamp &loopDelay = ↩
   TimeStamp());
bool append(const TimeStamp &ts, const uint8_t *packet, int length);
void nextPacketSet(int64_t size, int64_t repeats, const TimeStamp ↩
   &repeatDelay);
```

dal linguaggio `C` per gli *array* [6], in modo da consentire alle funzioni `C` l'accesso diretto a ogni singolo pacchetto nel *buffer*. Inoltre, in caso vengano aggiunti alla lista nuovi pacchetti, puntatori e riferimenti ai dati esistenti non devono essere invalidati. Un contenitore C++ STL adatto a questo scopo è `std::deque` su elementi di tipo `std::array` [8–10].

Navigare sulla `PacketList` è semplice; l'interfaccia da utilizzare è riportata nel Listato 3.2 e di seguito descritta.

Listato 3.2: L'interfaccia per iterare sulla `PacketList`

```
struct PacketInfo {
    TimeStamp ts; // inter-departure time rispetto al primo pacchetto
    const uint8_t *data; // puntatore al contenuto del pacchetto
    std::size_t len; // dimensione del pacchetto
    TimeStamp idt; // inter-departure time rispetto al pacchetto precedente
};

bool empty() const;
std::size_t size() const;

bool peek_first(PacketInfo &pktInfo);
bool first(PacketInfo &pktInfo);

bool has_next() const;
bool peek_next(PacketInfo &pktInfo, bool tsManip = false);
bool next(PacketInfo &pktInfo, bool tsManip = false);
```

`empty()` Ritorna il valore `vero` se la lista è vuota, altrimenti `falso`.

`size()` Ritorna il numero dei pacchetti nella lista, ovvero quelli che costituiscono una iterazione su di essa.

`peek_first()` Recupera il primo pacchetto della lista, senza avanzare l'iteratore sul secondo pacchetto.

`first()` Come la precedente, ma avanza l'iteratore sul secondo pacchetto.

`has_next()` Ritorna il valore `vero` se la lista ha un altro pacchetto, altrimenti `falso`.

`peek_next()` Recupera il prossimo pacchetto in lista, senza avanzare l'iteratore sul pacchetto successivo.

`next()` Come la precedente, ma avanza l'iteratore sul pacchetto successivo.

Il *thread* che si occupa della trasmissione può iniziare l'iterazione sulla lista dei pacchetti da trasmettere invocando la funzione `peek_first()`, oppure la funzione `first()`. Le informazioni rilevanti per i pacchetti successivi sono recuperabili mediante le funzioni `peek_next()` e `next()`, mentre il metodo `has_next()` può essere usato per controllare se tutti i pacchetti sono stati trasmessi o meno. Metadati e contenuto di ogni pacchetto sono incapsulati in oggetti di tipo `struct PacketInfo`.

È la `PacketList` a occuparsi di ogni aspetto legato alla rappresentazione interna della lista dei pacchetti da trasmettere. Il *thread* che si occupa della

trasmissione non ha bisogno di conoscere l'organizzazione interna della lista, gli è sufficiente incrementare un iteratore per scorrerla. La `PacketList`, inoltre, aggiusta le marcature temporali dei pacchetti recuperati in modo che i valori di questi *timestamp* corrispondano alla distanza temporale prevista rispetto al primo pacchetto della lista, mentre la differenza fra due *timestamp* successivi corrisponde sempre all'*inter-departure time* dei pacchetti corrispondenti.

3.5 L'algoritmo di trasmissione dei pacchetti

Il profilo di traffico prodotto dal generatore deve essere il più possibile conforme a quanto richiesto e atteso dal suo utilizzatore. Dimensioni e contenuto dei pacchetti sono facilmente riproducibili con precisione; più difficile è, invece, rispettare i vincoli temporali (la velocità trasmissiva e la distribuzione di probabilità degli intertempi fra pacchetti successivi) a causa delle interferenze *software* legate sia alla presenza di altri processi in esecuzione sul medesimo elaboratore, sia alle varie attività svolte dal generatore stesso. Nella definizione del motore di trasmissione dei pacchetti è quindi necessario valutare con attenzione tutte le possibili sorgenti di interferenza e il costo computazionale delle operazioni svolte.

Deviazioni impredicibili sul tempo di esecuzione sono principalmente causate dalle sospensioni, specie quando a queste conseguono commutazioni di contesto; è quindi importante limitare l'acquisizione di *lock* su oggetti e impiegare le attese attive. Operazioni costose sono le copie di memoria e le chiamate di sistema; dato che non è possibile eliminare completamente questo tipo di operazioni, ammortizzarle su più pacchetti alla volta si rivela spesso una strategia vincente, quando applicabile.

L'algoritmo di trasmissione dei pacchetti implementato da `NetmapPort` è mostrato nel Listato 3.3. Gran parte della complessità delle operazioni è celata dalle interfacce delle classi `TimeStamp` e `PacketList`, discusse nelle sezioni precedenti, e questo rende il codice dell'algoritmo snello e di semplice comprensione.

Listato 3.3: L'algoritmo di trasmissione dei pacchetti usato da `NetmapPort`

```
// Are there packets for transmission?
if (pktList.peek_first(pkt) == false) {
    return;
}
tsBase = pkt.ts;

// OK, let's start transmission!
timeBase = prevTxTime = startTxTime.now();
while (!stop && pktList.next(pkt, true)) {
    // (pkt.idt == 0) ==> TX immediately, without looking at current time: ↩
        you're asked to go as fast as you can.
    // (delay > 0) ==> you're in delay, so TX immediately packets that ↩
        should be already TX.
    if ((pkt.idt > tsZero) && (delay - pkt.idt) < tsZero) {
        if (burstSize > 0) {
```

```
            netmap_flush(nmd);
            burstSize = 0;
        }
        // side-effect: currTime is updated
        waitTime = (pkt.ts - tsBase) - (currTime.now() - timeBase);
        if (waitTime > tsZero) {
            // curTime: inout arg; delay: out arg.
            TimeStamp::activeWaitFor(waitTime, &currTime, &delay, ↩
                false);
        } else {
            // You're in delay.
            delay = -waitTime;
        }
        prevTxTime = currTime;
    } else {
        // IDT recovery mode: TX the packet immediately, trying to recover ↩
            the delay
        delay -= pkt.idt;
    }

    sent = nm_inject(nmd, pkt.data, pkt.len);
    if (!sent) {
        netmap_flush(nmd);
        burstSize = 0;

        // Skip stats update
        continue;
    }

    ++burstSize;
    if (burstSize >= maxBurstSize) {
        netmap_flush(nmd);
        burstSize = 0;
    }

    nmTxStats->txPkts++;
    nmTxStats->txBytes += pkt.len;
}

// Flush pending packets, if any
if (burstSize > 0) {
    netmap_flush(nmd);
    burstSize = 0;
}
```

L'oggetto `pktList`, di classe `PacketList`, contiene la lista dei pacchetti da trasmettere. Ogni pacchetto `pkt` estratto da questa lista è un oggetto di tipo `struct PacketInfo` e racchiude sia il pacchetto (o meglio, un puntatore all'area di memoria che lo contiene) sia i suoi metadati: dimensione, marcatura temporale

e *inter-departure time* con il pacchetto precedente.

Gli oggetti `tsBase` e `timeBase`, entrambi di classe `TimeStamp`, contengono rispettivamente il *timestamp* del primo pacchetto in lista, e l'orario di inizio della trasmissione. Della stessa classe sono l'oggetto costante `tsZero`, inizializzato con il valore temporale `0`, e l'oggetto `delay`, il quale indica con valori positivi l'ammontare del ritardo accumulato rispetto al profilo richiesto e con valori negativi di quanto si è in anticipo.

Per l'invio dei pacchetti si utilizza il *framework* netmap. In particolare, la funzione `nm_inject()` copia il pacchetto nella coda (*ring*) di trasmissione della scheda di rete, mentre la funzione `netmap_flush()` invoca la chiamata di sistema `ioctl(..TXSYNC..)` che dà luogo all'effettiva trasmissione dei pacchetti accodati sull'adattatore di rete in uso.

L'algoritmo cerca di trasmettere un pacchetto alla volta, in modo da garantire la massima accuratezza possibile. Quando la velocità trasmissiva richiesta è troppo elevata per questa modalità di funzionamento, ovvero si sta accumulando ritardo, il trasmettitore invia i pacchetti in blocchi (*burst*): più pacchetti vengono accodati tramite la funzione `nm_inject()`, quindi sono trasmessi invocando una sola volta la funzione `netmap_flush()` — pertanto, il costo delle chiamate di sistema associate è ammortizzato su più pacchetti.

La dimensione dei *burst* (variabile `burstSize`) è dinamica, adattata sull'ammontare del ritardo accumulato: l'algoritmo cerca di ridurre il più possibile il numero di pacchetti per *burst*, sempre per ragioni di accuratezza (far sì che la distribuzione degli intertempi fra i pacchetti sia vicina a quella attesa).

Il numero massimo di pacchetti per *burst* è regolato dal valore della variabile `maxBurstSize`, configurabile all'avvio di `drone` mediante un'apposita opzione dalla riga di comando. Il valore di default è 252 pacchetti.

Quando il trasmettitore determina invece di essere in anticipo, attende di poter riprendere la trasmissione dei pacchetti. Il metodo statico `activeWaitFor()` della classe `TimeStamp` realizza una attesa attiva di durata specificata (argomento `waitTime`) a partire dal valore di `currTime`.

$$targetTime = currTime + waitTime$$

Il valore di `currTime` e dell'argomento `delay` sono aggiornati rispettivamente con il valore dell'orario al termine dell'attesa attiva, e l'eventuale surplus sulla durata d'attesa. L'ultimo argomento passato alla funzione, di valore `false`, indica alla stessa di omettere ogni controllo sui suoi argomenti: in questo caso non è necessario farlo in quanto sono già stati validati dal chiamante.

L'algoritmo adotta una ottimizzazione per recuperare eventuali ritardi accumulati: i pacchetti che a un dato istante dovevano essere già stati inviati[8] vengono trasmessi immediatamente in *burst*, senza leggere l'orologio di sistema.

Un'altra ottimizzazione, basata sulla stessa tecnica adottata al caso precedente, consiste nel trasmettere immediatamente, in *burst*, i pacchetti che hanno un *inter-departure time* pari a zero, senza invocare chiamate di sistema per la lettura

[8] Ogni pacchetto ha una marcatura temporale associata, che indica il suo istante di trasmissione atteso (rispetto all'orario di inizio della trasmissione stessa).

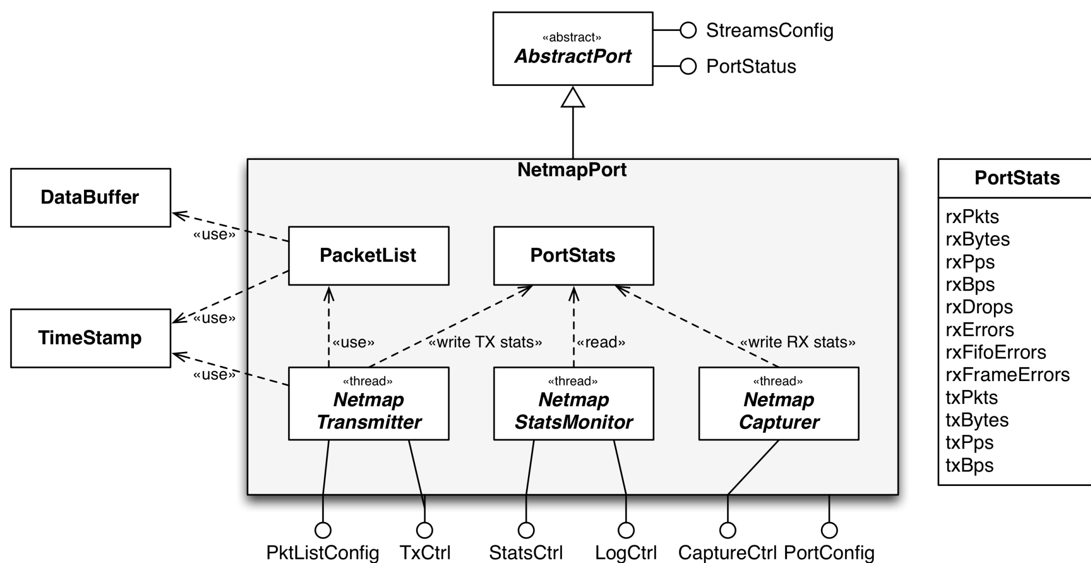


Figura 3.3: La struttura interna di `NetmapPort`

dell'orologio. Questo è infatti il caso in cui al trasmettitore è richiesto di andare più veloce possibile.

Infine, l'algoritmo si occupa dell'aggiornamento delle statistiche sul numero totale di pacchetti e di Byte trasmessi.

3.6 Controllo esclusivo della porta

Ostinato prevede la possibilità di marcare una porta a suo uso esclusivo, impedendone quindi l'accesso al sistema operativo e agli altri programmi in esecuzione sul medesimo calcolatore, e ciò risulta utile per creare un ambiente di *test* controllato.

Al momento, però, l'architettura di netmap non consente di implementare una modalità per il controllo esclusivo di una porta in modalità netmap. Tutte le schede di rete operanti in modalità netmap usano infatti la stessa area di memoria, la quale è accessibile anche a tutti i processi che hanno associato un descrittore di file `/dev/netmap/` a una delle suddette schede di rete — alle porte VALE sono invece associate regioni di memoria distinte.

3.7 Mettendo tutto insieme

Si descriverà ora come la classe `NetmapPort`, la cui struttura interna è illustrata in Figura 3.3, utilizza e combina gli elementi presentati nelle sezioni precedenti[9].

[9]La Figura 3.3 mostra l'esistenza di una interfaccia *software* a cui non si è mai fatto riferimento finora, ovvero `LogCtrl`; questa sarà presentata e descritta nella sottosezione 4.1.3, poiché si tratta di una estensione introdotta per controllare la registrazione delle statistiche di funzionamento, e che risulta utile nella valutazione delle prestazioni di Ostinato.

Per ogni porta in modalità netmap, sia essa un adattatore reale o la porta di uno *switch* VALE, il `PortManager` istanzia un oggetto di classe `NetmapPort` adibito alla sua gestione.

Ogni oggetto di classe `NetmapPort` istanzia a sua volta tre *thread*, ognuno dei quali svolge dei compiti ben precisi. Questi tre *thread*, operando su un oggetto condiviso di classe `PortStats`, collaborano all'aggiornamento delle statistiche di traffico per la porta.

Il *thread* `NetmapTransmitter` gestisce la lista dei pacchetti da trasmettere (una istanza della classe `PacketList`), e si occupa della trasmissione su richiesta di tali pacchetti secondo l'algoritmo descritto nella sezione 3.5, aggiornando le sole statistiche relative ai pacchetti e ai Byte totali trasmessi (`txPkts` e `txBytes`).

Il *thread* `NetmapCapturer` acquisisce i pacchetti ricevuti sulla porta, aggiorna le statistiche relative ai pacchetti/Byte totali ricevuti e agli errori di ricezione, salvando su richiesta il traffico su *file* `pcap`.

Il *thread* `NetmapStatsMonitor`, infine, verifica ogni secondo lo stato del collegamento, e calcola i valori al secondo delle statistiche sui pacchetti/Byte trasmessi (`txPps`, `txBps`) e ricevuti (`rxPps`, `rxBps`).

3.8 Revisione della classe `PcapPort`

Oltre a realizzare l'adattamento con netmap mediante lo sviluppo della classe `NetmapPort`, anche la classe `PcapPort` è stata ristrutturata utilizzando le strutture dati e gli algoritmi presentati in questo capitolo, sempre nell'ottica di risolvere i problemi di cui si è discusso alla sezione 3.1.

Tutta la base di codice esistente legata alla gestione delle marcature temporali e della lista dei pacchetti da trasmettere è stata quindi dismessa in favore delle nuove classi `TimeStamp` e `PacketList`.

Anche il motore di trasmissione dei pacchetti è stato rivisto, adottando un approccio analogo a quello presentato nella sezione 3.5, con la differenza che la libreria `pcap` non consente di ammortizzare i costi delle operazioni di trasmissione su più pacchetti alla volta. Di conseguenza, i benefici legati alla strategia di recupero di ritardi sono più limitati, così come l'effetto delle ottimizzazioni che entrano in gioco quando i pacchetti hanno un *inter-departure time* pari a zero. Il codice del nuovo algoritmo di trasmissione dei pacchetti per `PcapPort` è riportato nel Listato 3.4.

Listato 3.4: Il nuovo algoritmo di trasmissione dei pacchetti per `PcapPort`

```
// Are there packets for transmission?
if (pktList.peek_first(pkt) == false) {
   return;
}
tsBase = pkt.ts;

// OK, let's start transmission!
timeBase = prevTxTime = startTxTime.now();
while (!stop && pktList.next(pkt, true)) {
```

```
    // (pkt.idt == 0) ==> TX immediately, without looking at current time: ↩
        you're asked to go as fast as you can.
    // (delay > 0) ==> you're in delay, so TX immediately packets that ↩
        should be already TX.
    if ((pkt.idt > tsZero) && (delay - pkt.idt) < tsZero) {
        // side-effect: currTime is updated
        waitTime = (pkt.ts - tsBase) - (currTime.now() - timeBase);
        if (waitTime > tsZero) {
            // curTime: inout arg; delay: out arg.
            TimeStamp::activeWaitFor(waitTime, &currTime, &delay, ↩
                false);
        } else {
            // You're in delay.
            delay = -waitTime;
        }
        prevTxTime = currTime;
    } else {
        // IDT recovery mode: TX the packet immediately, trying to recover ↩
            the delay
        delay -= pkt.idt;
    }

    notSent = pcap_sendpacket(handle, pkt.data, pkt.len);
    if (notSent) {
        // Skip stats update
        continue;
    }

    // Stats update
    stats->txPkts++;
    stats->txBytes += pkt.len;
}
```

Nella sezione 4.3 si mostrerà che questa revisione della classe `PcapPort` permette di risolvere tutti i problemi riscontrati nella versione corrente di Ostinato (v0.5.1), analizzati invece nella sezione 4.2.

Capitolo 4

Valutazione delle prestazioni

Le prestazioni dei generatori *software* dipendono da diversi fattori, come le caratteristiche *hardware* dell'elaboratore, il sistema operativo adottato, il carico di lavoro e il volume di traffico a cui è sottoposto il sistema. Alcuni di questi aspetti sono impredicibili, o comunque fortemente variabili nel tempo, e possono influenzare in negativo il processo di generazione sintetica del traffico.

Ciò nonostante, i generatori *software* sono ampiamente utilizzati, sia nella ricerca scientifica, sia dagli addetti ai lavori nel settore delle reti. È quindi importante conoscere e saper valutare le caratteristiche metrologiche della piattaforma *software* impiegata per effettuare esperimenti e misurazioni in una data condizione operativa: qualora il profilo di traffico effettivamente generato non dovesse corrispondere a quello atteso, i risultati e le conclusioni che ne conseguono potrebbero non essere in realtà validi e riproducibili.

Questo capitolo presenta i risultati dell'analisi condotta su Ostinato. Si parlerà innanzitutto della metodologia seguita, descrivendo come sono stati realizzati i vari esperimenti e i criteri d'analisi adottati (sezione 4.1). Si mostrerà quindi il comportamento del generatore nella sua versione ufficiale, ovvero così come distribuito dal suo sviluppatore, evidenziandone problemi e limiti (sezione 4.2). Infine, si offrirà un confronto con i benefici legati ai cambiamenti introdotti nell'ambito di questo lavoro, sia rispetto alla libreria `pcap` (sezione 4.3), sia impiegando netmap per la generazione di traffico ad alta velocità (sezione 4.4).

4.1 Metodologia

Nella sezione 1.3, parlando dei fattori di costo associati all'elaborazione dei pacchetti in sistemi *software* (generatori di traffico, *firewall*, sistemi di monitoraggio, ecc.), si è fatta la distinzione fra costi legati al sistema operativo, e quelli legati all'applicazione considerata. Guardando da un altro punto di vista, nell'ottica cioè di effettuare valutazioni sperimentali sulle prestazioni di tali sistemi di elaborazione, si può parlare di costi per Byte elaborato, e di costi per singolo pacchetto elaborato [23].

Nei moderni *computer* i costi per Byte elaborato possono essere considerati trascurabili: questi sono infatti proporzionali alla banda (cioè, alla velocità) dei

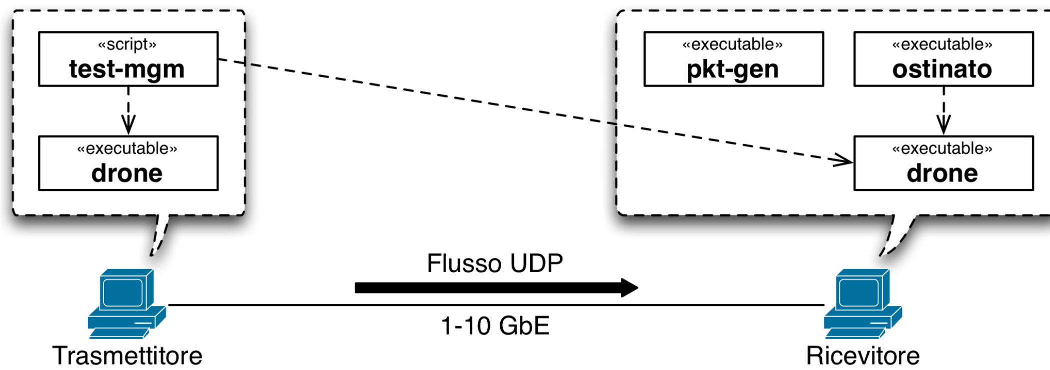


Figura 4.1: Configurazione di base per gli esperimenti

bus di sistema, e i calcolatori recenti sono caratterizzati da processori, memorie e *bus* molto veloci.

I costi per singolo pacchetto elaborato, invece, dipendono fortemente dalle caratteristiche del sistema e dell'applicazione presi in esame.

In generale, il costo di elaborazione dei pacchetti ha una componente fissa, che diventa dominante per pacchetti di piccole dimensioni. Di conseguenza, il caso peggiore per un sistema di elaborazione pacchetti si presenta quanto il traffico da gestire è costituito da pacchetti di dimensione minima (64 Byte per le reti Ethernet). Un simile scenario è spesso anche il modo corretto per valutare le prestazioni del sistema considerato; ovvero, sottoporre il sistema a una tale tipologia di traffico, andando a misurare il numero di pacchetti al secondo che riesce a elaborare (ricevere o trasmettere). Pertanto, nel seguito si farà spesso riferimento a questo criterio di analisi.

4.1.1 Scenari e configurazioni

Le prestazioni di Ostinato sono state valutate su elaboratori con caratteristiche *hardware* differenti e su più sistemi operativi, come sintetizzato nella Tabella 4.1.

In particolare, `fx8350` è stato impiegato soprattutto per la valutazione del comportamento di Ostinato utilizzando la libreria `pcap` sui diversi sistemi operativi supportati dal generatore stesso, con l'esclusione di OS X, per il quale subentra il portatile `mbp9,2`, dalle caratteristiche simili se non leggermente superiori. L'elaboratore `h102`, più potente e dotato di una scheda 10 Gbit Ethernet, è stato per lo più coinvolto negli esperimenti basati su netmap, per la generazione ad alta velocità del traffico; ad ogni modo, per confronto, anche per questo elaboratore verranno riportati i risultati delle misurazioni utilizzando Ostinato e la libreria `pcap`.

Gli esperimenti effettuati sono basati sulla configurazione di Figura 4.1: due *computer* direttamente connessi da un collegamento 1 Gigabit Ethernet o 10 Gigabit Ethernet (a seconda dei casi); una utilizzante Ostinato per la generazione di traffico secondo il profilo definito per l'esperimento, l'altra impiegante Ostinato

Tabella 4.1: Caratteristiche degli elaboratori impiegati per gli esperimenti

fx8350	
Scheda madre	Asus M5A99X EVO R2.0
Processore	AMD FX-8350 @4.00 GHz, octa core, 8 MB cache L3, 2 MB cache L2
Memoria	8 GB RAM DDR3 @2133 MHz (2x 4 GB)
Disco rigido	SSD 256 GB SATA III (OCZ Vertex 4) HDD 500 GB @5400 rpm SATA II (HGST Travelstar 5K750-500)
Schede di rete	1x Gigabit Ethernet (Realtek 8111F)
Sistema operativo	GNU/Linux 3.12.5-1-ARCH x86_64 (Arch Linux) Windows 7 Professional 64-bit SP1 FreeBSD 10.0-RELEASE
mbp9,2	
Modello	MacBook Pro (13-inch, mid 2012)
Processore	Intel Core i5 @2.5 GHz, dual core, 3 MB cache L3, 256 KB cache L2 (per core)
Memoria	16 GB RAM DDR3 @1600 MHz (2x 8 GB)
Disco rigido	SSD 256 GB SATA III (Samsung 830)
Schede di rete	1x Gigabit Ethernet (Broadcom 57765-B0)
Sistema operativo	OS X v10.8.5 (Mountain Lion)
h102	
Scheda madre	Asus P7Q57-M DO
Processore	Intel Core i7 870 @2.93 GHz, quad core, 8 MB cache L3, 1 MB cache L2
Memoria	4 GB RAM DDR3 @1333 MHz (2x 2 GB)
Disco rigido	Seagate Barracuda 1 TB @7200 rpm, cache 32 MB (ST3100528AS)
Schede di rete	1x 10 Gigabit Ethernet SFI/SFP+ (Intel 82599ES rev 01) 1x Gigabit Ethernet (Intel 82578DM rev 06)
Sistema operativo	GNU/Linux 3.12.8-1-netmap x86_64 (Arch Linux)

o `pkt-gen`[1] per misurare le statistiche in ricezione.

L'esecuzione di ogni esperimento è stata orchestrata da uno *script* `Python`, chiamato `test-mgm`, sviluppato nell'ambito di questo lavoro di tesi e impiegato al posto dell'interfaccia grafica[2] per (i) automatizzare la configurazione del trasmettitore[3], (ii) dare l'avvio alla generazione del traffico, (iii) regolare la durata della trasmissione e (iv) coordinare la registrazione delle statistiche lato trasmettitore e, eventualmente, lato ricevitore.

Lo *script* in questione si basa su una *patch*, circolata di recente sulla *mailing-list* del progetto, che introduce in Ostinato il supporto a `Python` [17]. È bene notare che tale (prezioso) contributo riguarda solo l'interfaccia di comunicazione

[1]Per una introduzione su `pkt-gen`, fare riferimento alla sottosezione 1.2.1.
[2]`ostinato`, la componente *client* di Ostinato
[3]`drone`, la componente *server* di Ostinato

fra *client* e *server*: esso non modifica né altera aspetti legati alla generazione sintetica del traffico; la sua adozione è quindi ininfluente dal punto di vista della valutazione delle prestazioni presentata e discussa in questo capitolo.

Nelle varie prove, Ostinato è stato configurato per generare flussi costanti di traffico UDP su IPv4. Ostinato è infatti un generatore di pacchetti ad anello aperto (il profilo del traffico prodotto dipende solo dalla sua configurazione, ovvero non è influenzato da misurazioni sulle prestazioni della rete effettuate durante la trasmissione); inoltre, tutti i pacchetti da trasmettere sono preparati in anticipo, durante la fase di configurazione, prima cioè che la trasmissione abbia inizio (questo vale anche qualora la configurazione preveda pacchetti aventi contenuto variabile nel tempo, ad esempio per realizzare trasmissioni verso più destinazioni). Si è optato quindi per una configurazione semplice (ma comunque significativa) in quanto la complessità del contenuto dei pacchetti non influisce sulle prestazioni del programma.

Per limitare interferenze esterne durante gli esperimenti, i *firewall* sulle varie macchine sono stati configurati in modo da sopprimere eventuali messaggi di errore ICMP, in particolare quelli di tipo 3 (ad esempio, porta di destinazione non raggiungibile). Inoltre, laddove possibile il controllo di flusso Ethernet, sulla scheda di rete del trasmettitore, è stato disabilitato.

4.1.2 Variabili e metriche

I dati riportati in questo capitolo riguardano le statistiche di *trasmissione*. Per la maggior parte degli esperimenti sono state raccolte anche le statistiche lato ricevitore; queste non saranno però qui presentate poiché non sono stati rilevati scostamenti rispetto alle statistiche di trasmissione.

Negli esperimenti relativi alle prestazioni di Ostinato con la libreria `pcap`, l'analisi effettuata punta a evidenziare se il numero di pacchetti al secondo effettivamente prodotto corrisponde al valore richiesto. Sono state effettuate numerose misurazioni, a varie velocità di trasmissione, per alcune dimensioni significative dei pacchetti (64, 512, 1024 e 1518 Byte).

I grafici mostrati nella sezione 4.2 e nella sezione 4.3 confrontano quindi i profili di traffico richiesti con quelli effettivamente prodotti, evidenziando le discrepanze in termini di errore percentuale. Ogni singolo campione nei grafici è stato ottenuto effettuando più rilevazioni, ognuna della durata di 30 secondi.

Nella sezione 4.4, relativa al funzionamento di Ostinato con il *framework* netmap, verrà mostrato anche come la dimensione dei *burst* influenza le prestazioni raggiungibili e verrà dimostrato sperimentalmente il comportamento adattativo dell'algoritmo di trasmissione presentato nella sezione 3.5.

4.1.3 Registrazione delle statistiche

La registrazione delle statistiche lato trasmettitore e lato ricevitore è stata realizzata estendendo l'interfaccia di comunicazione fra *client* e *server* (`OstService`, presentata nella sottosezione 2.2.1).

Le nuove funzioni introdotte, elencate nel Listato 4.1, si affiancano a quelle già esistenti che controllano il processo di cattura e salvataggio dei pacchetti trasmessi e ricevuti su *file* `pcap`; in questo caso vengono però registrate le *statistiche* sui pacchetti e Byte trasmessi e ricevuti su intervalli temporali di durata prefissata (un secondo).

Se l'interfaccia grafica (`ostinato`) mostra le stesse statistiche in tempo reale, questa nuova interfaccia *software* consente di ottenere una registrazione permanente delle statistiche di funzionamento di `drone`, per una successiva analisi, in un modo meno oneroso, dal punto di vista computazionale, della cattura su *file* `pcap`.

Listato 4.1: Estensione dell'interfaccia `OstService` per il *logging*

```
rpc startLogging(PortIdList) returns (Ack);
rpc stopLogging(PortIdList) returns (Ack);
rpc getLoggingBuffer(PortId) returns (LoggingBuffer);
```

L'interfaccia di `AbstractPort` e le implementazioni delle classi derivate sono state quindi similmente estese (vedi Listato 4.2) per supportare questa nuova funzionalità.

Listato 4.2: L'interfaccia per il *logging* esposta dalla classe `AbstractPort`

```
virtual void startLogging();
virtual void stopLogging();
virtual bool isLoggingOn();
virtual QIODevice* loggingData();
```

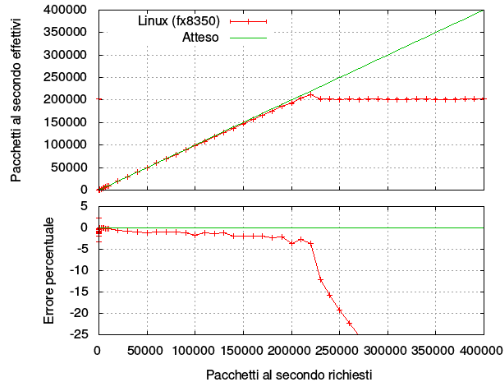
4.2 Implementazione ufficiale di Ostinato

La Tabella 4.2 presenta le prestazioni massime raggiunte dai vari elaboratori utilizzando la versione più recente di Ostinato (v0.5.1)[4], in termini di numero di pacchetti al secondo prodotti al variare della dimensione dei pacchetti trasmessi. Tali risultati sono stati ottenuti imponendo una velocità di trasmissione di zero pacchetti al secondo, che per `drone` equivale a dire "trasmetti alla massima velocità che ti è possibile".

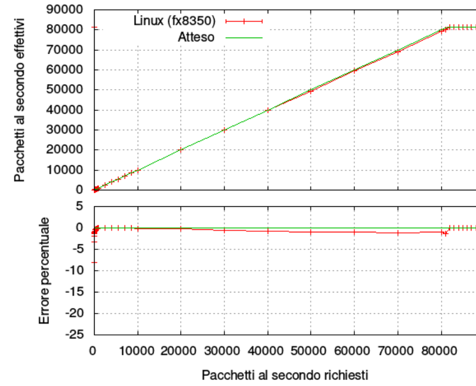
Per pacchetti piccoli, la percentuale di utilizzo del collegamento è decisamente bassa, ben lontana dalla banda (limite fisico) del collegamento stesso. Solo un elaboratore potente come `h102` riesce a produrre un volume di traffico intorno al Gbps, un valore comunque esiguo rispetto alla banda disponibile su un collegamento 10 Gbit Ethernet.

Con pacchetti grandi, invece, i vari sistemi arrivano alla saturazione del collegamento a 1 Gbps, e `h102` si avvicina ai 10 Gbps di traffico.

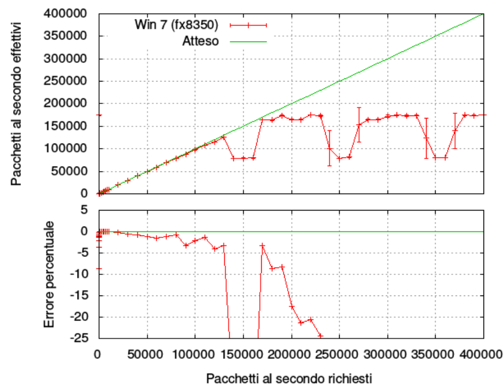
[4]Rispetto alla versione ufficiale, sono state apportate solo modifiche marginali (descritte nella sezione precedente), relative alla registrazione delle statistiche e al supporto dell'interfaccia di *scripting* `Python` nelle comunicazioni fra *client* e *server*.



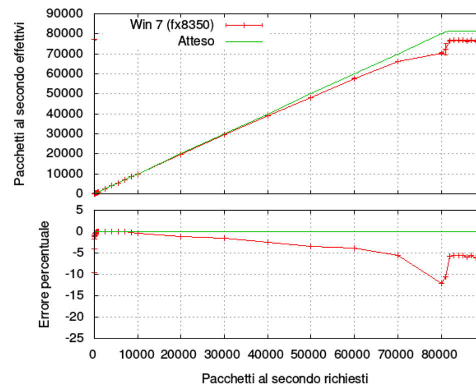
(a) Pacchetti 64 Byte (caso peggiore)



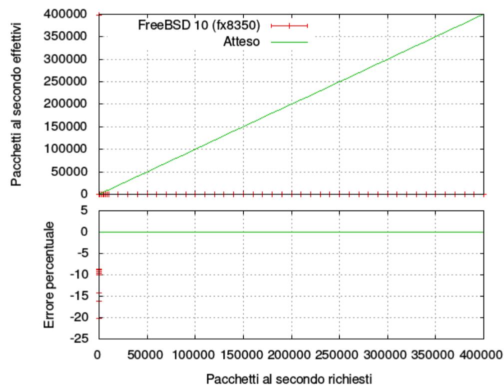
(b) Pacchetti 1518 Byte (caso migliore)



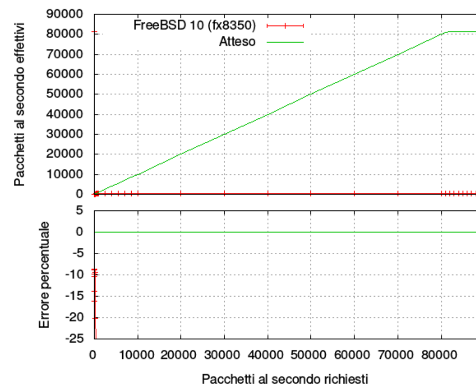
(c) Pacchetti 64 Byte (caso peggiore)



(d) Pacchetti 1518 Byte (caso migliore)



(e) Pacchetti 64 Byte (caso peggiore)

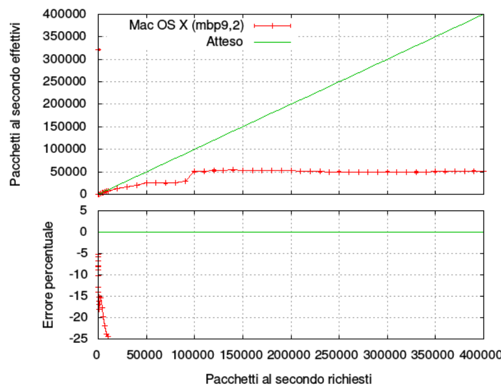


(f) Pacchetti 1518 Byte (caso migliore)

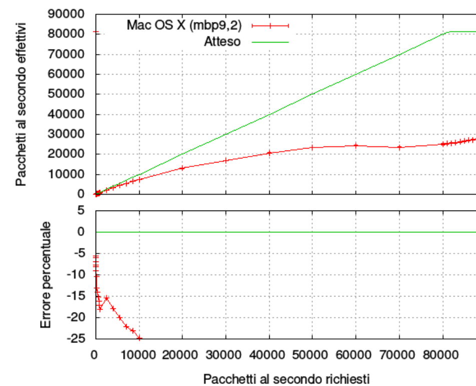
Figura 4.2: Ostinato-libpcap, profilo pacchetti trasmessi per `fx8350` (Linux, Windows, FreeBSD)

Tabella 4.2: Ostinato-libpcap, confronto fra i vari sistemi (max pps generati)

Elaboratore	GbE	Pacchetti	pps max	bps max	Uso
fx8350 (Linux)	1	64 Byte	201845	134.025 Mbps	13.40%
		512 Byte	188360	800.153 Mbps	80.02%
		1024 Byte	119739	999.10 Mbps	99.91%
		1518 Byte	81280	999.415 Mbps	99.94%
fx8350 (Win 7)	1	64 Byte	174796	116.065 Mbps	11.61%
		512 Byte	144170	612.436 Mbps	61.24%
		1024 Byte	73381	612.293 Mbps	61.23%
		1518 Byte	77079	947.759 Mbps	94.78%
fx8350 (FreeBSD)	1	64 Byte	398883	264.858 Mbps	26.49%
		512 Byte	205075	871.157 Mbps	87.12%
		1024 Byte	119734	999.060 Mbps	99.91%
		1518 Byte	81276	999.365 Mbps	99.94%
mbp9.2 (OS X)	1	64 Byte	321238	213.302 Mbps	21.33%
		512 Byte	190171	807.847 Mbps	80.78%
		1024 Byte	119731	999.038 Mbps	99.90%
		1518 Byte	81274	999.350 Mbps	99.94%
h102 (Linux)	10	64 Byte	1411731	937.389 Mbps	9.37%
		512 Byte	1286555	5465.285 Mbps	54.65%
		1024 Byte	741273	6185.185 Mbps	61.85%
		1518 Byte	744012	9148.376 Mbps	91.48%



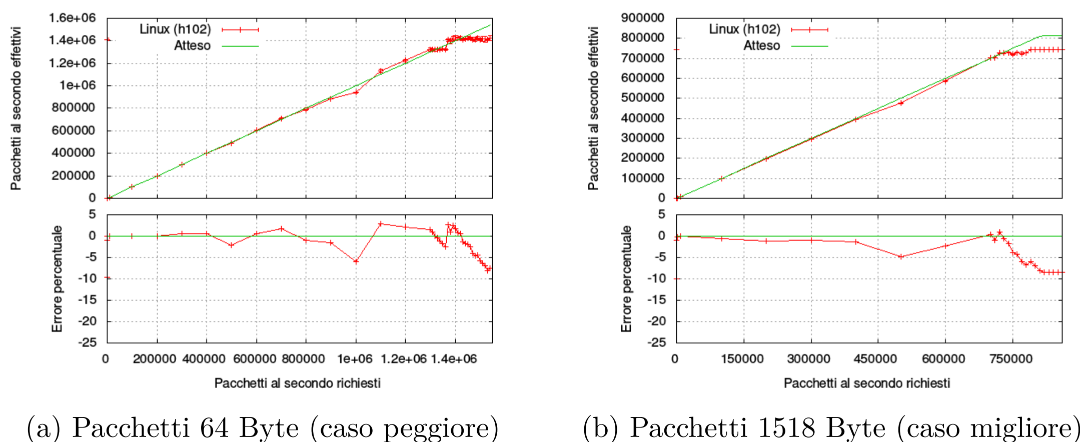
(a) Pacchetti 64 Byte (caso peggiore)



(b) Pacchetti 1518 Byte (caso migliore)

Figura 4.3: Ostinato-libpcap, profilo pacchetti trasmessi per `mbp9,2` (OS X)

Dall'analisi di questa tabella si notano inoltre sensibili discrepanze in funzione del sistema operativo impiegato. In particolare, con riferimento a `fx8350`, le prestazioni registrate in ambiente Windows sono mediamente più basse di quelle raggiunte su Linux (con valori che oscillano fra $-13,40\%$ per pacchetti piccoli, $-5,17\%$ per pacchetti grandi, e punte di $-38,72\%$ per pacchetti di medie dimensioni), mentre FreeBSD riporta le migliori prestazioni, specie con pacchetti



(a) Pacchetti 64 Byte (caso peggiore)   (b) Pacchetti 1518 Byte (caso migliore)

Figura 4.4: Ostinato-libpcap, profilo pacchetti trasmessi per `h102` (Linux)

piccoli (quasi il doppio di quanto conseguito su Linux, per pacchetti da 64 Byte).

Per quanto riguarda il rispetto del profilo di traffico imposto, solo in ambiente Linux si riscontra un comportamento soddisfacente, con una generazione abbastanza fedele fino al punto di saturazione per il sistema utilizzato, e un comportamento stabile oltre tale limite. La Figura 4.2a e la Figura 4.2b mostrano il profilo di traffico registrato da `fx8350` con Linux, rispettivamente per pacchetti da 64 Byte e 1518 Byte. In maniera simile, la Figura 4.4 illustra il risultato degli esperimenti condotti su `h102`, dalla quale è possibile osservare prestazioni maggiori (il fondo scala è differente) a scapito però di una minore precisione e linearità nel seguire il profilo di traffico imposto.

In ambiente Windows, con pacchetti piccoli (cfr. Figura 4.2c), si registra un comportamento instabile nei pressi del punto di saturazione (raggiunto cioè l'80% di tale valore): la velocità trasmissiva collassa per diverse decine di secondi, probabilmente a causa di interferenze legate al sistema operativo stesso; questo è anche il motivo per cui alcuni campioni presentano una forte incertezza statistica. Con pacchetti grandi, invece, il comportamento è più lineare, ma lo scostamento fra velocità di trasmissione richiesta ed effettiva diventa apprezzabile già intorno al 75% del valore di saturazione (cfr. Figura 4.2d).

Su OS X, guardando alla Figura 4.3, non si esagera nel dire che la situazione è disastrosa. Anche peggio avviene su FreeBSD (cfr. la Figura 4.2e e la Figura 4.2f). Su tali sistemi operativi non si presentano problemi fintanto che Ostinato è libero da vincoli di temporizzazione, come nello scenario di cui alla Tabella 4.2. Altrimenti, il profilo di traffico richiesto non viene minimamente rispettato, neanche per velocità trasmissive decisamente inferiori ai valori massimi raggiungibili. Questo è legato a una errata gestione delle marcature temporali associate ai pacchetti e al mancato utilizzo di tecniche di attesa attiva (il cosiddetto *polling*): le attese sono realizzate facendo sospendere il *thread* trasmettitore mediante chiamate alla funzione `usleep()`.

In conclusione, l'implementazione ufficiale di Ostinato presenta dei limiti sensibili sulle prestazioni, intrinseci all'utilizzo della libreria `pcap`, e una scarsa accuratezza nella generazione dei pacchetti su sistemi operativi *non* Linux. Si

Tabella 4.3: Ostinato-libpcap (rev.), confronto fra i vari sistemi (max pps generati)

Elaboratore	GbE	Pacchetti	pps max	bps max	Uso
fx8350 (Linux)	1	64 Byte	200174	132.915 Mbps	13.29%
		512 Byte	188031	798.755 Mbps	79.88%
		1024 Byte	119738	999.092 Mbps	99.91%
		1518 Byte	81279	999.410 Mbps	99.94%
fx8350 (FreeBSD)	1	64 Byte	398857	264.841 Mbps	26.48%
		512 Byte	205068	871.129 Mbps	87.11%
		1024 Byte	119734	999.060 Mbps	99.91%
		1518 Byte	81275	999.363 Mbps	99.94%
mbp9.2 (OS X)	1	64 Byte	321597	213.540 Mbps	21.35%
		512 Byte	190228	808.087 Mbps	80.81%
		1024 Byte	119732	999.040 Mbps	99.90%
		1518 Byte	81274	999.347 Mbps	99.93%
h102 (Linux)	10	64 Byte	1471809	977.281 Mbps	9.77%
		512 Byte	1292968	5492.529 Mbps	54.93%
		1024 Byte	743927	6207.324 Mbps	62.07%
		1518 Byte	743927	9147.321 Mbps	91.47%

consiglia pertanto di limitarsi all'utilizzo di Ostinato in ambiente Linux, avendo però l'accortezza di verificare che l'elaboratore di cui si dispone sia effettivamente in grado di produrre il volume di traffico richiesto dall'esperimento da svolgere.

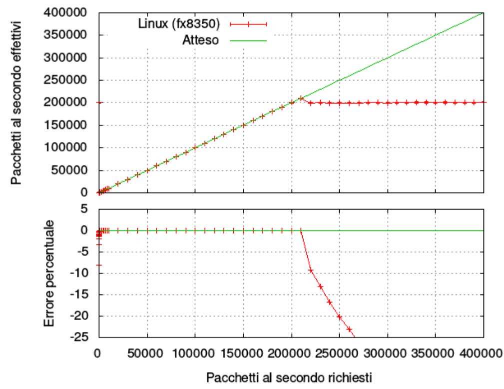
4.3 Dopo la revisione di strutture dati e algoritmi

Il codice sviluppato per realizzare l'adattamento di Ostinato con netmap ben si presta a sostituire la base di codice esistente e deputata alla trasmissione dei pacchetti utilizzando la libreria `pcap`, come descritto nella sezione 3.8.

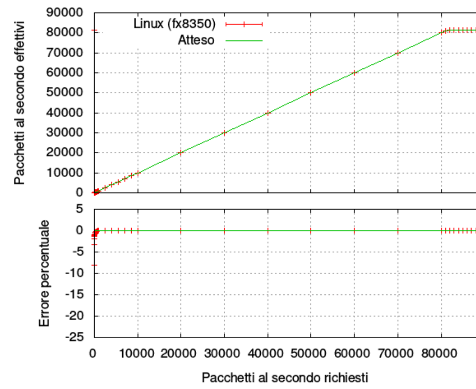
Cambiando la gestione delle marcature temporali, le strutture dati che realizzano la lista dei pacchetti da trasmettere e adottando un nuovo algoritmo per la generazione del traffico si ottengono benefici legati non tanto alle prestazioni massime raggiungibili, quanto piuttosto all'accuratezza con cui viene rispettato il profilo di traffico imposto.

I valori riportati nella Tabella 4.3, che mostrano il numero massimo di pacchetti producibili dalle varie piattaforme in funzione della dimensione dei pacchetti trasmessi, sono praticamente analoghi a quelli della Tabella 4.2, analizzati nella sezione precedente (sezione 4.2)[5].

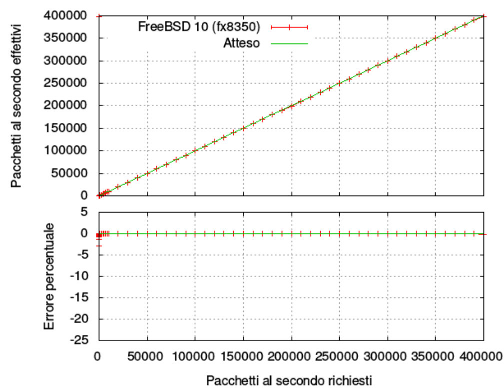
[5]Purtroppo è stato possibile realizzare accurate misurazioni anche in ambiente Windows, per motivi di tempo. Si tratta comunque di un aspetto marginale rispetto agli obiettivi prefissati con questo lavoro di tesi.



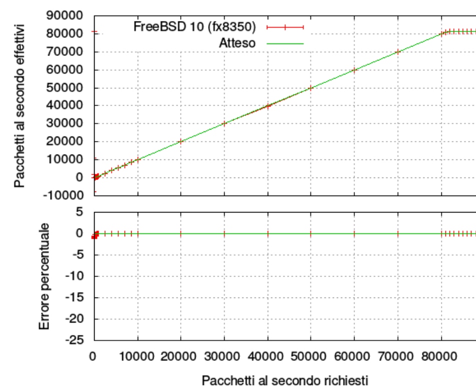
(a) Pacchetti 64 Byte (caso peggiore)



(b) Pacchetti 1518 Byte (caso migliore)

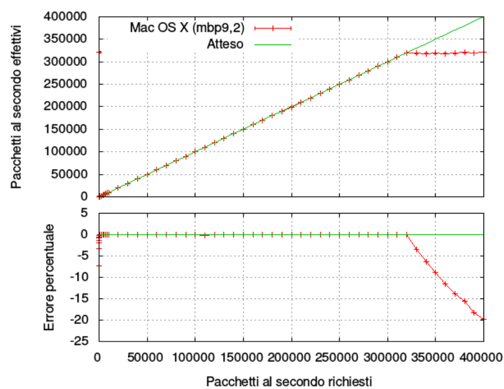


(c) Pacchetti 64 Byte (caso peggiore)

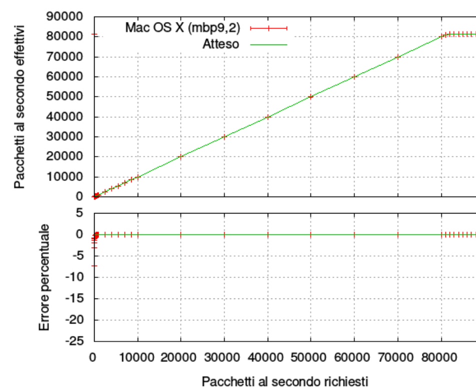


(d) Pacchetti 1518 Byte (caso migliore)

Figura 4.5: Ostinato-libpcap (rev.), profilo pacchetti trasmessi per `fx8350` (Linux, FreeBSD)

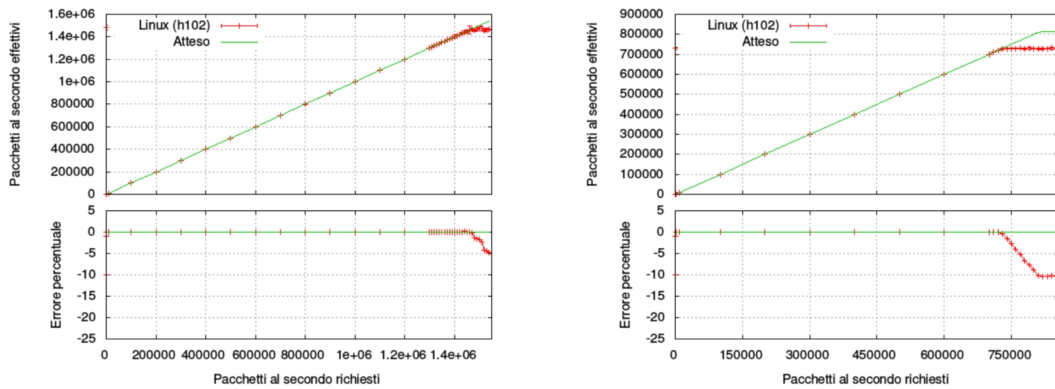


(a) Pacchetti 64 Byte (caso peggiore)



(b) Pacchetti 1518 Byte (caso migliore)

Figura 4.6: Ostinato-libpcap (rev.), profilo pacchetti trasmessi per `mbp9,2` (OS X)



(a) Pacchetti 64 Byte (caso peggiore) (b) Pacchetti 1518 Byte (caso migliore)

Figura 4.7: Ostinato-libpcap (rev.), profilo pacchetti trasmessi per `h102` (Linux)

Le differenze relative all'accuratezza del processo di generazione del traffico sono però evidenti nel momento in cui si osservano i profili registrati nelle varie configurazioni (cfr. la Figura 4.5, la Figura 4.6 e la Figura 4.7), che sono stati ottenuti sempre adottando la libreria `pcap`, ma usando una versione della classe `PcapPort` modificata come appena spiegato.

Le discrepanze fra la velocità di trasmissione richiesta e quella effettivamente prodotta è praticamente nulla fino al punto di saturazione (che dipende dalle caratteristiche della piattaforma utilizzata)[6]. Anche i problemi evidenziati alla sezione precedente per FreeBSD e OS X risultano risolti.

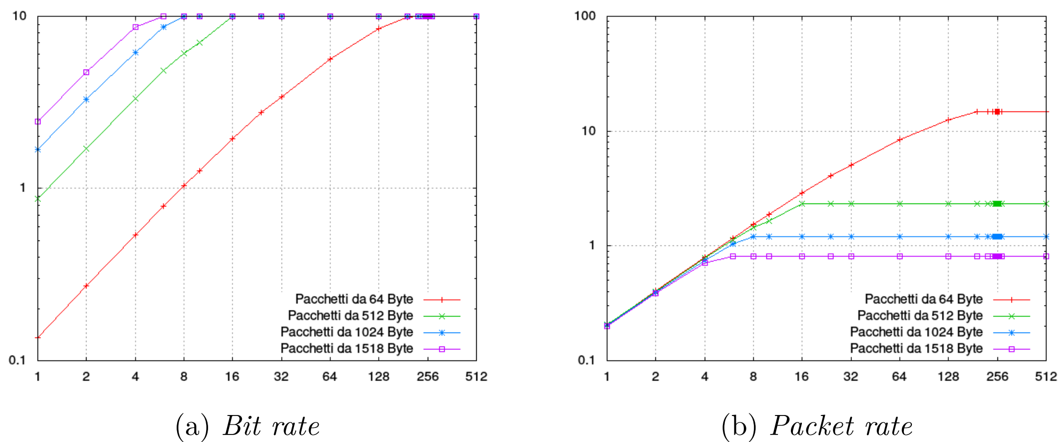
Ciò dimostra la bontà della soluzione qui proposta, che rimuove le instabilità su sistemi *non* Linux, e migliora anche per tale sistema l'accuratezza complessiva. Restano comunque valide, anche per tale soluzione, le riserve legate alle prestazioni raggiungibili da Ostinato, che possono sciolte utilizzando il *framework* netmap.

4.4 Utilizzando netmap

Utilizzando il *framework* netmap il volume di traffico prodotto da Ostinato satura un collegamento a 10 Gbit Ethernet anche nel caso peggiore di pacchetti di dimensione minima.

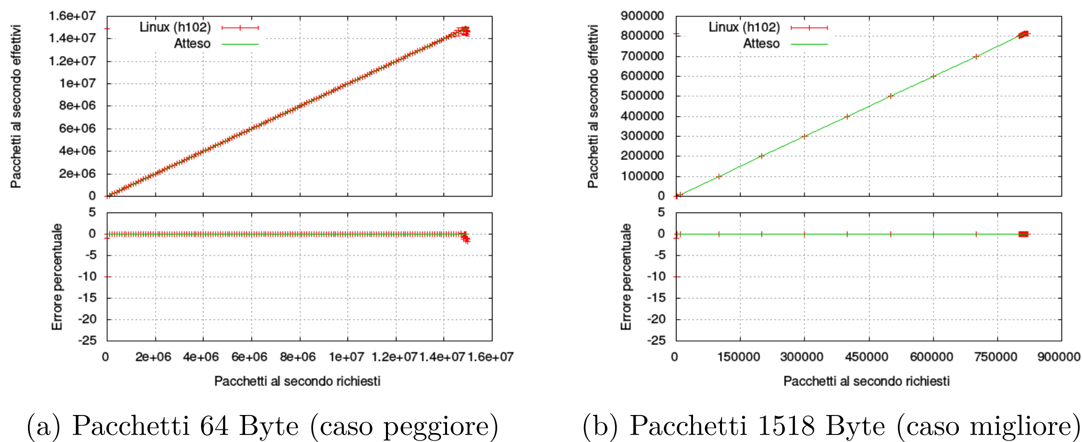
La Figura 4.8 mostra le prestazioni raggiunte, su Linux, dall'elaboratore `h102` al variare della dimensione dei pacchetti trasmessi e in funzione del limite imposto sulla dimensione massima per i *burst* (ovvero il numero massimo di pacchetti trasmessi uno dietro l'altro, senza pause nel mezzo) — si noti che i grafici sono in scala logaritmica su entrambi gli assi.

[6]Le apparenti discrepanze che appaiono nei grafici per velocità bassissime (dell'ordine dei pochi pacchetti per secondo) sono in realtà legate alle approssimazioni introdotte dal troncamento a numero intero nelle divisioni (fra numero di pacchetti inviati e tempo trascorso dall'ultima osservazione).



(a) *Bit rate* (b) *Packet rate*

Figura 4.8: Ostinato-netmap, capacità trasmissiva al variare della dimensione massima dei *burst* per `h102` (Linux)



(a) Pacchetti 64 Byte (caso peggiore) (b) Pacchetti 1518 Byte (caso migliore)

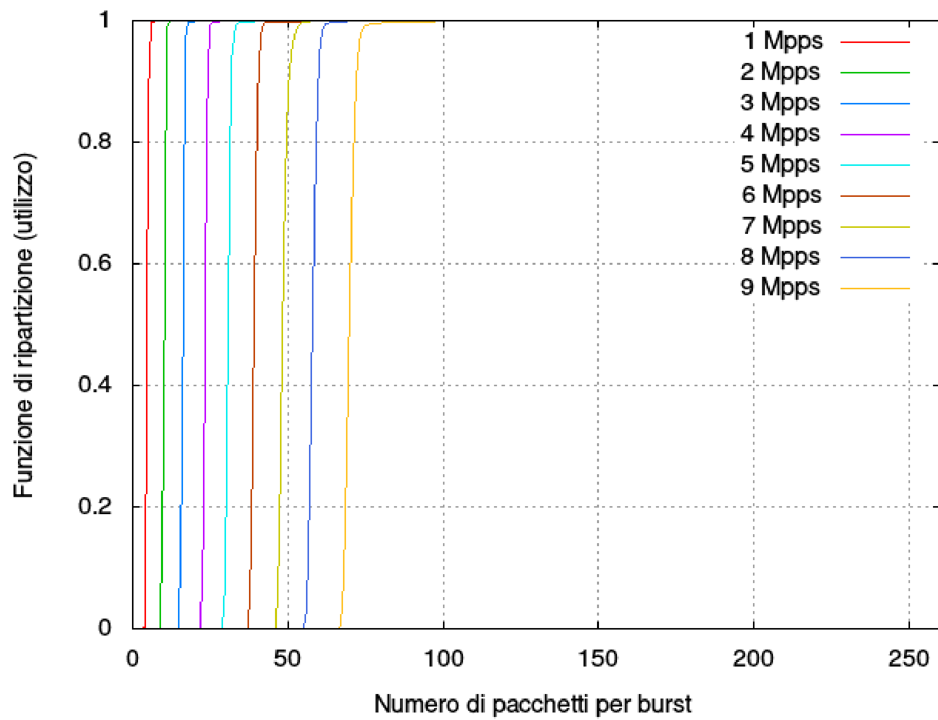
Figura 4.9: Ostinato-netmap, profilo pacchetti trasmessi per `h102` (Linux)

L'algoritmo di trasmissione adottato dalla classe `NetmapPort` produce *burst* quando i pacchetti hanno un *inter-departure time* pari a zero, oppure per recuperare eventuali ritardi accumulati. In generale, la dimensione dei *burst* prodotti è variabile nel tempo (l'algoritmo cerca di minimizzare il numero di pacchetti per *burst*, in modo da garantire la massima accuratezza possibile), ed è comunque limitata dal valore imposto dall'utente all'avvio di `drone`.

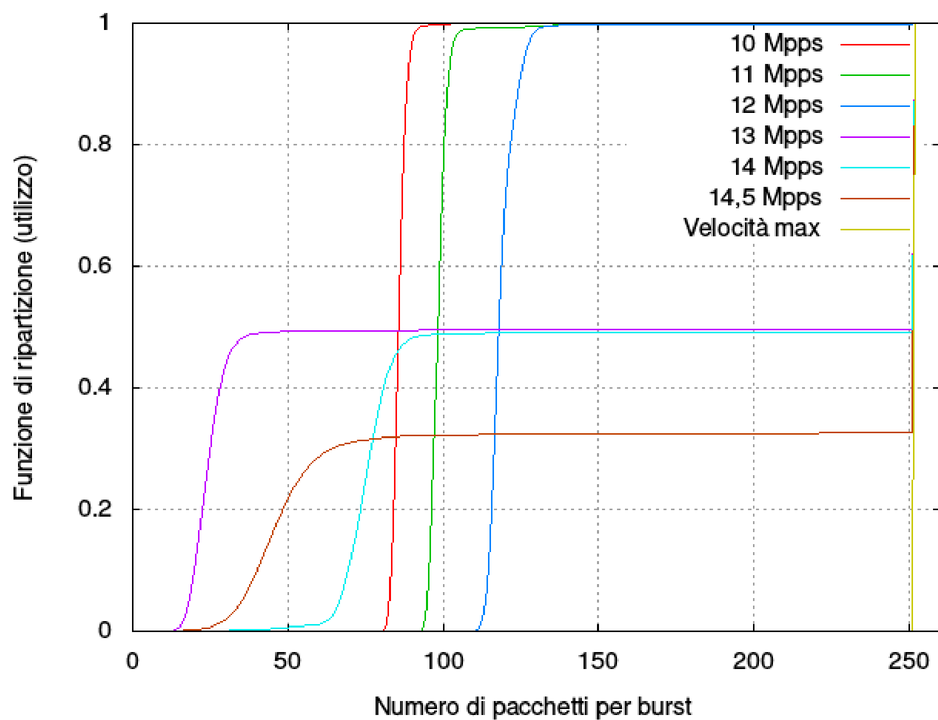
Chiaramente, a un valore minore per la dimensione massima dei *burst* corrispondono prestazioni inferiori, ma una maggiore uniformità nella distribuzione dei tempi fra pacchetti successivi. È quindi importante saper trovare il giusto compromesso.

Nel caso peggiore, per produrre 10 Gbps di traffico è necessario che questo limite sia di almeno 240 pacchetti per *burst*; con pacchetti di media/grande dimensione, 16 pacchetti per *burst* sono invece sufficienti.

Esperimenti simili, effettuati sempre su `h102`, utilizzando però VALE [21; 22] (uno *switch software* basato su netmap) per commutare il traffico fra Ostinato e una istanza di `pkt-gen` in ricezione sullo stesso elaboratore, dimostrano che il



(a) Velocità trasmissiva 1–9 Mpps



(b) Velocità trasmissiva 10 Mpps e oltre

Figura 4.10: Ostinato-netmap, distribuzione delle dimensioni dei *burst* al variare della velocità trasmissiva per `h102` (Linux)

programma può generare fino a 40 Gbps di traffico nel caso migliore di pacchetti Ethernet di dimensione massima e potendo utilizzare 32 o più pacchetti per *burst*.

In tutte le situazioni, il profilo di traffico prodotto rispetta con perfetta accuratezza il valore richiesto. Si faccia ad esempio riferimento alla Figura 4.9, ottenuta imponendo un limite massimo di 252 pacchetti per *burst*. Si registrano solo delle minime oscillazioni oltre i 13 Mpps (nel caso di pacchetti da 64 Byte) legate al fatto che per tali volumi di traffico i pacchetti vengono trasmessi in *burst* di dimensioni simili, prossime al valore massimo imposto.

Per comprendere meglio quest'ultima affermazione si consideri la Figura 4.10, che illustra la distribuzione delle dimensioni dei *burst* al variare della velocità trasmissiva, ottenuta sempre su Linux, impiegando `h102`, imponendo un limite massimo di 252 pacchetti per *burst*.

Per basse velocità trasmissive (vedi Figura 4.10a), la distribuzione è unimodale; minore è la velocità trasmissiva richiesta, minore è il numero di pacchetti che compongono ogni *burst*. Questa è una conseguenza della strategia adattativa impiegata dall'algoritmo trasmissivo di `NetmapPort` che, come detto, cerca di evitare di accorpare i pacchetti in *burst*.

Raggiunti i 13 Mpps (vedi Figura 4.10b), la distribuzione diventa bimodale. Il valore modale di sinistra si sposta verso un numero di pacchetti per *burst* crescente con la velocità trasmissiva; la sua frequenza di apparizione, invece, decresce all'aumentare della velocità di trasmissione. Il valore modale superiore corrisponde alla dimensione imposta sul numero massimo di pacchetti per *burst*.

Nel caso in cui `drone` sia configurato per trasmettere alla massima velocità possibile, la distribuzione torna a essere unimodale: i pacchetti sono accorpati in *burst* di dimensione massima, in accordo con quanto descritto alla sezione 3.5.

In sintesi si può quindi affermare che, a seguito del lavoro di revisione di strutture dati e algoritmi impiegati (vedi Capitolo 3), e grazie all'utilizzo del *framework* netmap, Ostinato si propone come uno strumento essenziale per realizzare esperimenti e misurazioni su reti ad alta velocità (10 Gigabit Ethernet), in quanto capace di coniugare prestazioni elevate, precisione, accuratezza, flessibilità e semplicità di utilizzo.

Capitolo 5

Conclusioni e sviluppi futuri

In questa tesi è stato mostrato come "Ostinato", un generatore di pacchetti *software*, versatile e multipiattaforma, è stato adattato per funzionare efficientemente su reti 10 Gigabit Ethernet utilizzando il *framework* "netmap".

Innanzitutto si è parlato dei generatori di traffico in generale, che sono piattaforme *hardware* o *software* atte a iniettare pacchetti sulla rete in maniera controllata, utilizzate comunemente come strumenti per la misurazione delle prestazioni e del corretto funzionamento dei sistemi di rete (siano essi dispositivi fisici, protocolli di comunicazione, o applicazioni). La discussione è stata incentrata sui generatori di tipo *software*, in quanto largamente impiegati da ricercatori e operatori nell'ambito delle reti, fornendo i criteri per una loro classificazione (accompagnandoli con esempi significativi). Sono state quindi esposte le difficoltà legate alla realizzazione di tale classe di generatori per reti ad alta velocità, facendo anche riferimento allo stato dell'arte in tale settore.

Successivamente sono state presentate le funzionalità di Ostinato, che è caratterizzato da una grande flessibilità e configurabilità, e ne è stata illustrata l'organizzazione interna, mettendo in luce quelli che sono i difetti principali del programma rispetto agli stringenti vincoli imposti dalle reti ad alta velocità, anche attraverso un ricco insieme di dati e misurazioni sperimentali.

Si è passati quindi a descrivere come Ostinato è stato ristrutturato, limitatamente alle parti deputate alla gestione e alla trasmissione dei pacchetti costituenti i flussi di traffico configurati, e come è stato realizzato l'adattamento con netmap (utilizzando l'API nativa di questo *framework*), indicando anche tutti quegli accorgimenti da adottare nello sviluppo di generatori *software* ad alta velocità.

Gli esperimenti effettuati evidenziano un apprezzabile miglioramento di accuratezza nella generazione del traffico (intesa come la misura della differenza fra il profilo di traffico richiesto e quello effettivamente prodotto).

È stato poi mostrato come, utilizzando netmap, le prestazioni di Ostinato aumentano di un fattore 10 (e oltre) rispetto a quanto registrato con la libreria `pcap`; difatti, con netmap il generatore è in grado di saturare un collegamento a 10 Gbps anche nel caso peggiore (trasmettendo pacchetti Ethernet di dimensione minima).

Questa tesi dimostra quindi che, con opportuni accorgimenti, è possibile sfruttare le capacità di elaborazione dei recenti calcolatori per realizzare gene-

ratori *software* capaci di coniugare versatilità, prestazioni elevate e un'ottima accuratezza.

Riguardo agli sviluppi futuri, è necessario intervenire sull'interfaccia grafica di Ostinato (la componente *client*, chiamata `ostinato`) per integrarvi le nuove funzionalità relative a netmap; ad esempio, introdurre la possibilità di commutare una porta in modalità netmap (e, viceversa, di farla ritornare alla modalità operativa normale), o di creare tramite la GUI una nuova porta VALE. In questa tesi ci si è infatti concentrati sul *back end* di Ostinato (la componente *server*, chiamata `drone`), configurando tutti i parametri relativi a netmap attraverso opzioni date dalla linea di comando, all'avvio del programma stesso.

Inoltre, se in questa tesi ci si è occupati della generazione di traffico ad alta velocità mediante netmap, similmente è possibile pensare di utilizzare tale *framework* in Ostinato per la cattura su *file* del traffico ad alta velocità. La struttura del codice è già predisposta per accogliere tale funzionalità; è quindi sufficiente implementare il semplice salvataggio su file del traffico trasmesso e ricevuto.

Vale la pena infine notare che l'approccio e gran parte del codice sviluppato nell'ambito di questo lavoro di tesi (che sarà reso disponibile sulla *mailing list* del progetto) possono essere riutilizzati per introdurre in Ostinato il supporto nativo ad altri *framework* di rete ad alte prestazioni (come PF_RING DNA), o ad *hardware* specializzato (come gli adattatori di rete Napatech).

Bibliografia

[1] Nicola Bonelli, Andrea Di Pietro, Stefano Giordano, and Gregorio Procissi. Flexible High Performance Traffic Generation on Commodity Multi–core Platforms. In *Traffic Monitoring and Analysis*, volume 7189 of *Lecture Notes in Computer Science*, pages 157–170. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28533-2. URL http://pam2012.ftw.at/TMA/papers/TMA2012paper14.pdf. 10

[2] Alessio Botta, Alberto Dainotti, and Antonio Pescapé. Do you trust your software-based traffic generator? *IEEE Communications Magazine*, 48(9):158–165, September 2010. ISSN 0163-6804. doi: 10.1109/MCOM.2010.5560600. URL http://wpage.unina.it/a.botta/pub/COMMAG_GENERATION.pdf. 3, 9, 11

[3] Luca Deri. Building a 10 Gbit Traffic Generator using PF_RING and Ostinato, 2011. URL http://www.ntop.org/pf_ring/building-a-10-gbit-traffic-generator-using-pf_ring-and-ostinato/. 38

[4] Sally Floyd. Traffic Generators for Internet Traffic, 2010. URL http://www.icir.org/models/trafficgenerators.html. 4

[5] Philipp K. Janert. *Gnuplot in Action.* Manning Pubblications Co., 1st edition, August 2009. ISBN 978-1-933988-39-9.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice Hall, 2nd edition, April 1988. ISBN 9780131103627. 42

[7] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach.* Prentice Hall, 6th edition, May 2012. ISBN 978-0273768968. 1

[8] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley Professional Computing Series. Addison-Wesley Professional, 1st edition, December 1995. ISBN 978-0-13-280013-6. 42

[9] Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library.* Addison-Wesley Professional Computing Series. Addison-Wesley Professional, 1st edition, June 2001. ISBN 978-0-201-74962-5.

[10] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley Professional Computing Series. Addison-Wesley Professional, 3th edition, May 2005. ISBN 978-0-321-33487-9. 42

[11] Sándor Molnár, Péter Megyesi, and Géza Szabó. How to validate traffic generators? In *IEEE International Conference on Communications 2013 (IEEE ICC'13), 1st IEEE Workshop on Traffic Identification and Classification for Advanced Network Services and Scenarios (TRICANS)*, pages 1340–1344. IEEE, June 2013. ISBN 978-1-4673-5753-1. doi: 10.1109/ICCW.2013.6649445. URL `http://hsnlab.tmit.bme.hu/~molnar/files/tricans2013.pdf`. 11

[12] Napatech Inc. Whitepaper: Traffic Generation for the Mainstream Ethernet Market, November 2010. URL `http://www.nextcomputing.com/images/whitepapers/traffic_generation_whitepaper.pdf`. 12

[13] NextComputing. Continuum TGEN, 2013. URL `http://www.nextcomputing.com/products/network-test-equipment/continuum`. 12

[14] Ostinato. Issue 11: CLI/Scripting interface, 2010. URL `http://code.google.com/p/ostinato/issues/detail?id=11`. 15

[15] Georgios Z Papadopoulos. *Experimental Assessment of Traffic Generators.* M.sc., Universidad Carlos III de Madrid, 2012. URL `http://georgiospapadopoulos.com/MScThesis.pdf`. 4

[16] Marcos Paredes-Farrera, Martin Fleury, and Mohammed Ghanbari. Precision and accuracy of network traffic generators for packet-by-packet traffic analysis. In *Proceedings of the 2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM 2006)*, pages 37–42. IEEE, 2006. ISBN 1-4244-0106-2. doi: 10.1109/TRIDNT.2006.1649124. URL `http://www.researchgate.net/publication/224636756_Precision_and_accuracy_of_network_traffic_generators_for_packet-by-packet_traffic_analysis/file/e0b49515db5a7e79e9.pdf`. 2, 3

[17] Nicolas Planel. Add Python support based on ProtocolBuffer RPC Channel, 2013. URL `https://groups.google.com/forum/#!topic/ostinato/WnNSv5d1SO8`. 15, 24, 53

[18] Riverbed Technology. WinPcap library, 2013. URL `http://www.winpcap.org`. 16

[19] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, pages 9–20. USENIX Association Berkeley, June 2012. URL `http://info.iet.unipi.it/~luigi/papers/20120503-netmap-atc12.pdf`. iii, 10

[20] Luigi Rizzo. The netmap project, 2013. URL `http://info.iet.unipi.it/~luigi/netmap/`. iii, 10

[21] Luigi Rizzo and Giuseppe Lettieri. VALE, a Switched Ethernet for Virtual Machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies (CoNEXT '12)*, pages 61–72, New York, New York, USA, December 2012. ACM Press. ISBN 9781450317757. doi: 10.1145/2413176.2413185. URL `http://info.iet.unipi.it/~luigi/papers/20121026-vale.pdf`. 39, 62

[22] Luigi Rizzo and Giuseppe Lettieri. VALE, a Virtual Local Ethernet, 2013. URL `http://info.iet.unipi.it/~luigi/vale/`. 39, 62

[23] Luigi Rizzo, Luca Deri, and Alfredo Cardigliano. 10 Gbit/s Line Rate Packet Processing Using Commodity Hardware: Survey and new Proposals, 2011. URL `http://luca.ntop.org/10g.pdf`. 8, 10, 51

[24] Luigi Rizzo, Marta Carbone, and Gaetano Catalli. Transparent acceleration of software packet forwarding using netmap. In *2012 Proceedings IEEE INFOCOM*, pages 2471–2479. IEEE, March 2012. ISBN 978-1-4673-0775-8. doi: 10.1109/INFCOM.2012.6195638. URL `http://info.iet.unipi.it/~luigi/netmap/20110729-rizzo-infocom.pdf`. 11

[25] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation (NSDI '06)*, pages 239–252. USENIX Association Berkeley, May 2006. URL `http://users.cms.caltech.edu/~adamw/papers/openvsclosed.pdf`. 4

[26] Henning Schulzrinne. Traffic Generators, 2011. URL `http://www.cs.columbia.edu/~hgs/internet/traffic-generator.html`. 4

[27] P. Srivats. Ostinato project, 2012. URL `http://code.google.com/p/ostinato/`. iii, 15

[28] William Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley Professional Computing Series. Addison-Wesley Professional, 3th edition, May 2013. ISBN 978-0-321-63773-4.

[29] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, May 2013. ISBN 978-0-321-56384-2. 23

[30] The Tcpdump Group. Libpcap savefile format, 2008. URL `http://www.tcpdump.org/manpages/pcap-savefile.5.html`. 16, 29

[31] The Tcpdump Group. Tcpdump/Libpcap public repository, 2013. URL `http://www.tcpdump.org`. 16

[32] The Wireshark Foundation. Wireshark project, 2013. URL http://www.wireshark.org. 16

[33] Traffic Research Group. Other Internet Traffic Generators, 2013. URL http://traffic.comics.unina.it/software/ITG/link.php. 4

[34] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and responsive network traffic generation. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications (ACM SIGCOMM '06)*, pages 111–122. ACM, 2006. ISBN 1-59593-308-5. doi: 10.1145/1159913.1159928. URL http://pdf.aminer.org/000/588/979/realistic_and_responsive_network_traffic_generation.pdf. 2

[35] Petr Zach, Martin Pokorny, and Arnost Motycka. Design of Software Network Traffic Generator. *Recent Advances in Circuits, Systems, Telecommunications and Control*, pages 244–251, October 2013. ISSN 1790-5117. URL http://www.wseas.us/e-library/conferences/2013/Paris/CCTC/CCTC-35.pdf. 2