Politecnico di Torino
Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Development of an Architecture for Packet Capture and Network Traffic Analysis

Relatori:

Ing. Mario Baldi

Prof. Marco Mezzalama

Candidato:

Loris Degioanni

Marzo 2000

Table of contents

1. Introduzione

La rapida evoluzione del settore delle reti di calcolatori, unita al vasto utilizzo delle tecnologie di networking ed alla crescente diffusione di *Internet*, fa sì che la potenza e la complessità delle strutture di comunicazione cresca ogni giorno. Questo fatto, se da una parte aumenta le potenzialità a disposizione degli utenti, dall'altro rende più difficile il lavoro di chi deve progettare, configurare e rendere sicura una rete. Per questo motivo, è di fondamentale importanza la disponibilità di efficienti strumenti che permettano l'analisi e il monitoraggio del funzionamento delle reti. Alcuni di questi strumenti richiedono hardware dedicato, altri invece funzionano tramite un normale computer connesso alla rete. Anche se meno efficienti, questi ultimi garantiscono bassi costi, versatilità e facilità di aggiornamento, rivelandosi indispensabili in tutte quelle situazioni dove non è necessaria la potenza di una soluzione dedicata. Per poter esaminare il traffico di una rete con l'hardware di un normale Personal Computer, è necessario che il sistema operativo fornisca alle applicazioni un insieme di funzioni per la cattura e il filtraggio dei dati.

Questo lavoro di tesi si propone lo sviluppo di un'architettura –*winpcap*- che aggiunge ai sistemi operativi della famiglia Windows di Microsoft la capacità di catturare e filtrare il traffico di rete, e di un programma di analisi –*WinDump*- che funziona utilizzando tale architettura.

Fanno parte dell'architettura un driver di cattura (*packet capture driver*), una libreria dinamica con funzioni a basso livello (*PACKET.DLL*) e una libreria statica di alto livello (*libpcap*).

Il lavoro trae ispirazione, sviluppandone le capacità, dal *Berkely Packet Filter* (BPF) [1] sviluppato alla University of California per i sistemi UNIX della famiglia BSD.

Binari e sorgenti dell'intero progetto sono distribuiti gratuitamente tramite la licenza Berkeley, ed è possibile scaricarli assieme a documentazione ed esempi dai siti Internet di winpcap (http://netgroup-serv.polito.it/winpcap) e WinDump (http://netgroup-serv.polito.it/windump).

1.1. Architettura

Le applicazioni per la cattura e l'analisi del traffico di rete hanno bisogno, per svolgere il loro lavoro, di interagire con l'hardware di rete del computer su cui girano in modo da ottenere i pacchetti nella loro forma originale, senza che essi siano elaborati dalla pila protocollare. In un sistema operativo moderno, d'altra parte, a un'applicazione non è consentita l'interazione diretta con i dispositivi fisici, in quanto essi sono gestiti esclusivamente dal sistema.

Per catturare dalla rete è allora necessaria una vera e propria *architettura*, costituita da due parti principali: una funzionante all'interno del kernel del sistema operativo e un'altra funzionante a livello utente.

La prima delle due parti ha il compito di interagire direttamente con l'interfaccia di rete per ottenere i pacchetti e consegnarli alle applicazioni che ne fanno richiesta. Questo compito dovrà essere svolto in maniera il più possibile efficiente, in modo da non gravare sulle risorse di sistema anche nel caso di reti veloci o con alto traffico. Inoltre, la parte a basso livello dell'architettura di cattura dovrà essere altamente versatile, in modo che un'applicazione abbia la possibilità di implementare

caratteristiche aggiuntive e nuovi protocolli senza dover aggiornare l'intera architettura.

La parte funzionante a livello utente è invece l'applicazione vera e propria, che fornisce all'utente un'interfaccia per l'analisi o il monitoraggio della rete. Essa si avvale delle primitive offerte dal kernel per ottenere i pacchetti transitanti in rete. Le principali caratteristiche richieste a questa parte dell'architettura sono facilità di utilizzo, potenza delle funzioni fornite e portabilità.

La struttura dello stack di cattura sviluppato in questa tesi è rappresentato in figura 1.1.

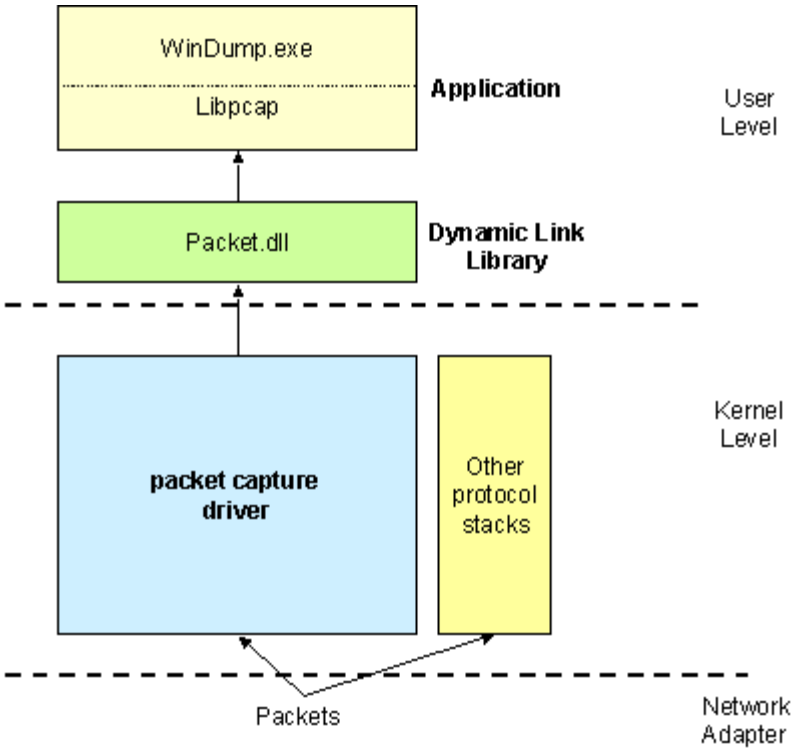

Figura 1.1: struttura di winpcap

Al livello più basso c'è l'adattatore di rete, che viene usato per catturare i pacchetti in transito sul mezzo trasmissivo. Durante una cattura, solitamente l'adattatore

funziona in una modalità particolare, chiamata *promisquous mode*, che forza il dispositivo ad accettare tutti i pacchetti e non solamente quelli destinati a lui.

Il Packet Capture Driver è il modulo software di livello più basso. Funziona a livello kernel e si occupa di ottenere i pacchetti dall'adattatore di rete e passarli alle applicazioni.

PACKET.DLL fa da tramite fra il Packet Capture Driver e le applicazioni. Il suo scopo principale è fornire un'interfaccia di cattura comune a tutti i sistemi operativi della famiglia Windows. Questo permette alle parti superiori dell'architettura di essere system-independent.

libpcap è una libreria che offre funzioni di cattura con un alto livello di astrazione dall'hardware e dal sistema operativo. È linkata staticamente all'applicazione, perciò fa parte dell'eseguibile del programma di analisi.

La parte più ad alto livello dello stack di cattura è il programma vero e proprio, in questo caso WinDump, che fornisce l'interfaccia per l'utilizzatore.

Nel resto di questa introduzione verranno trattati con maggiori dettagli ognuno dei moduli appena descritti e i loro meccanismi di funzionamento.

1.2. Il Packet Capture Driver

Il Packet Capture Driver aggiunge ai sistemi operativi Windows la capacità di catturare il traffico di rete, offrendo un'interfaccia simile a quella del BPF [1]. Esso inoltre da la possibilità di scrivere pacchetti e fornisce alcune funzioni per l'analisi della rete che non sono presenti nel BPF.

Il Packet Capture Driver è visto dalle applicazioni come un normale device driver, ed è realizzato come Kernel Driver (packet.sys) per Windows NT e Windows 2000, e come Virtual Device Driver (packet.vxd) per Windows 95 e Windows 98. In questo

modo le applicazioni possono accedere ai suoi servizi tramite normali primitive di lettura e scrittura. In realtà esso è un driver di periferico anomalo, perché non gestisce direttamente l'interfaccia di rete, ma si installa nella parte dei kernel Microsoft dedicata al networking, NDIS.

NDIS è responsabile della gestione degli adattatori di rete e dell'interazione fra gli adattatori e i driver di protocollo. Il Packet Capture Driver è implementato in NDIS come driver di protocollo. Questo gli impedisce l'interazione diretta con l'hardware, ma gli permette di funzionare in maniera indipendente dal tipo di rete sottostante.

La struttura del driver, condivisa da tutte le versioni, è mostrata in figura 1.2.

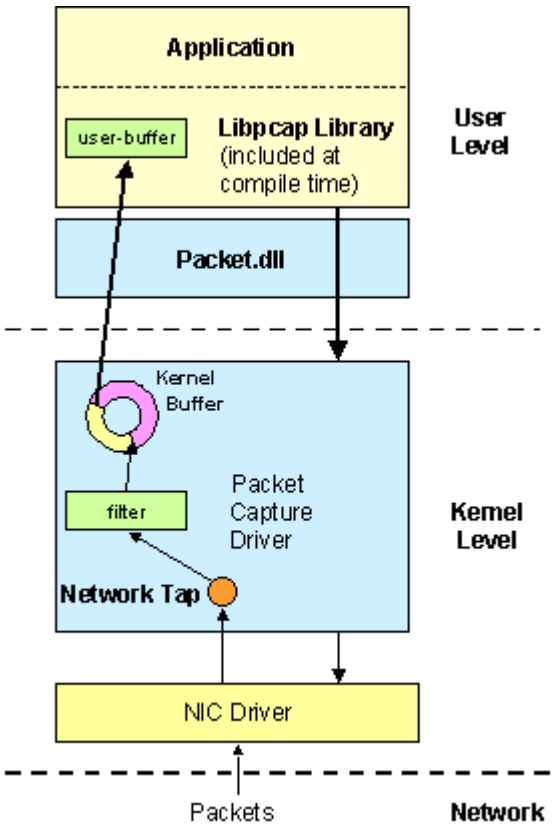

Figura 1.2: struttura del Packet Capture Driver

Le frecce verso l'alto indicano il percorso dei pacchetti dalla rete all'applicazione di cattura. Ogni volta che un'applicazione inizia una cattura, il driver alloca un filtro e un buffer, che si occuperanno rispettivamente di scremare e memorizzare il traffico proveniente dalla rete. Sia il tipo di filtro che la dimensione del buffer sono modificabili dall'applicazione. I dati memorizzati saranno poi passati all'applicazione nel momento in cui essa sarà pronta a riceverli. La freccia di spessore maggiore fra il kernel-buffer e lo user-buffer indica che più di un pacchetto può essere trasferito fra queste due entità con una singola system call.

Il driver supporta un numero qualsiasi di adattatori di rete e di applicazioni. È anche possibile per una singola applicazione eseguire più catture contemporaneamente.

1.2.1. Il processo di filtraggio

Il processo di filtraggio serve a decidere se un pacchetto debba essere accettato e passato all'applicazione in ascolto. Questa operazione di scrematura del flusso di dati in arrivo si rende necessaria perché la maggior parte delle applicazioni ha bisogno solamente di un sottoinsieme del traffico di rete. Copiare e memorizzare l'intero flusso in arrivo sarebbe pertanto uno spreco di risorse. Un filtro è, nella sua accezione più semplice, una funzione booleana che viene applicata ad ogni pacchetto. Se il valore ritornato è 'vero', il driver passa il pacchetto all'applicazione, se è 'falso' il pacchetto è immediatamente scartato.

Il meccanismo usato dal Packet Capture Driver deriva direttamente dal sistema di filtraggio del BPF, che è stato opportunamente modificato per poter essere usato nei kernel Windows. Esso prevede un passo avanti nel concetto filtraggio: non solo

decide se un pacchetto deve essere tenuto, ma stabilisce anche la dimensione della parte da conservare. Questo è utile in tutti quei casi in cui non c'è bisogno dell'intero pacchetto (per esempio quando serve solamente l'header), perché permette di scartare fin dal principio le parti che non saranno utilizzate.

A livello implementativo, il filtro BPF è un processore virtuale con un accumulatore, un registro indice, una memoria e un program counter implicito. È in grado di eseguire istruzioni di load and store, confronto, salto, più una serie di operazioni aritmetiche. Un'applicazione che desidera filtrare il flusso di pacchetti in arrivo crea un programma di filtraggio adeguato, e poi lo passa al driver con una chiamata IOCTL. Il driver lo applicherà ad ogni pacchetto in arrivo.

Un meccanismo del genere ha alcuni importanti vantaggi:

- È estremamente versatile, in quanto permette configurabilità totale e un vasto insieme di utilizzi. Inoltre è indipendente da protocolli e architetture di rete, e ha la possibilità di supportarne di nuovi in maniera semplice e trasparente.
- È abbastanza generico da poter essere usato in situazioni nuove o particolari.
- Offre alte prestazioni, in quanto il processore virtuale può essere mappato in maniera efficiente su quello reale.

Il problema più grosso, invece, è dovuto al fatto che, senza gli adeguati controlli, uno pseudo-programma contenente errori potrebbe causare grossi problemi al sistema. Si pensi ad esempio al caso di divisioni per zero o di accessi a zone di memoria non legali: tali operazioni, se eseguite a livello kernel, implicano l'immediato crash del sistema. Per evitare questo genere di problemi, il driver effettua un controllo preventivo sul codice ogni volta che un nuovo filtro viene installato, per individuare situazioni potenzialmente pericolose. Se il filtro passa questo controllo, verrà eseguito su ogni nuovo pacchetto catturato.

È importante che il filtro sia eseguito il prima possibile, perché prima si decide se scartare il pacchetto, meno risorse saranno richieste per la sua gestione.

Per questo motivo, il Packet Capture Driver applica il filtro quando il pacchetto è ancora nella memoria della scheda di rete, cioè non appena esso arriva nel sistema. Questo permette al driver di non effettuare nessuna copia se il pacchetto è scartato e porta a una gestione molto efficiente di quei frequenti casi in cui l'applicazione voglia solo un sottoinsieme molto piccolo del traffico di rete (per esempio quello generato da una singola stazione).

1.2.2. Il processo di lettura

Il Packet Capture Driver offre alle applicazioni un metodo per ottenere il traffico di rete molto simile alla normale lettura da file. Un programma, dopo aver aperto il packet driver, riceve i pacchetti effettuando una system call di read e scrive sulla rete con una chiamata di write[1].

È possibile, grazie a una serie di timer interni al driver, effettuare letture temporizzate che scadono dopo un certo tempo anche se nessun pacchetto è arrivato. Questa caratteristica è stata inserita per compatibilità col BPF, ma soprattutto per supportare lo *statistics mode*, che serve per ottenere in modo efficiente statistiche sulla rete e che verrà presentato in seguito.

Quando un pacchetto appena arrivato supera il filtro, ci si può trovare in due diverse situazioni:

---

[1] In realtà il meccanismo in Windows 95 e Windows 98 è più complesso, in quanto questi sistemi non danno la possibilità di effettuare chiamate di read e write sui device driver

- L'applicazione è pronta a ricevere il pacchetto, in quanto aveva eseguito in precedenza un read sul driver e adesso è in attesa dei dati. In questo caso il pacchetto viene immediatamente passato all'applicazioe.
- L'applicazione è al momento impegnata in altre operazioni. Per evitare di perdere il pacchetto, esso dovrà essere memorizzato nel kernel buffer (si veda la fig 1.2) e verrà consegnato all'applicazione alla prossima lettura.

Il kernel buffer è implementato come buffer circolare. Un pacchetto è conservato nel buffer assieme a un header che contiene informazioni come la dimensione e il tempo di arrivo. Inoltre fra i pacchetti sono inseriti opportuni spazi vuoti, che hanno lo scopo di allineare i dati in memoria per ottimizzare le operazioni di copia.

La scelta di utilizzare un unico buffer circolare è abbastanza originale: il BPF, per esempio, utilizza una struttura a doppio buffer che facilita l'isolamento fra adattatore di rete e applicazione permettendo al contempo buone prestazioni. Un'architettura a singolo buffer circolare permette di sfruttare meglio la memoria allocata, ma ha il problema di rendere più pesante il processo di copia (perché non si conosce a priori la quantità di dati da copiare). Per ridurre al minimo l'influenza di questo problema, varie ottimizzazioni sono state apportate. I test dimostrano che grazie a queste ottimizzazioni e al migliore sfruttamento della memoria, la tecnica di buffering del Packet Capture Driver è più efficiente di quella del BPF. Inoltre il Packet Capture Driver ha il vantaggio di essere molto meno rigido nella gestione dei buffer del driver e dell'applicazione, che possono avere dimensioni arbitrarie e non correlate fra di loro (al contrario di quanto succede con il BPF).

---

ed è necessario eseguire delle IOCTL. In ogni non caso la differenza è solo sintattica, e la modalità di accesso rimane la stessa.

Se nel momento in cui arriva un nuovo pacchetto il kernel buffer è pieno, il pacchetto viene scartato. Un contatore consultabile dall'applicazione tramite una IOCTL tiene conto dei pacchetti scartati per questo motivo[2].

La dimensione del buffer è un parametro di fondamentale importanza per evitare la perdita di dati, per questo motivo l'applicazione può modificarla in ogni momento secondo le sue necessità.

Una delle operazioni che influenza maggiormente le prestazioni del processo di cattura è la copia dei dati dal kernel buffer alla memoria dell'applicazione. Copiare un pacchetto per volta sarebbe improponibile, perché l'alta frequenza con cui i pacchetti arrivano richiederebbe un gran numero di system call, cioè di cambi di contesto fra user mode e kernel mode. Essendo il cambio di contesto un processo notoriamente lento, anche le macchine più veloci vedrebbero gran parte della potenza di calcolo consumata dal processo di copia. Per questo motivo, se il kernel buffer contiene diversi pacchetti, questi saranno spostati nello user buffer in una sola system-call. Sarà poi compito dell'applicazione interpretare in maniera corretta il blocco di dati ricevuto. Per facilitare questo compito, ogni pacchetto è passato all'applicazione insieme all'header che il driver aveva costruito al momento della sua ricezione. Questo permette all'applicazione di risalire alla dimensione del pacchetto prima e dopo il filtraggio e all'esatto tempo di ricezione.

---

[2] In realtà c'è un altro modo di perdere un pacchetto: se la distanza temporale fra due pacchetti è molto bassa, il secondo può arrivare mentre il driver sta ancora elaborando il primo. Questo può succedere se si cattura su reti molto veloci o con macchine molto lente.

1.2.3. Il processo di scrittura

Il Packet Capture Driver offre la possibilità di spedire pacchetti in modo diretto, evitando di passare attraverso i diversi strati di rete. Questa caratteristica, non sfruttata da libpcap e WinDump, si rivela utile quando è necessario 'forgiare' i pacchetti da spedire, compresi gli header che di solito sono costruiti dal sistema operativo. È usata per esempio per testare la sicurezza di una rete o le prestazioni di un apparato.

Per aumentare la velocità di scrittura, dalla versione 2.01 il packet driver offre una funzione chiamata 'multiple write'. Essa permette di ottimizzare il processo di generazione quando bisogna ripetere molte volte lo stesso pacchetto, ed è ideale per testare le prestazioni di bridge, router o analizzatori di rete.

1.2.4. Lo 'statistics mode'

Capita sovente che il network manager non abbia bisogno di tutti i pacchetti, ma di valori statistici come il carico della rete, il livello di broadcast, il numero di richieste web al secondo.

Lo statistics mode è una particolare modalità di funzionamento del Packet Capture Driver concepita per fornire un supporto semplice ed efficiente alle applicazioni che devono ottenere statistiche sul traffico della rete. Quando funziona in questa modalità, il driver non cattura nulla, ma si limita a contare il numero di pacchetti e la quantità di bytes che soddisfano il filtro settato dall'utente. Questi valori sono passati all'applicazione a intervalli regolari la cui frequenza è configurabile tramite una IOCTL.

La versatilità di questo meccanismo è garantita dall'uso del filtro BPF, che permette di selezionare il traffico in maniera elaborata e flessibile. I vantaggi rispetto ai normali metodi di cattura sono due.

- Si evita qualsiasi copia, perché i pacchetti sono filtrati e contati, ma solo i contatori sono passati al livello utente. Ciò comporta un notevole risparmio di CPU.
- Non viene utilizzato alcun buffer, perché non vi è necessità di memorizzare i pacchetti. Si può quindi evitare di allocare memoria per il processo di cattura.

1.3. PACKET.DLL: packet driver API

PACKET.DLL è una libreria dinamica che fa da tramite fra il Packet Capture Driver e le applicazioni a livello utente. Essa offre un set di funzioni che permettono di accedere a tutte le funzioni del driver evitando la complessità dell'interazione tramite chiamate di sistema. Inoltre gestisce per l'utente in maniera trasparente cose come i buffer per i pacchetti o la gestione degli adattatori di rete.

Il vero vantaggio maggiore che si ha usando PACKET.DLL è però quello dell'indipendenza dal sistema operativo. Infatti il packet driver, essendo parte integrante del kernel, è strettamente dipendente dall'architettura del sistema[3], e quindi prevede meccanismi di interazione diversi a seconda che ci si trovi in Windows 95, 98 NT o 2000. PACKET.DLL è fornita in varie versioni specifiche per ogni sistema, ma tutte queste versioni esportano la medesima interfaccia di programmazione. Ciò

[3] Vi sono infatti varie versioni del Packet Capture Driver, una per ogni sistema operativo della famiglia Windows, e ognuna di esse è diversa dalle altre.

significa che un applicativo che usa PACKET.DLL per la cattura funzionerà su ogni sistema Windows senza modifiche e senza essere ricompilato.

L'utilizzo di PACKET.DLL ha permesso di avere un'unica versione di libpcap che funziona su tutte le piattaforme Win32.

1.3.1. Differenze fra PACKET.DLL e libpcap

Il programmatore che voglia usare le funzioni di cattura fornite dal Packet Capture Driver ha due possibili scelte: utilizzare direttamente le funzioni di PACKET.DLL oppure fare uso di libpcap.

Per normali operazioni di cattura il consiglio è di usare sempre libpcap. Essa infatti offre un'interfaccia di livello superiore di gran lunga più potente e semplice da usare: catturare da rete con libpcap è questione di una decina di linee di codice. Inoltre il salvataggio su file, la gestione degli adattatori, lo *statistics mode* sono implementati in maniera immediata e di facile utilizzo. Infine, libpcap offre un avanzato supporto per la generazione dei filtri di cattura: un filtro può essere descritto con una funzione booleana in forma testuale[4], libpcap si occupa di 'compilare' la funzione ottenendo un programma per il filtro del packet driver.

Tutte queste cose, se si utilizza direttamente PACKET.DLL, devono essere fatte dal programmatore. A questo va aggiunto che i programmi basati su libpcap sono facilmente portabili verso i sistemi UNIX, dove esiste una libreria equivalente.

---

[4] Per esempio il filtro 'src host 1.1.1.1 and dst host 2.2.2.2' seleziona tutti i pacchetti spediti dall'host con indirizzo IP 1.1.1.1 a quello con indirizzo IP 2.2.2.2

Ci sono però casi in cui l'utilizzo di PACKET.DLL si rivela indispensabile. Essa permette infatti l'accesso a tutte le funzioni del driver, mentre libpcap usa solo le più importanti. Per esempio, per sfruttare le capacità di scrittura del driver libpcap non si rivela di nessun aiuto, mentre PACKET.DLL offre tutte le funzioni necessarie.

1.4. libpcap

libpcap (abbreviazione per packet capture library) è una libreria di alto livello per la cattura di pacchetti scritta originariamente per sistemi UNIX alla Berkeley University. Essa fornisce un'interfaccia di programmazione indipendente dal sistema operativo e dall'architettura di rete sottostanti. La versione originale di libpcap è usata in UNIX da tcpdump e da una miriade di altre applicazioni per l'analisi e il monitoraggio delle reti. La versione per Windows, sviluppata durante questo lavoro di tesi, mantiene la compatibilità con la versione originale ed aggiunge alcune nuove caratteristiche, fra cui:

- La possibilità di variare in qualsiasi momento, con la funzione *pcap_setbuff*, la dimensione del kernel-buffer che il Packet Capture Driver alloca per ogni sessione di cattura. Questo permette di sfruttare la versatilità di gestione della memoria dell'architettura per Windows.
- Il supporto per lo *statistics mode*. Siccome questo modo di funzionamento è presente solo nel Packet Capture Driver, la versione originale di libpcap non ne prevede l'utilizzo. La versione per Windows invece permette di usarlo in maniera semplice e trasparente.

- In Win32 si utilizza PACKET.DLL invece di accedere direttamente al driver tramite chiamate di sistema. Questo permette l'utilizzo di un'unica versione di libpcap su tutte le piattaforme Microsoft.

1.4.1. Libpcap in Windows

La versione Win32 è stata realizzata come un'aggiunta ai sorgenti originari più che come una modifica.

Ciò significa che libpcap per Windows è ancora compilabile senza problemi sotto UNIX.

Le aggiunte risiedono soprattutto nella parte di codice che comunica direttamente con l'hardware e con il sistema. In particolare, il codice di cattura e interfacciamento con l'hardware di rete sotto Windows è stato scritto da zero. In realtà questo compito non è stato eccessivamente gravoso, perché l'interfaccia offerta dal Packet Capture Driver è molto simile a quella del BPF su cui si basa libpcap originale.

Ci sono state invece alcune difficoltà nel reperimento di frammenti di codice e definizioni che nel mondo Windows non esistono. Per risolvere questo problema sono stati usati i file sorgenti del sistema operativo FreeBSD, che è una versione open source di UNIX.

Infine è stato svolto del lavoro per permettere l'uso di libpcap anche tramite il linguaggio C++. Questa caratteristica è molto importante in Windows dove questo linguaggio è usato molto più frequentemente che in UNIX.

1.5. WinDump

WinDump è il porting per Win32 di tcpdump. Il progetto di creare una versione di tcpdump che girasse sotto Windows è nato per testare le funzionalità e le prestazioni dell'architettura di cattura winpcap, ma si è evoluto ed ha acquistato importanza in seguito all'ottima accoglienza avuta dal programma nel mondo dell'analisi protocollare e della network security.

L'accoppiata tcpdump-WinDump è infatti il primo esempio di programma di cattura che gira sia sui sistemi Windows che su quelli UNIX. Inoltre la disponibilità dei sorgenti (cosa molto rara in Windows) rende il programma ideale nel campo della ricerca, perché da la possibilità di fare modifiche e aggiungere funzioni.

WinDump offre tutte le caratteristiche della versione 3.4 di tcpdump, più alcune nuove, fra cui:

- La possibilità, tramite lo switch '-D', di ottenere l'elenco degli adattatori di rete installati sul computer e la descrizione di ognuno di essi. Questo si rende necessario perché i nomi degli adattatori in Windows sono difficili da individuare e non seguono standard precisi come in UNIX.
- La possibilità di scegliere tramite il parametro '-B' la dimensione del buffer che il Packet Capture Driver allocherà per la sessione di cattura. Questa caratteristica è stata aggiunta dopo aver constato l'importanza che la dimensione del buffer ha nell'evitare la perdita di pacchetti e nell'influenzare le prestazioni. Si è pertanto deciso di dare all'utente la possibilità di decidere liberamente questa dimensione.

Per il resto, far funzionare tcpdump sotto Windows non ha presentato problemi particolari, perché basa la cattura e l'interazione con l'hardware sull'API di libpcap.

Avere una versione di libpcap per Win32 ha facilitato molto la realizzazione di WinDump.

1.6. Prestazioni

Sono stati eseguiti svariati test per rendersi conto delle prestazioni di cattura dell'architettura creata. I test hanno messo a confronto le varie versioni del Packet Capture Driver con il BPF per UNIX FreeBSD.

Le prove sono state eseguite utilizzando due PC connessi in maniera diretta tramite un collegamento Fast Ethernet come mostrato in figura 1.3

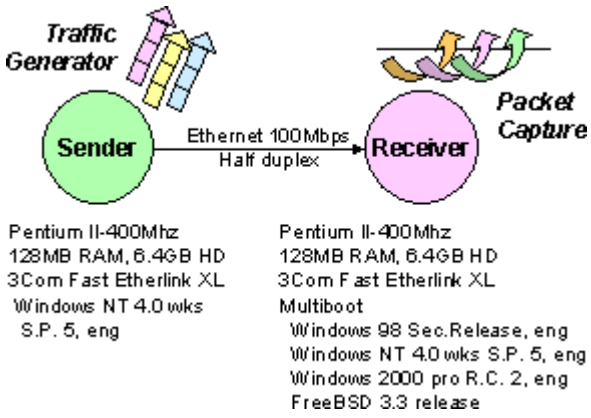

Figura 1.3: configurazione per i test

La prima delle due macchine è usata come generatore di traffico, e utilizza per spedire pacchetti un programma chiamato *Traffic generator* che sfrutta le capacità di scrittura ottimizzata del Packet Capture Driver. Questo programma fa parte del *developer's pack* di winpcap ed è scaricabile sul sito internet http://netgroup-serv.polito.it/winpcap.

La seconda macchina è invece usata per i test veri e propri. Il suo disco è stato diviso in varie partizioni, ciascuna ospitante un diverso sistema operativo. Le prove sono state effettuate sotto Windows 98 second release, Windows NT 4 workstation con service pack 5, Windows 2000 professional Release Candidate 2 e FreeBSD 3.3.

I test hanno cercato di isolare il più possibile i vari moduli che entrano in gioco durante la cattura, per poter avere dei confronti certi fra i vari sistemi. Questo però non sempre si è rivelato possibile, a causa delle differenze architetturali.

Vista la mancanza di spazio, in questa introduzione verranno mostrati risultati di due soli test, che sono particolarmente significativi. Per tutti gli altri risultati e una discussione più completa si consulti il capitolo 8.

Test 1: prestazioni del sistema di filtraggio

L'obiettivo di questo test è misurare l'impatto che il sistema di filtraggio ha sull'utilizzo del processore. WinDump e tcpdump sono lanciati con un filtro che scarta tutti i pacchetti: il driver applica il filtro a tutti i pacchetti in arrivo ma non ne passa nessuno all'applicazione. Non c'è quindi nessun overhead dovuto al processo di copia. I pacchetti che il filtro deve elaborare sono circa 67000 ogni secondo. I risultati di questo test sono mostrati in figura 1.4.

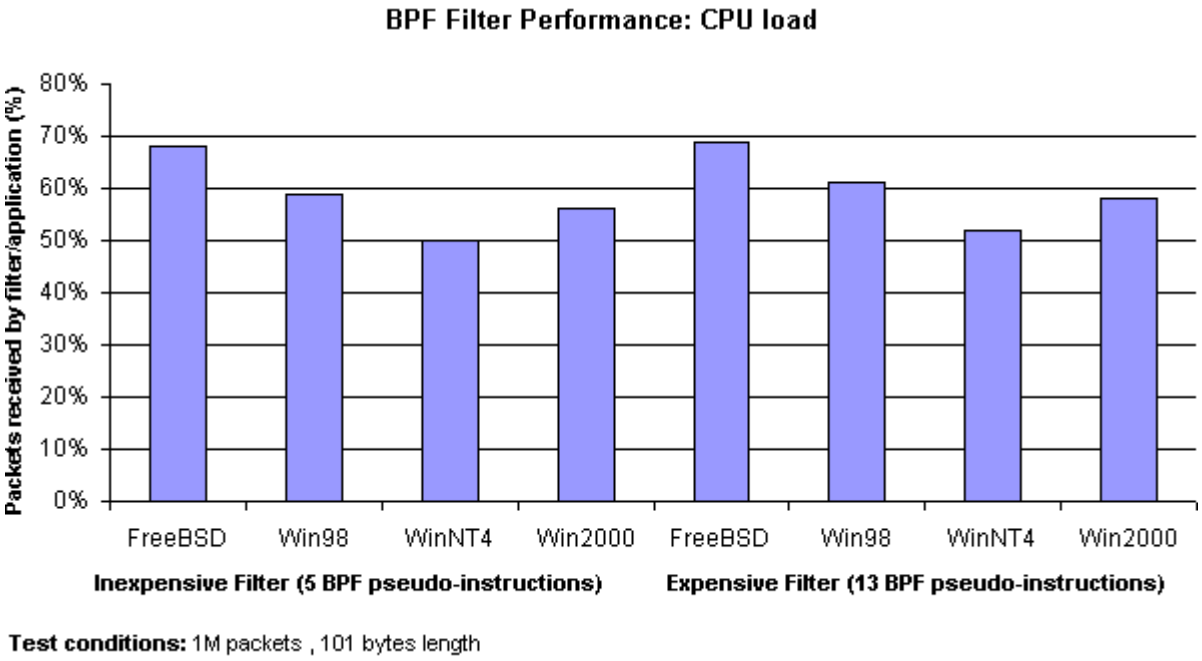

Figura 1.4: risultati del test 1

Il test è stato eseguito con due diversi filtri: uno più semplice e 'leggero' e un altro più complesso. Come si vede le prestazioni dei vari sistemi sono abbastanza simili, con un piccolo vantaggio per le architetture dove l'interazione del packet driver col sistema è leggermente più efficiente.

Test 2: Prestazioni dell'intera architettura di cattura

Questo test vuole misurare le prestazioni dell'intera architettura nella condizione più stressante che si possa incontrare. Il test prevede l'uso di WinDump e tcpdump senza filtri per salvare su disco l'intero traffico di rete.

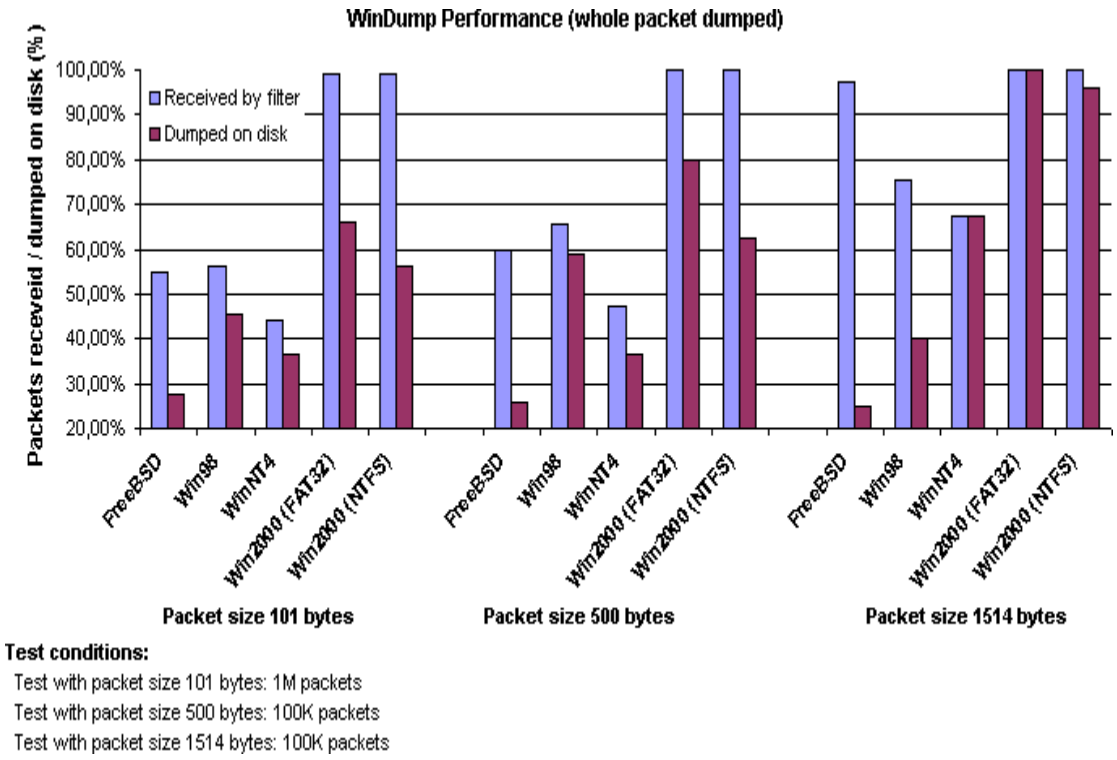

Figura 1.5: risultati del test 2

Il grafico mostra la netta superiorità delle architetture Win32, ed in particolare di Windows 2000. Questa superiorità è dovuta principalmente al miglior metodo con cui è gestito il kernel-buffer da parte del Packet Capture Driver e alla efficiente gestione dei dischi di alcuni sistemi Microsoft.

1.7. Conclusioni

WinDump e winpcap sono ormai giunti alla versione 2.02. Durante lo svolgimento di questa tesi sono state fatte molte aggiunte e migliorie, ed ormai l'architettura è abbastanza potente e stabile da poter essere usata non solo come strumento di ricerca, ma anche in applicazioni 'reali'.

La dimostrazione di questo è il successo, superiore a ogni aspettativa, ottenuto distribuendo l'architettura su Internet. Fra maggio 1999 e marzo 2000 il sito

contenente binari, sorgenti e documentazione del progetto ha avuto più di 80000 visitatori, con un numero di contatti giornalieri che ormai è stabilmente sopra i 500. Fare una stima del numero di copie di WinDump installate è molto difficile, in quanto esso è distribuito anche da tutti i maggiori siti di sicurezza e network administration. Si stima il numero di copie sia dell'ordine delle decine di migliaia.

Ancora più importante, numerosi programmi per l'analisi e il monitoraggio delle reti e per la sicurezza sono stati sviluppati usando winpcap. Alcuni sono conversioni di applicazioni già esistenti nel mondo UNIX, altri sono del tutto nuovi e usano appieno tutte le funzioni di winpcap. Fra di essi ci sono anche applicazioni commerciali.

Questi risultati, oltre a dimostrare la bontà del lavoro svolto, fanno capire l'importanza che hanno al giorno d'oggi le tecnologie per la cattura da rete e la vivacità del settore.

Per quel che riguarda i possibili sviluppi, le possibilità di lavoro su un'architettura come quella winpcap-WinDump sono veramente molte, forse perfino troppe per un team piccolo come il gruppo reti del Politecnico di Torino. Per questo motivo il codice di WinDump verrà probabilmente unificato con la versione 3.5 di tcpdump e poi abbandonato dal gruppo reti. Tcpdump.org, l'organizzazione che si occupa dello sviluppo di tcpdump, ha già richiesto di poter subentrare nello sviluppo di WinDump e di poter gestire la sua evoluzione.

Al Politecnico di Torino rimarrà invece lo sviluppo di winpcap, di gran lunga più interessante dal punto di vista della ricerca. Le possibili evoluzioni di questa architettura prevedono:

- L'implementazione di un meccanismo di sicurezza in Windows NT e Windows 2000 per arbitrare l'accesso al driver. Questa è l'ultima grossa caratteristica del BPF che ancora manca al Packet Capture Driver.

- Un'interazione più evoluta con il sistema operativo. In particolare si sta sperimentando la possibilità di gestire 'dinamicamente' il Packet Capture Driver, caricandolo in memoria solo quando serve. Questo meccanismo dovrebbe evitare le procedure di installazione e configurazione, semplificando l'utilizzo per l'utente finale.
- Lo sviluppo di una versione per Windows CE. Questo estenderebbe la protabilità delle applicazioni basate su winpcap ed aprirebbe la strada a strumenti di analisi 'tascabili'.
- Il perfezionamento del sistema di filtraggio. Il settore dei filtri per pacchetti ha avuto un notevole sviluppo negli ultimi tempi, e nuove architetture molto efficienti hanno visto la luce. Lo stato dell'arte in questo momento è il BPF+ [12], che fa uso addirittura di un compilatore JIT per convertire lo pseudocodice in codice macchina. Le ottimizzazioni del BPF+ riguardano però solo il caso di un singolo filtro, mentre non contemplano la presenza di più filtri in contemporanea. Le prestazioni in questo secondo caso possono essere ulteriormente migliorate eliminando le parti di codice in comune fra i vari filtri. Il sistema MPF cerca di fare proprio questo, anche se è limitato a situazioni molto semplici. Un'evoluzione di MPF in grado di avere un comportamento più genereale sarebbe probabilmente l'ideale per il Packet Capture Driver.
- L'estensione di winpcap per supportare la cattura in remoto. Ciò permetterebbe a una qualsiasi applicazione di cattura di essere estesa per poter ottenere il traffico di rete tramite l'interfaccia di un altro computer.

2. Introduction

Computer networks and telecommunication technologies are nowadays used in a wide range of applications. The success of the *Internet* brought networking in every house and company, and every day new applications and technologies are created.

In this scenario, the power and complexity of computer networks are growing every day. This enhances the possibilities of the final user, but makes harder the work of who has to design, maintain and make a network secure.

For this reason there is an increasing need of tools able to analyse, diagnose and test the functionality and the security of networks.

These tools, in order to perform their work, need usually to obtain the data transiting on a network, *capturing* it while the network is working. The capture process consists in obtaining, listening on the network, every transiting frame, independently from its source or destination.

The great number of transmission techniques and communication protocols complicates this task. Moreover, performance is very important in order to capture from fast networks at full speed without loosing data.

There are two main methods to capture data from a network: the first is based on the use of dedicated hardware, while the second makes use of the hardware of a normal PC or workstation connected to the communication channel. In this second method, the network adapter of the computer is used to obtain the frames from the network, and the software carries on a large amount of the capture process.

The software solution has usually lowest performance, particularly on slow machines, but it is cheaper, easier to modify and upgrade. For this reason it is widely adopted on the most used network architectures, where the performance of dedicated hardware is not needed.

This book describes a software capture architecture for the Microsoft's Win32 operating system family. This architecture includes a low-level framework - winpcap - that adds to Win32 operating systems the ability to efficiently capture data from the most used network families, and a program –WinDump - that uses winpcap to dump and analyze the network traffic.

The work takes inspiration from the *Berkeley Packet Filter* (BPF) [1], developed by S. McCanne and V. Jacobson at University of California. BPF is a well-know kernel component used as capture driver in a lot of UNIX flavors for its effectiveness, easiness to implement and its powerful user-level interface, the *libpcap* library. The most known tool exploiting this library is the famous tcpdump, widely used as platform-independent sniffer. Even though almost every UNIX platform has its own implementation of tcpdump, a version for Windows was never created because of the absence of an adequate capture framework. Such a framework is quite difficult to implement under Windows, because it needs a deep interaction with the networking portion of the kernel, which is more difficult in the Microsoft world where kernel's source code is not available.

2.1. Goals of this work

This work originates from the increasing need of high performance and cross-platform network capture tools and wants to remedy to the lack of a complete, easy to use and free capture infrastructure for the Win32 family of operating systems.

The main goals are:

- The creation of a complete capture architecture for Windows, with the functionality and performance of BPF for UNIX.
- The development of a high-performance and complete Windows version of the tcpdump network analysis tool.

The first goal aims at obtaining a system-independent capture API, able to work both under all the Windows flavors and under UNIX. This API should be powerful and easy to use, allowing to develop portable network tools without caring to low-level details. Moreover, this should help the porting toward the Windows platforms of the plethora of useful UNIX network applications based on libpcap.

The second goal wants to test the functionality and the performance of this capture API with a real and exacting network analysis tool. Moreover, it tries to satisfy the great number of requests for a Windows version of tcpdump, that is the most used tool in the UNIX world in the fields of security and network analysis. Our port allows a network manager to use the same tool on all the most diffused platforms.

The binaries and the source code of winpcap and WinDump are distributed freely under the Berkeley license, in order to encourage the use of this architecture for the development of new network tools for Windows. Moreover, a set of examples of the use of the programming interface of winpcap is provided to help the work of the developers.

All the work developed during this thesis, and the corresponding documentation is available on winpcap (http://netgroup-serv.polito.it/winpcap) and WinDump (http://netgroup-serv.polito.it/windump) web sites.

2.2. Chapters' Organization

This thesis is organized in nine chapters, plus two appendixes. Present chapter is a general introduction on the subjects discussed in the rest of the book, and has the purpose of giving a brief overview of the work. As for the other chapters, this paragraph gives a concise description of each of them.

- Chapter 3: is a detailed introduction to the work. It provides a description of the whole architecture, of the interaction between the various components, and of some related subjects like BPF in UNIX and NDIS.
- Chapter 4: describes the packet capture driver, its functions and its internal structure.
- Chapter 5: shows the structure of the PACKET.DLL dynamic link library and provides a detailed description of its API.
- Chapter 6: describes the structure of libpcap for windows and the differences from the original UNIX version.
- Chapter 7: contains practical information on the capture architecture. Gives instructions to compile the source code of winpcap and WinDump, shows how to build applications using PACKET.DLL or libpcap, and finally describes the developer's pack distributed on the web.
- Chapter 8: describes WinDump, its structure and the differences from tcpdump.
- Chapter 9: discusses the performance of the capture architecture and shows the results obtained testing it in various situations and comparing it with the UNIX BPF architecture.
- Chapter 10: draws up a balance of the work and shows possible future development.

- Appendix A: contains the programmer's manual of libpcap, describing the functions provided by the library.
- Appendix B: contains the user manual of WinDump, obtained from the manual of tcpdump.

3. Basic concepts on the architecture

Winpcap is an architecture that adds to the operating systems of the Win32 family the ability to capture the data of a network using the network adapter of the machine. Moreover, it provides to the applications a high level API that makes simpler the use of its low-level capabilities. It is subdivided into three separate components: a packet capture device driver, a low-level dynamic library and a high level static library.
This architecture can be used to create new capture tools for Windows, but also to make the porting of UNIX applications, since it is compatible with the BPF-libpcap architecture.
In this chapter I will describe the structure and the basic principle of the architecture, and I will explain some concepts that will be useful in the rest of the book.

Note: I will use the term *packet* even though *frame* is more accurate, since the capture process is done at the data-link layer and the data-link header is included in the frames received.

3.1. Structure of the capture stack

To capture packets transferred by a network, a capture application needs to interact directly with the network hardware. For this reason the operating system should offer

a set of capture primitives to communicate and receive data directly from the network adapter. Goal of these primitives is basically to capture the packets from the network (hiding the interaction with the network adapter), and transfer them to the calling programs. This is heavily system dependent so the implementation is quite different in the various operating systems. The packet capture section of the kernel should be quick and efficient because it must be able to capture packets also on high-speed LANs with heavy traffic, limiting losses of packets and using a small amount of system resources. It should also be general and flexible in order to be used by different types of applications (analyzers, network monitors, network test applications...).

The user-level capture application receives packets from the system, interprets and processes them, and outputs them to the user in a comprehensible and productive way. It should be easy to use, system independent, modular and expandable, so to support the higher number of protocols. Moreover is should allow to increase the number of decoded protocols in a simple way. These features are essential because of the high number of network protocols currently available and the speed they change, so completeness and expandability are important.

WinDump.exe is only the highest part of a packet capture stack that is composed by a module that runs at the **kernel level** and one that runs at **user level**. These two modules have different purposes and are independent and isolated one from another. The first runs at ring 0 on Intel based machines, while the second runs at ring 3 like a normal Windows program. The kernel part is Windows specific and it is very different according to various Windows flavors. The user-level part is very similar to the UNIX implementation and it is the same under Win95 and WinNT. The structure of the capture stack from the network adapter to an application like WinDump is shown in the next figure.

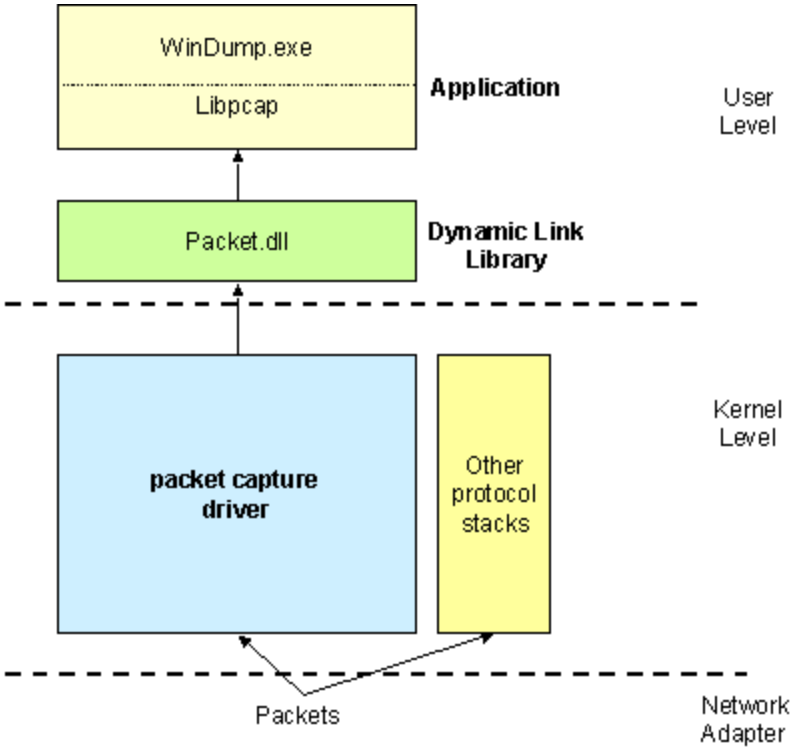

Figure 3.1: structure of the capture stack

At the lowest level there is the network adapter. It is used to capture the packets that circulate in the network. During a capture the network adapter usually works in a particular mode ('promiscuous mode') that forces it to accept all the packets instead of the ones directed to it only.

Packet Capture Driver is the lowest level software module of the capture stack. It is the part that works at kernel level and interacts with the network adapter to obtain the packets. It supplies the applications a set of functions used to read and write data from the network at data-link level.

PACKET.DLL works at the user level, but it is separated from the capture program. It is a dynamic link library that isolates the capture programs from the driver providing a system-independent capture interface. It allows WinDump to be executed on different Windows flavors without being recompiled.

The pcap library, or libpcap, is a *static* library that is used by the packet capture part of the applications. It uses the services exported by PACKET.DLL, and provides to the applications a higher level and powerful capture interface. Notice that it is statically linked, i.e. it is part of the application that uses it.

The user interface is the higher part of the capture program. It manages the interaction with the user and displays the result of a capture.

We will now describe these modules, their behavior and the architectural choices that affect them.

3.1.1. User level: WinDump program and the libpcap library

The WinDump program is the higher part of the capture stack and manages the interaction between the user and the system. It gets the user's inputs (for example, which packets must be captured and in which way they must be showed to the user) from the command line and outputs the results on the screen. The WinDump executable running on a Windows platform is identical to the TcpDump executable running on a UNIX workstation from the user viewpoint. This means that the two programs have almost the same input parameters and the same output format.

TcpDump on UNIX makes use of a library for the packet capture process, called Packet Capture library, or *pcap library,* or *libpcap*, a system-independent interface for user-level packet capture. Libpcap provides a set of functions independent from the hardware and the operating system that an application can use to capture packets from a network. It is important to remember that original libpcap is a general-purpose capture library, and is used by TcpDump, as well as several other network tools and applications. TcpDump does not interact with hardware directly, but uses the functions

exported by libpcap to capture packets, set packet filters and communicate with the network adapter. Basically it is system-independent and can easily be ported to any system on which the libpcap library is available. For this reason the WinDump project includes a full porting of the libpcap library toward the Win32 platform. Furthermore, the pcap library is not limited to be used with WinDump, and we think that it can be very useful to people that want to convert network monitors and analyzers from the UNIX world, and it can be also a powerful base to create new network tools for the Win32 environment. Therefore, we tried to maintain the structure of the original version in our implementation. We modified only the section of libpcap that communicates with the kernel, implementing the interaction with the NDIS packet capture driver. An important characteristic of libpcap for Win32 is that there is only a version that works both on Windows 95, 98, NT and 2000. This can be obtained by putting a dynamic link library, called PACKET.DLL, between libpcap and the capture driver, so that the system calls to the driver are the same in the various Windows environments. In this way a network tool based on libpcap will work on Windows 95 and Windows NT without any modifications.

The porting of TcpDump, once a working implementation of libpcap for Win32 was ready, was not too difficult. Only few improvements were introduced in WinDump, to handle the difference in the names of the network adapters and the dynamic kernel buffer.

3.1.2. Kernel level: the NDIS packet capture driver

The basic role of the kernel part of the capture stack is to take the link-layer packets from the network and to transfer them to the application level without modifica-

tions. We implemented it as a kernel driver (*packet.sys*) under Windows NT and as a Virtual Device Driver (*packet.vxd*) under Windows 95. Applications can have access to the capture driver with read and write primitives and can consider the network adapter somehow similar to a normal file, reading or writing the data that comes from the network. The capture driver can be used by any Win32 application that needs to capture packets, and is not limited to the use with WinDump or Analyzer. The interaction with the packet capture driver usually passes through the PACKET.DLL dynamic link library. The DLL implements a set of functions that make simpler the communication with the driver, avoiding the use of things such system calls or IOCTLs.

The packet capture driver interacts with the network adapter's device drivers through NDIS, that is a part of the network code of Win32. NDIS is responsible of the management of the various network adapters and of the communication between the adapters and the software portions that implement the protocols.

A basic network capture driver can be quite simple. It needs only to read the packets from the network driver and copy them to the application. However, in order to obtain acceptable performances, substantial improvements need to be done to this basic structure. The most important are:

- To limit packet loss, the driver should be able to store the incoming packets in a buffer because the user-level application could not be ready to process them at their arrival. Buffered packets will be transferred as soon as the application will be ready.

- In order to minimize the number of context switch between the application (that runs in user mode) and the driver (that runs in kernel mode), it should be possible to transfer several packets from the buffer to the application using a single read call.

- The user-level application must receive only the packets it is interested in, usually a subset of the whole network traffic. An application must be able to specify the type of packets it wants (for example the packets generated by a particular host) and the driver will send to it only these packets. In other words the application must be able to set a *filter* on the incoming packets, receiving only the subset of them that satisfy the filter. A packet filter is simply function with a boolean returned value applied on a packet. If the returned value is TRUE, the driver copies the packet to the application, otherwise the packet is ignored.

The implementation of these features and the architecture of the driver were inspired by the BSD Packet Filter (BPF) of the UNIX kernel, of which we make a brief description in the next paragraph.

3.2. BPF in Unix

BPF, or BSD Packet Filter, described in [1], is a kernel architecture for packet capture proposed by Steven McCanne and Van Jacobson. It was created to work in UNIX kernels and exports services used by TcpDump and many other UNIX network tools.

BPF is essentially a device driver that can be used by UNIX applications to read the packets from the network through the network adapter in a highly optimized way. It is an anomalous driver because it does not have a direct control on the network adapter: the adapter's device driver itself calls the BPF passing it the packets.

BPF has two main components: the network tap and the packet filter.

3.2.1. The network tap

The network tap is a callback function that is a part of the BPF code but it is not executed directly by BPF. It is invoked by the network adapter's device driver when a new packet arrives. The adapter's driver MUST call the network tap for each packet, otherwise BPF will not work on the adapter.

The network tap collects copies of packets from the network device drivers and delivers them to listening applications. The incoming packets, if accepted by the filter, are put in a buffer and are passed to the application when the buffer is full.

3.2.2. The packet filter

The filter decides if a packet should be accepted and copied to the listening application. Most applications using BPF reject far more packets than those accepted, therefore good performance of the packet filter is critical for a good over-all performance. The packet filter is a function with boolean output that is applied to a packet. If the value of the function is true the kernel copies the packet to the application; if it is false the packet is ignored. BPF packet filter is quite more complex, because it determines not only if the packet should be kept, but also the number of bytes to keep. This feature is very useful if the capture application does not need the whole packet. For example, a capture application is often interested only in the headers and not in the data of a packet. Such an application can set a filter that accepts only the firsts bytes of the packet so that only the headers are copied. This action speeds up the capture process and decrease the loss probability because it decreases the number of byte to copy from the driver to the application and decreases the space needed to store the packet into the buffer.

Historically there have been two approaches to the filter abstraction: a boolean expression tree and a directed acyclic control flow graph or CFG. More details about them can be found in [1]. These two models of filtering are computationally equivalent, i.e., any filter that can be expressed in one can be expressed in the other. However implementation is very different: the tree model can be easily mapped into code for a stack machine while the CFG model can be mapped into a register machine code. BPF creators choose CFG because it can be implemented more efficiently on modern computers that are register based. The BPF pseudo-machine is a virtual processor, and its abstraction consists of an accumulator, an index register (x), a scratch

memory store, and an implicit program counter. It is able to execute load and store instructions, branches, arithmetic instructions and so on. Therefore, a UNIX application that wants to set a filter on the incoming packets has to build a program for the pseudo-machine and send it to BPF through an IOCTL call. BPF executes the filter program on each packet and discards the ones that do not satisfy the filter. The BPF pseudo-machine has some nice features:

- It is protocol independent. This means that the kernel has not to be modified to add new protocol support.
- It is general enough to handle unforeseen uses.
- It is optimized for speed, to have good performances.

3.2.3. Overview of the BPF capture process

Figure 3.2 illustrates BPF's interface with the rest of the system.

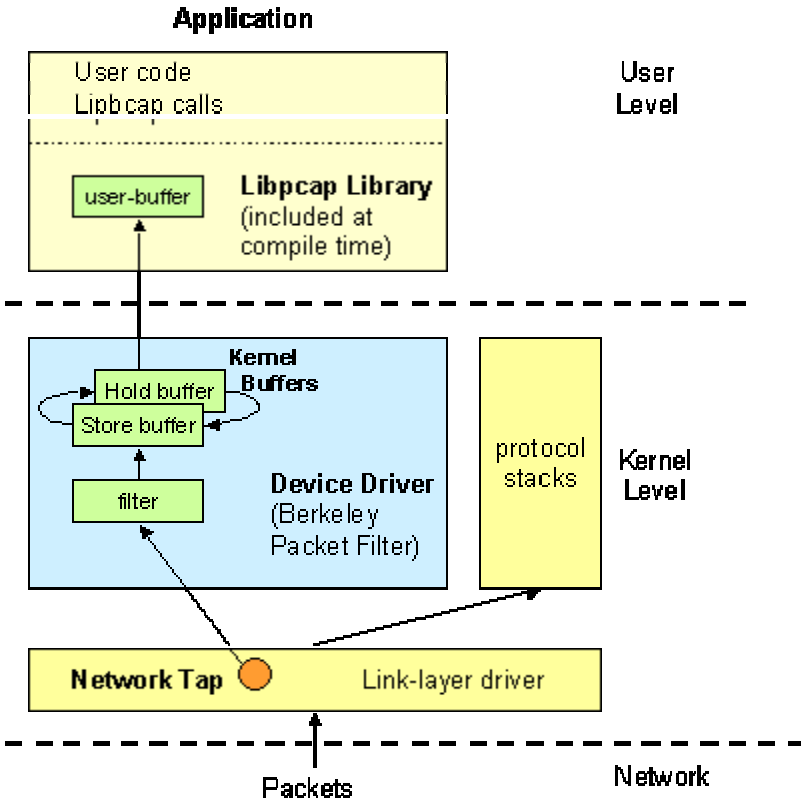

Figure 3.2: structure of BPF

BPF associates a filter and two buffers to every capture process that requests its services. The filter is created by the application and passed to BPF through an IOCTL call. The buffers are statically allocated by BPF and their dimension is usually 4 KB. The first of the two buffers (called the *store* buffer) is used to receive the data from the adapter, and the second (called the *hold* buffer) is used to copy the packets to the application. When the store buffer is full and the hold buffer is empty, BPF swaps them. In this way the process does not interfere with the adapter's device driver.

When a packet arrives at a network interface, the link level device driver normally sends it up to the system protocol stack. But when BPF is listening on this interface, the driver first calls BPF's network tap function.

The tap feeds the packet to each participating application's filter. This user-defined filter decides whether a packet is to be accepted and how many bytes of each packet

should be saved. Notice that the filter is applied to the packet while it is still in the link-level driver's memory, without copying it. This optimizes greatly performances and memory usage, because in case the packet is not accepted, it is discarded before any copy. If the filter accepts the packet, the tap copies the number of bytes specified by the filter from the link-level diver's memory to the store buffer associated with that filter. At this point the interface's device driver re-obtains control and the normal protocol processing proceeds.

A packet is discarded if the store buffer is full, and the hold buffer is not available (i.e., the process is still reading data from it).

The process performs a read system call to receive packets from BPF. When the hold buffer is full (or when a special timeout elapses), BPF copies it to the process' memory and awakes the process. An application can receive more then one packet at a time. To maintain packet boundaries, BPF encapsulates the captured data from each packet with a header that includes a time stamp, length, and offsets for data alignment.

3.2.4. An important consideration

Notice that not all UNIX versions have BPF (i.e. filtering and buffering capabilities in the kernel), but the pcap library can compensate this lack. The architecture is able to work by filtering packets in a BPF compatible way at user level. This solution was adopted by the first release of the NDIS packet driver. It is easier to implement but it has limited performances particularly for two reasons:

- The filtering process is done at user level, so that each packet must be copied from the kernel to the application buffer before detecting if the user application wants it. This clearly, wastes CPU time and memory. If the filtering proc-

ess is done in the kernel, all the packets not needed by the capture application are discarded by the system and are not copied to the user level.

- There is no buffering for the packets in the kernel. In a multitasking environment, the capture application must share processor time with other programs. It is possible that the capture application is not being executed when a packet arrives. Furthermore, the application could be doing some other task and could not be waiting for the packet. In absence of a kernel buffer, these situations bring to the loss of the packet.

These reasons brought to the implementation of the filtering process and the buffering of the packets in the packet capture driver and not in WinDump.

3.3. Interaction with NDIS

NDIS (Network Driver Interface Specification) is a set of specifics that defines the communication between a network adapter (or, better, the driver that manages it) and the protocol drivers (IP, IPX...). Main NDIS purpose is to act as a wrapper that allows protocol drivers to send and receive packets onto a network (LAN or WAN) without caring either the particular adapter or the particular Win32 operating system.

NDIS supports three types of network drivers:

1. **Network interface card or NIC drivers**. NIC drivers directly manage network interface cards, referred to as NICs. The NIC drivers interface directly to the hardware at their lower edge and at their upper edge present an interface to allow upper layers to send packets on the network, to handle interrupts, to reset the NIC, to halt the NIC and to query and set the operational characteristics

of the driver. A NIC driver cannot communicate with user-mode applications, but only with NDIS intermediate drivers or protocol drivers. NIC drivers can be either miniports or legacy full NIC drivers.

- Miniport drivers implement only the hardware-specific operations necessary to manage a NIC, including sending and receiving data on the NIC. Operations common to all lowest level NIC drivers, such as synchronization, is provided by NDIS. Miniports do not call operating system routines directly; their interface to the operating system is NDIS. A miniport does not keep track of bindings. It merely passes packets up to NDIS and NDIS makes sure that these packets are passed to the correct protocols.

- Full NIC drivers have been written to perform both hardware-specific operations and all the synchronization and queuing operations usually done by NDIS. Full NIC drivers, for instance, maintain their own binding information for indicating received data.

2. **Intermediate drivers**. Intermediate drivers interface between an upper-level driver such as a legacy transport driver and a miniport. To the upper-level driver, an intermediate driver looks like a miniport. To a miniport, the intermediate driver looks like a protocol driver. An intermediate protocol driver can layer on top of another intermediate driver although such layering could have a negative effect on system performance. A typical reason for developing an intermediate driver is to perform media translation between an existing legacy transport driver and a miniport that manages a NIC for a new media type unknown to the transport driver. For instance, an intermediate driver could

translate from LAN protocol to ATM protocol. An intermediate driver cannot communicate with user-mode applications, but only with other NDIS drivers.

3. **Transport drivers or protocol drivers**. A protocol driver implements a network protocol stack such as IPX/SPX or TCP/IP, offering its services over one or more network interface cards. A transport driver services application-layer clients at its upper edge and connects to one or more NIC driver(s) or intermediate NDIS driver(s) at its lower edge.

Next figure shows a sample NDIS structure with two capture stacks on the same network adapter: one with the NIC driver and a protocol driver, the other with the NIC driver, an intermediate driver and a protocol driver.

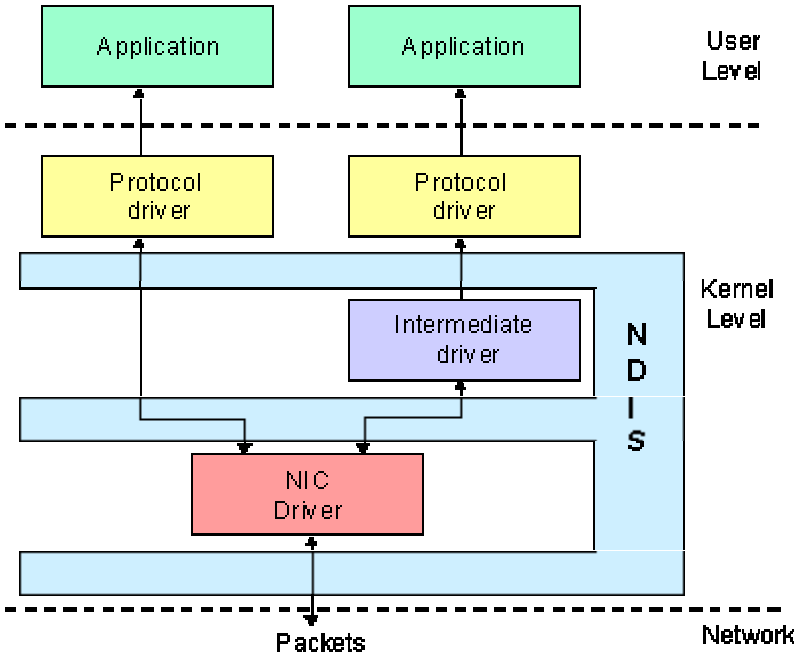

Figure 3.3: simple NDIS structure

The packet capture driver needs to communicate both with the network drivers (to get data from the net) and with user-level applications (to provide them the packets), so it is implemented in the NDIS structure as a protocol driver. This allows it to be

independent from the network hardware, thus working with all the network interfaces supported by Windows. Notice however that the packet capture driver works at the moment only on Ethernet adapters, loopback adapters and on some WAN connections due to limits imposed by the driver's and filter's architecture. Notice also that a WAN connection is usually seen by the protocol drivers as an Ethernet NIC, and every received packet has a fake Ethernet header created by NDIS. This allows to the protocol drivers written for Ethernet to work on WAN connections without any change, but implies also that specific packets like PPP NCP-LCP are not seen by the protocol drivers, because the PPP connection is virtualized. This means that the packet driver cannot capture this kind of packets.

Notice that the various Win32 operating systems have different versions of NDIS: the version of NDIS under Windows 95 is 3.0, while Windows NT has NDIS 4 and Windows 2000 has NDIS 5. NDIS 4 and 5 are supersets of NDIS 3, therefore a driver written to work with NDIS 3 (usually) works also with NDIS 4 and 5. The packet capture driver is written for NDIS 3, but works also with the more recent versions of NDIS. This means that the interaction between the driver and NDIS is the same under Windows 95/98 and under Windows 2000.

Next figure shows the position of the packet capture driver in the Win32 architecture.

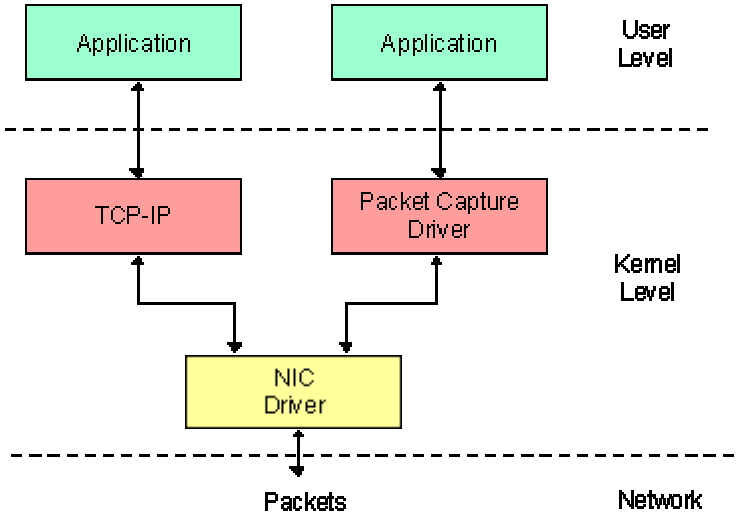

Figure 3.4: the Packet Capture Driver in NDIS

A protocol driver that communicates with lower level NDIS drivers uses NDIS-provided functions. For instance, a protocol driver must call *NdisSend* or *NdisSend-Packets* to send a packet or packets to a lower level NDIS driver.

Lower level drivers, on the other hand, communicate with the protocol driver in an asynchronous way. They indicate the arrival of a new packet by calling a callback function of the protocol driver and passing a pointer to a lookahead buffer, its size, and the total size of the received packet. The name of this callback function in the packet capture driver is *Packet_tap*. The behavior of the packet driver is however quite different from the one of a standard protocol driver. In fact:

- The packet driver receives and processes all the packets that are currently transferred in the network. Such a behavior can be obtained setting the adapter in 'promiscuous mode', i.e. forcing it to accept all the packets from the network. A standard protocol driver manages only packets (unicast and multicast) directed to or coming it and the broadcast ones.

- The packet driver does not implement a protocol, but it stores the packets and transfers them *as they are*, with their timestamp and their length, to the appli-

cations. A standard protocol driver removes the various headers from the packets and passes to the applications only the data. Packet driver leaves the headers and copies them to the applications with the captured data.

Note that in UNIX implementation, BPF is called *before* the protocol stack, directly by the network interface's driver. This is not possible for the packet capture driver, that is *a part* of the protocol stack. That is why the packet capture driver is not able to capture PPP specific packets, because it does not work at hardware level, but on top of NDIS. Having the precedence on NDIS would imply changes in the kernel or in the NIC drivers, which is not possible in Windows.

3.4. Compatibility

Supported Operating Systems

Versions of the packet capture driver exist for all the main Win32 operating systems: Windows 95,98 NT4 and 2000.

The Windows 9x version consists in a virtual device driver (*.vxd*) that can be installed in Windows 95 and Windows 98. Windows NT4 and Windows 2000, on the other hand, need a different drivers (*.sys*) because the installation and the initialization processes are different.

A version for Windows CE is currently under development.

Hardware compatibilty

The packet capture driver was developed to work primarily with Ethernet adapters. Support for other MACs was added during the development, but Ethernet remains the

preferred one. The main reason is that all our development stations have Ethernet adapters so all our tests were made on this type of network. However, the current situation is:

- Windows 95/98: the packet driver works correctly on Ethernet networks. It works also on PPP WAN links, but with some limitations (for example it is not able to capture the LCP and NCP packets).

- Windows NT4/2000: the packet driver works correctly on Ethernet networks. We were not able to make it working on PPP WAN links, because of a binding problem on the NDISWAN adapter. Support for FDDI, ARCNET and ATM was added starting from version 2.02; however we did not test them because we do not have the hardware. Do not expect them to work perfectly.

Token Ring is not supported, and we don't plan to support it in the near future.

4. The Packet Capture Driver for Windows

The Packet Capture Driver adds to Windows kernels the capability to capture raw packets from a network in a way very similar to UNIX kernels with BPF. In addition, it provides some functions, not available in the original BPF driver, to help the development of network test and monitor programs. Main goals of the packet capture driver are high capture performance, flexibility and compatibility with the original BPF for UNIX.

The result is a protocol driver that can:

- capture the raw traffic from the network and pass it to a user level application
- filter the incoming packets executing BPF pseudo-machine code. This means that the capture application can define a standard BPF program and pass it to the driver. The driver will discard the incoming packets that do not satisfy the filter
- hold the packets in a buffer when the application is busy or it is not fast enough to sustain the flow of packets coming from the network
- collect the data from several packets and return it as a unit when the application does a read. To maintain packet boundaries, packets are encapsulated in a header (the same used by BPF) that includes a time stamp, length, and offsets for data alignment.

- write raw packets to the network
- calculate statistics on the network traffic

4.1. Structure of the driver

This paragraph will describe the structure of the driver and the main architectural chioces.

4.1.1. Basic architecture of the various versions

The basic structure of the driver is shown in the next figure.

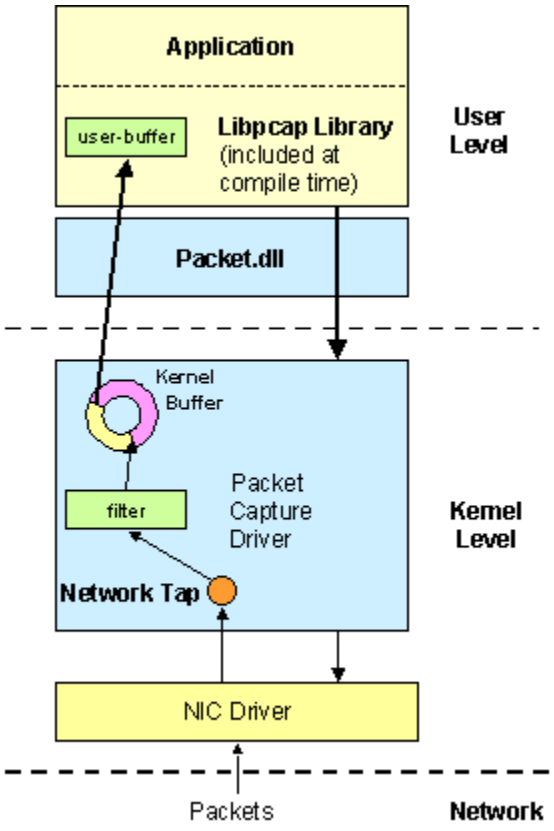

Figure 4.1: structure of the driver

The arrows toward the top of the picture represent the flow of packets from the network to the capture application. The bigger arrow between the kernel buffer and the application indicates that more than one packet can transit between these two entities in a single read system call. The arrows toward the bottom of the picture indicate the path of the packets from the application to the network. The thick arrow between the kernel buffer and the application indicates that, in particular situations, more than one packet can transit between these two entities in a single write system call. WinDump and libpcap do not send packets to the network, therefore they use only the path from the bottom to the top. The driver however is not limited to the use with WinDump and can be used to create new network tools. For this reason it has been included the possibility to write packets, that can be exploited through a direct use of the PACKET.DLL dynamic link library.

The structure shown in Figure 4.1 (i.e. *a single* adapter and *a single* application) is a simplified description of the packet driver. The real structure is more complex and can be seen in figure 4.2. This figure shows the driver's configuration with two network adapters and two capture applications.

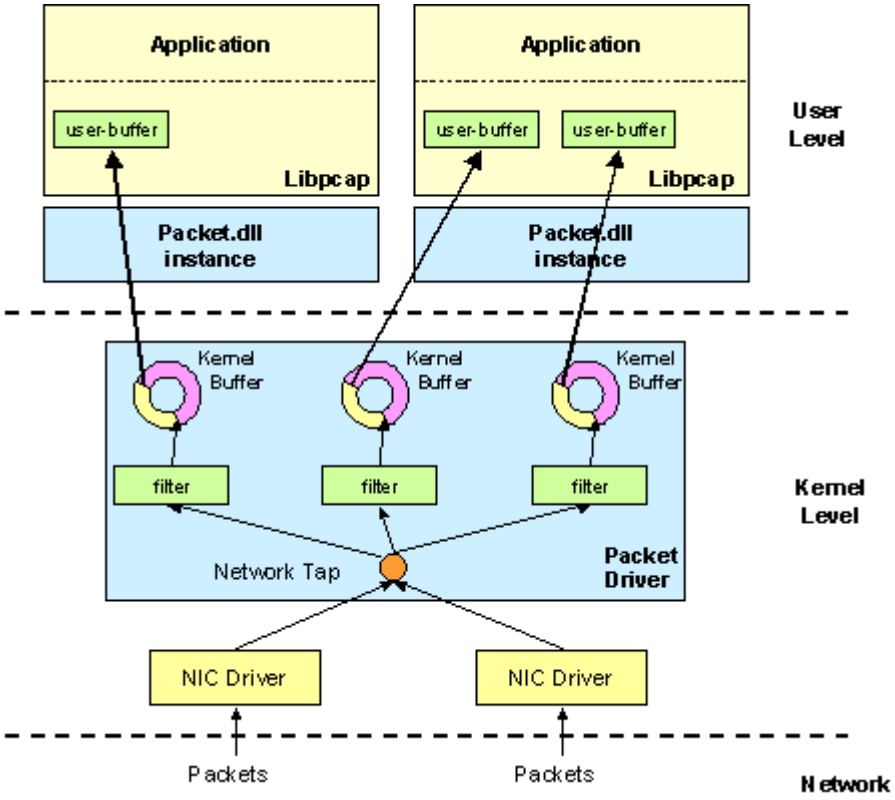

Figure 4.2: structure of the driver with two adapters and two applications

For each capture session established between an adapter and a capture program, the driver maintains a filter and a buffer. The single network interface can be used by more than one application at the same time. For example, a user that wants to capture the IP and the UDP traffic and save them in two separate files, can launch two sessions of WinDump on the same adapter (but with different filters) at the same time. The first session will set a filter for the IP packets (and a buffer to store them), and the second a filter for the UDP packets. It is also possible to write an application that, interacting with the capture driver, is able to receive packets from more than one interface at the same time.

Notice that in versions older than 2.02, the only configuration possible in Windows 95 and Windows 98 was the one shown in Figure 4.1 (i.e. *a single* network adapter and *a single* capture application). This was due to limitations in the architecture of

those versions. From version 2.02, in Windows 95/98 it is possible, like in Windows NT, to have more instances of the driver, and this allows to have more capture applications working at the same time. Furthermore, it is possible for a single application to work on more than one network adapter.

Core structure between the various Windows flavors is quite similar. Internal data structures are not very different, the packet buffer and filter are handled in the same way. The interaction with NDIS is very similar under the different platforms and is obtained by a set of callback functions exported by the driver and a group NDIS library functions (*NdisTransferData*, *NdisSend*...) used by the packet driver to communicate with the NIC driver. Differences between the different flavors concerns the interaction with the other parts of the operating system (read and write call handling from user-level applications, timer functions...), since the philosophy of the various operating systems is quite different.

4.1.2. The filtering process

The filter mechanism present in the packet capture driver derives directly from the BPF filter in UNIX, therefore all the things already said about the BPF filter are still valid. An application that needs to set a filter on the incoming packets can build a standard BPF filter program (for example through a call to the *pcap_compile* function of libpcap) and pass it to the driver, and the filtering process will be done at kernel level. The BPF program is transferred to the driver through an IOCTL call with the control code set to *pBIOCSETF*. A very important thing is that the driver needs to be able to verify the application's filter code. In fact, as we said, the BPF pseudo-machines can execute arithmetic operations, branches, etc. A division by zero or a

jump to a forbidden memory location, if done by a driver, bring inevitably to a blue screen of death. Therefore, without any protection, a buggy or bogus filter program could easily crash the system. Since the packet capture driver can be used by any user, it could be easy for an ill-intentioned person to cause damages to the system through it. For this reason, each filter program coming from an application is checked by the *bpf_validate* function of the driver before being accepted. If a filter is accepted, the driver stores the filter program and executes it on each incoming packet, discarding the ones that do not satisfy the filter's conditions. If the packet satisfy the filter, it is copied to the application, or put in the buffer if the application is not ready to receive it. If no filter is defined, the driver accepts all the incoming packets.

The filter is applied to a packet when it's still in the NIC driver's memory, without copying it to the packet capture driver. This allows to reject a packet before any copy thus minimizing the load on the system.

A very interesting feature of the BPF filter exploited by the packet capture driver is the use of a numeric return value. When a filter program is applied to a packet, the BPF pseudomachine tells not only if the packet must be transferred to the application, but also the *length* of the part of the packet to copy. This is very useful to optimize the capture process, because only the portion of packet needed by the application is copied.

The source code of the filter is in the file *bpf_filter.c*, and derives from the corresponding file of the BPF's source code. This file contains three main functions: *bpf_filter*, *bpf_filter_with_2_buffers* and *bpf_validate*.

- *bpf_filter*: the filtering function that is normally used by the capture driver. It implements the register machine that executes the filtering code. It receives two buffers: one containing a packet and one containing the BPF program to

execute. It returns the length of the packet's portion to store, or zero if the packet must be discarded.

- *bpf_filter_with_2_buffers*: this function is very similar to the *bpf_filter* function, but it can filter packets whose header and data are stored in different buffers. It receives three buffers: one containing the header of the packet, one containing the data of the packet and one containing the BPF program to execute. This function is slower, but is more general than the *bpf_filter* function. It is needed because of the particular architecture of NDIS, and is used by the capture driver in some particular situations in which the header and the data of the packets are stored by the underlying driver in different buffers. This can happen for example in the case of ATM LAN emulation, in which the Ethernet header is built up by software level and can be separate from data. For more details see the description of the Packet tap function in section 4.3.

- *bpf_validate*: the function that checks a new filter program, returning true only in case of valid program. This procedure checks that the jumps are within the code block, that memory operations use valid addresses, and that constant divisions by 0 are not present in the code.

4.1.3. The reading and buffering processes

When an application wants to obtain the packets from the network, it performs a *read* call on the NDIS packet capture driver (this is not true in Windows 95, where the application retrieves the data through a IOCTL call; however the result is the same). This call can be synchronous or asynchronous, because the driver offers both the possibilities. In the first case, the read call is blocking and the application is stopped until

a packet arrives to the machine. In the second case, the application is not stopped and must check when the packet arrives. The usual and RECOMMENDED method to access the driver is the synchronous one because of the difficulties in implementing the asynchronous one that can bring to errors. The asynchronous method can be used to do application-level buffering, but this is usually not needed because the buffering in the driver is more efficient and clean. WinDump uses the synchronous method, and all the considerations that we will make apply only to this method.

The packet driver supports timed reads. An application can set the read timeout through an IOCTL with code pBIOCSRTIMEOUT. If the timeout is different from 0, every read call performed by the application will return when the timeout expires even if no packets were received.

After an incoming packet is accepted by the filter, the driver can be in two different situations:

- The application is ready to get the packet, i.e. it executed a read call and is currently asleep waiting for the result of the call. In this case the incoming packet is immediately copied in the application's memory, the read call is completed and the application is waked up.

- There is no pending read at the moment, i.e. the application is doing other stuff and is not blocked waiting for a packet. To avoid the loss of the packet, it must be stored in the driver's buffer and transferred to the application when it will be ready to receive it, i.e. at the next read call.

The driver uses a circular buffer to store the packets. A packet is stored in the buffer with a header that maintains information like the timestamp and the size of the packet. Moreover, a padding is inserted between the packets in order to word-align them to increase the speed of the copies. The size of the buffer at the beginning of a

capture is 0. It can be set or modified in every moment by the application through an IOCTL call. When a new dimension of the buffer is set, the packets currently in the buffer are lost.

Incoming packets are discarded by the driver if the buffer is full when a new packet arrives. The dimension of the driver's buffer affects HEAVILY the performances of the capture process. In fact, it is likely that a capture application, that needs to make operations on each packet, sharing at the same time the processor with other tasks, will not be able to work at network speed during heavy traffic or bursts. This problem is more noticeable on slower machines. The driver, on the other hand, runs at kernel level and is written explicitly to capture the packets, so it is very fast and usually it does not loose packets. Therefore an adequate buffer in the driver can store the packets while the application is busy, can compensate the slowness of the application and can avoid the loss of packets during bursts or high network activity.

If the buffer is not empty when the application performs a read system call, the packets in the driver's buffer are copied to the application memory and the read call is immediately completed. More than one packet can be copied from the driver's circular buffer to the application with a single read call. This improves the performances because it minimizes the number of reads. Every read call in fact involves a context switch between the application and the driver, i.e. between user mode (ring 3 on Intel machines) and kernel mode (ring 0). Since these context switches are quite slow, decreasing their number means improving the capture speed. To maintain packet boundaries, the driver encapsulates the captured data from each packet with a header that includes a timestamp and length. The application must be able to properly unpack the incoming data. The data structure used to perform the encapsulation is the same used by BPF in the UNIX kernel, so the format of the data returned by the driver is the same returned by BPF in UNIX.

Notice that a single circular buffering method like the one of the BPF capture driver, compared with a double buffering method, allows a better use of the memory. The double buffer method is suited when the buffers are very small (like in BPF): in this case the frequency of the swaps between buffers is high and the memory is used efficiently. When buffers become big, this method wastes a lot of memory and is not efficient because the operations on a buffer can take a lot of time, impeding the use of the buffer's memory. The circular buffer method uses the memory more efficiently with big buffers because the free memory is always available.

The packet driver handles the kernel-level and user-level packet buffers in a very versatile way. It is possible to choose any dimension for the driver's circular buffer, limited only by the RAM of the machine. Furthermore, also the buffer used by the user-level application can have any size and can be changed whenever the application needs to do so. An important feature is that the two buffers are not forced to have the same size. The packet capture driver detects the dimension of the application's buffer and fills it in the right way even when it has different size from the circular buffer. This is important when the driver's buffers is big, because a large application buffer would be a waste of memory. On the other hand, this mechanism has also a drawback: since the size of the packets is not fixed, when the dimension of the application's buffer is smaller than the number of bytes in the driver's buffer, it can happen that the driver has to scan the headers of the packets in its buffer in order to determine the amount of bytes to copy. This process slows down a bit the capture process, therefore best performance are obtained when application and kernel buffers have the same size. In fact, the driver detects when the dimension of the application's buffer is greater than the number of bytes in the driver's buffer, and when this is true, copies all the data without performing any scan. To avoid this problem, before the beginning of

a copy the driver tries to determine the amount of data to copy without scanning the buffer. If this operation is successful, a small amount of CPU time is saved.

From version 2.01, a function (called *PacketMoveMem*) to perform the copies between the driver and the application was introduced. This functions has an important feature: it updates the head of the circular buffer while copying. In this way the driver has not to wait the end of a copy process to use the memory freed by it. In this way buffer's memory is used better and the loss probability is lower.

4.1.4. The writing process

The packet capture driver allows to write raw data to the network. This can be useful to test either the network or the protocols and applications working on it. To send data, a user-level application performs a write system call (a write IOCTL call in Windows 95/98) on the packet driver's device file. The data is sent to the network as is, without encapsulating it in any protocol, therefore the application will have to build the various headers for each packet. The application does not need to generate the FCS because it is calculated by the network adapter hardware and it is attached automatically at the end of a packet before sending it to the network.

Notice that the sending rate of the packets to the network is not very high because of the need of a system call for each packet. Version 2.02 of the driver allows (on Windows NT/2000 only) to send a single packet more than one time with a single write system call. The user-level application can set, with an IOCTL call (code pBIOCSWRITEREP), the number of times a single packet will be sent to the network. For example, if this value is set to 1000, every raw packet wrote by the application on the driver's device file will be sent 1000 times. This feature can be used to generate

high speed traffic because the overload of the context switches is no longer present, and is particularly useful to write tools to test networks, routers, and server programs. This optimization is made only in Windows NT and Windows 2000, while it is emulated at user level in PACKET.DLL in Windows 95/98. This means that an application that uses the multiple write method will run also in Windows 9x, but with a very low write speed compared to WindowsNTx.

An example of the use of the multiple write feature is the Traffic Generator (TG) application, whose source code can be found in the developer's pack [15].

4.1.5. Other functions

In addition to the functions described in the previous paragraphs, the packet capture driver offers a pair of minor functions to grant full compatibility with original BPF:

- The driver associates a timestamp to each packet that satisfies the filter. The timestamp should be associated to the packet before it is put in the buffer because the buffering process tends to distort the time variations. Furthermore, it is important to get the timestamp of a packet as soon as possible, to obtain a precise reference. Therefore taking the timestamp is one of the first operations that the driver executes when a new packet arrives. The precision of the timestamp is 1 microsecond, a value compatible with the UNIX implementation. The timestamp is associated by the driver with the packet data and they are passed together to the application when a read call is performed.

- Every instance of the driver counts the number of the packets received from the network and the number of the packet that were discarded. A packet is dis-

carded when the application is not ready to get it and the buffer of the driver is full. In this situation the packet cannot be copied to the application and cannot be stored in the buffer and the driver must reject it. These values are requested by the pcap library and are very useful to understand the performances of the capture process and to see how many packets were lost. The application can obtain in any moment the value of these two counters through an IOCTL call with the control code set to *pBIOCSETF*. WinDump prints these values on the screen when the user terminates the capture process.

4.1.6. 'Statistics' mode

A network manager is often not interested in the whole sequence of packets transiting on a network, but in statistical values about them. For example the user could be interested in network's utilization, broadcast level, but also in more refined information, like amount of mail traffic or number of web requests per second. 'Statistics' mode, or *mode 1*, is a particular working mode of the BPF capture driver that can be used to perform real time statistics on the network's traffic in an easy way and with the minimum impact on the system.

To put the driver in statistics mode, the user-level application makes an IOCTL call with code pBIOCSMODE, and the value '1' as input parameter. To set again the normal 'capture' working mode the same IOCTL, but with '0' as input parameter, can be called.

When in mode 1, the driver does not capture anything but limits itself to count the number of packets and the amount of bytes that satisfy the user-defined BPF filter. These values are passed to the application at regular intervals, whenever a timeout

expires. The default value of this timeout is 1 second, but it can be set to any other value (with a 1 ms precision) with an IOCTL call (parameter pBIOCSRTIMEOUT). The counters are encapsulated in a `bpf_hdr` structure before being passed to the application. This allows to have microsecond-precise timestamps, and easy interaction with libpcap. libpcap is in fact able to decode the `bpf_hdr` structure, and pass the counters to the application.

Statistics mode is a powerful and versatile framework to calculate statistics on the network traffic. It is powerful because it requires very few system resources: only the BPF filter is applied to every packet, but no copy is performed. This ensures a low processor usage and avoids the need of any kernel buffer. It is versatile because it is based on the BPF filter machine. This allows the user to easily calculate complex statistics setting only the proper filter, and makes the use of this feature fully configurable from user-level.

To know more about how to use statistics mode, see chapter 5. An example of the use of statistics mode is NetMeter, a simple monitor application available in the developer's pack [5].

4.2. Differences from the original BPF

Since Windows network interface drivers are shipped as binary files, the programmer does not have access to the network driver sources in order to insert any call to the Network Tap. The Network Tap in Windows is configured as a new protocol driver, so that NDIS threats it like IP or IPX protocols. In this way a single instance of a protocol stack is able to exploit different network technologies. Choosing NDIS made programming easier because the packet driver is able to use the same set of

NDIS functions on all the platforms. The interaction with NDIS is however quite heavy, and this makes the packet capture driver (and above all the crucial 'tap' function) more complex than original BPF.

On the other hand, choosing to capture packets at protocol level allows the capture driver to work without any requirement from the network adapter's driver, while original BPF requires that the adapter's driver calls directly the *tap* function.

The location of the network tap influences the behavior of the capture driver. For example PPP (Point to Point Protocol), the most used protocol for managing serial links, uses some auxiliary protocols (LCP, Link Control Protocol, and NCP, Network Control Protocol) in order to configure and maintain the link up. These packets do not reach the NDIS upper layer because they are generated directly by the network link driver, therefore the Network Tap cannot capture them. Vice versa, UNIX tap is able to capture everything that is being transmitted on the wire because the tap call is inserted into the "write" and "read" functions of the network interface's driver.

4.2.1. The kernel-level buffer

BPFs in UNIX and Windows differ significantly in the management of the kernel and user packet buffers. As seen in chapter 3, BPF in Unix has the kernel buffer made by 2 blocks of memory, whose dimension is usually 4 KB. The main drawbacks of this architecture are the following:

- The standard size of the driver's buffer is 8 KB and the maximum size is 32 KB. This size is quite small compared to the amount of RAM available in modern computers.

- the choice to subdivide the kernel buffer in 2 blocks allows a fast copy process, but it does not make a good use of memory (i.e. it is possible to have a portion of free memory that cannot be used by BPF).

To get over these limitations, our architecture makes use of a single dynamic circular buffer (default size 1 Mbytes). The increased size is possible thanks to the larger memories available nowadays and to the dynamic management of that buffer, which is allocated at the beginning of a capture and deallocated when the capture ends. This choice could make, in certain situations, less efficient copies from kernel to user space because of the impossibility to copy buffers with predetermined size, but it makes better use of memory.

Having a small buffer penalizes the original BPF architecture especially in case the application is not able to read as faster as the driver captures for a limited time interval. Such situation is common when data has to be transferred to disk (or even to screen) or when the network has bursty traffic. WinDump is expected to have a significantly better behavior than TCPdump in these conditions. Moreover WinDump is expected to be more affordable on slow machines in case it is deployed together with more complex (and slower) network analysis tools.

Another difference between UNIX and Windows is the mechanism to move packets from kernel to user space. In UNIX, packets captured by BPF are copied to the application either when the hold buffer is full or when a timeout expires, a technique that can be seen as a *delayed write*. In this way the application is awakened either when a certain number of packets has arrived or when a certain time interval has elapsed. This guarantees a low frequency of system calls and therefore a low processor usage, but it makes the capture application less responsive. Moreover this choice

increases the risk of packets lost because there can be less free space into the kernel buffers (that are not freed as soon as possible) when a burst arrives.

A capture application in Windows is blocked *only* if the kernel buffer is empty; therefore it receives data as soon as the kernel buffer contains anything. If the application is very fast or if the frequency of incoming packets is not high, the amount of data present in the kernel buffer (and immediately passed to the application) could be very low. In this situation, the number of system calls used by the read process in Windows could be higher than in UNIX. This ensures a better use of the kernel buffer and a more responsive behavior of the capture application, but generates a higher processor load. Therefore, WinDump is expected to use more processor resources than TCPdump for low packets' frequencies, but approximately the same for high frequencies.

4.2.2. The user-level buffer

User buffer is implemented by Libpcap for UNIX as a *fixed-size* buffer with the same size of the two kernel buffers (usually 4Kb). Having both user and kernel buffer the same size, the copy process can be optimized. The entire kernel buffer is copied in a single read call, decreasing the number of system calls (i.e. of context switches between user and kernel mode).

In our architecture a user buffer with the same size of the kernel buffer could be a waste of memory because the kernel buffer can be very large. Therefore the user buffer can have any size, and usually is smaller than the kernel one (because kernel buffer is more important to avoid packet loss). If needed, the user can set a user buffer

with the same dimension of the kernel buffer like in UNIX. This wastes more memory, but minimizes the overhead of the copy process and the number of system calls, allowing better performances. Default size of the user buffer is 256Kbytes.

4.2.3. The Filter

The packet filter used by the packet driver under Windows is essentially the same used by BPF in UNIX. This means same processor and memory usage. *bpf_filter_with_2_buffers* is a bit slower, above all with long filtering programs, but is used only with very particular network interfaces.

4.3. Driver's source code

Warning: the following section of the documentation is quite complex, because it is intended for people that need to modify or upgrade the packet capture driver. If you don't need to do this and you are not interested in the driver's internals, you can skip this chapter.

4.3.1. Introduction

The versions of the packet capture driver for the various Windows flavors are quite similar in the structure and in the organization of the source code. This derives from the fact that the interaction with the underlying levels of the network stack is handled using NDIS services and primitives that are the same in the various operating systems

of the Windows' family. This means that functions provided by NDIS are exactly the same in the various Windows operating systems; also the interaction with the network hardware is based on the same mechanisms.

What is different in the two versions is the interaction with the non-network parts of the operating system and with the user-level applications. In particular:

- The interaction with the operating system is often completely different under the two platforms. An example is the system timer: the packet capture driver must obtain the value of the system timer to get the timestamps of the packets. In Windows NT/2000 this is done by a call to the kernel function *KeQueryPerformanceCounter*(). In Windows 95/98 the driver obtains the value of the timer calling a service of the Virtual Timer Device, *VTD_Get_Real_Time*, through a function written in assembler. The two approaches give the same result, but are very different in the implementation.

- The interaction with the applications is different in the various platforms. In Windows NT/2000 the packet driver is seen by the applications as a network service. This means that an application uses the driver like a normal file: it reads and writes data from the network using the *readfile* and *writefile* system calls. Particular actions like setting a filter or a buffer in the driver are handled through IOCTL functions. In Windows 95 the packet capture driver is seen by the applications as a protocol. All the operations on the driver, included reading and writing packets, are performed through IOCTL calls.

As we said, the structure of the different drivers is quite similar, i.e. the data structure and the functions used are the same. However the implementation of some functions and the use of some data structures can be very different. In the next chapters we

will describe the most important functions and structures, trying to make a system-independent explanation of them.

Note: a IOCTL, (or IO Control) function, is the method used on most operating systems (including the UNIX family) to perform operations different from the standard read and write (for example to set parameters or retrieve values) on a driver. In Win32 an application performs an IOCTL call with the *DeviceIoControl* system call, passing as parameters the handle of the driver, the IOCTL code and the buffers used to send or receive data from the driver.

4.3.2. Organization of the source code

The following files are present and have the same meaning both in the Windows 95/98 and in the Windows NT/2000 driver's source code:

File	Description
DEBUG.H	This file contains the macros for used for the debug.
PACKET.H	Contains the declarations of functions and data structures used by the driver.
PACKET.C	This is the main driver's file. It contains the DriverEntry function, which starts and initializes the driver, the functions to query the registry and the IOCTL handler function.
WRITE.C	Contains the functions to send packets.
READ.C	Contains the functions to read packets.
OPENCLOS.C	Contains the functions to open or close an instance of the driver.
BPF.H	Defines the data structures and the values used by the BPF filter functions.
BPF_FILTER.C	This file contains the BPF filter's procedures.

The following files are present only in the Windows 95/98 version:

File	Description
LOCK.C	This file contains the routines to lock the driver's buffers.
REQUEST.C	Contains the functions to perform OID query/set operations on the adapter.
NDISDEV.ASM	This file contains the assembler function *C_Device_Init*, that is called by the operating system when the driver is loaded to begin the initialization process.
FUNCTS.ASM	This file contains a couple of assembler functions used to call the services of the Virtual Timer Device to obtain the system timer's value.

4.3.3. Data structures

In this paragraph there is a simple description of the most important data structures of the driver. The structures will not be described in deep because our goal is to give an overview of them and to point out the location of the most important driver's variables.

The main data structures of the packet capture driver are:

- System structures
- OPEN_INSTANCE
- INTERNAL_REQUEST
- DEVICE_EXTENSION
- timeval

- BPF structures

These structure are used by both the versions of the driver.

System structures used by the driver

The following is a list of the kernel and NDIS data structures mostly used by the driver. We provide only a short description of them and of their use in the driver. Detailed descriptions can be found in the manuals of NDIS and of Windows 95 and Windows NT DDKs.

Name	Description
NDIS_PACKET	This NDIS structure defines the packet descriptors with chained buffer descriptors for which pointers are passed to many Ndis*Xxx*, Miniport*Xxx*, and Protocol*Xxx* functions. A protocol driver that wants to write a packet to the network through the NDIS primitives must encapsulate it in a NDIS_PACKET structure. Similarly, a protocol driver receives a packet from the underlying NIC drivers in a NDIS_PACKET structure. Therefore, this structure is used heavily by the *PacketRead* and *PacketWrite* functions.
NDIS_PROTOCOL_ CHARACTERISTICS	This NDIS structure is used by the *DriverEntry* function during the initialization of the packet capture driver and contains all the information that NDIS will need to interact with the driver: the name of the protocol, the requested version of NDIS, the pointer to the various callback functions, and so on. This data is passed to NDIS through the *NdisRegisterProtocol* function.
LIST_ENTRY	This is a kernel structure defined both in Windows 9x and in Windows NTx, to support handling of doubly linked lists. The kernel provides a set of primitives to handle lists, insert and remove elements, and so on. To use these primitives the LIST_ENTRY data structure must be used. All the linked lists of the drivers are handled through this structure.
IRP	This structure is used only by the NT/2000 versions of the driver. IRP, or I/O Request Packet, is the fundamental structure that the Windows NTx kernels use for the communication between applications and drivers. All the data received by the driver from an application are packed in an IRP. All the data that the driver passes to the applications must be packed in an IRP. IRP is a VERY complex structure that contains all the data needed to communicate with the user-level: addresses of buffers, process IDs...

The OPEN_INSTANCE structure

This structure is use by almost all the functions of the driver. It contains the variables and the data associated with a running instance of the driver. The OPEN_INSTANCE structure is enough to identify completely and univocally an in-

stance of the driver. A new instance of the driver with its own OPEN_INSTANCE structure is associated to every application that performs *CreateFile()* system call on the packet capture driver. This means that more than one application can have access to the packet capture driver.

This structure has some differences in the two versions of the driver, because it has some system-dependent fields. However the most important members are present in both the versions.

The following are the main fields:

Variable	Datatype	Description
AdapterHandle	NDIS_HANDLE	The NDIS identifier of the adapter on which current instance is working.
PacketPool	NDIS_HANDLE	Pointer to a list of NDIS_PACKET structures (see the documentation of NDIS for more details about NDIS_PACKET) that the driver can use to receive or send packets to the adapter.
RcvList	LIST_ENTRY	Pointer to the list of pending reads that the driver must satisfy
RcvQSpinLock	NDIS_SPIN_LOCK	Spin lock used to protect the RcvList from simultaneous access by driver functions
Received	Int	Number of packets received by current instance from its opening, i.e. number of packet received by the network adapter since the beginning of the capture.
Dropped	Int	Number of packet that current instance had to drop from its opening. A packet is dropped if there is no more space to store it in the circular buffer that the driver associates to current instance.
Bpfprogram	PUCHAR	The BPF filter program associated with current instance of the driver.
StartTime	LARGE_INTEGER	This field contains a time reference used to

		convert the NTx and 95x system timer's format into the BPF timestamp's format.
Buffer	PUCHAR	Pointer to the memory that contains the circular buffer associated with this driver's instance. The packets captured from the interface and accepted by the filter are put in this buffer.
BufSize	UINT	Dimension in bytes of the circular buffer.
Bhead	UINT	Head of the circular buffer. This variable indicates the first valid byte of data in the circular buffer.
Btail	UINT	Tail of the circular buffer. This variable indicates the last valid byte of data in the circular buffer. Notice that this value can be smaller than Bhead, since the tail of a circular buffer can be below the head.
BlastByte	UINT	Pointer to the last byte of data in the memory allocated for the buffer. This variable is used to store the end of the buffer when Btail is smaller than Bhead. This is needed because of the non-fixed size of the captured packets, so the buffer can have an empty portion (too small for a packet) at its end.
TimeOut	int	Read timeout in ms. Every read on current instance of the driver expires after the number of milliseconds hold in this variable is elapsed.
ReadTimeoutTimer	KTIMER in WinNTx UINT in Win9x	The timer associated with the last read. Since the access to the driver is supposed to be synchronous, only a single read can be pending. This variable holds the kernel timeout associated to this read. Note that the format of this field in Windows NTx is different than Windows 9x, because kernel timers are handled in different ways.
mode	int	The working mode of current instance of the driver. There are two possible values: 0 (normal capture mode) and 1 (statistics mode).
Nbytes	__int64	This field is used when current instance of the driver is in mode 1, to store the amount of bytes accepted by filter.

Npackets	__int64	This field is used when current instance of the driver is in mode 1, to store the number of packets accepted by filter.
Nwrites	UINT	Contains the number of times a single write must be repeated. See *The writing process* section for more details.

The INTERNAL_REQUEST structure

This structure is used by the driver to perform OID query and set operations on the network adapter through the underlying NIC driver. The query/set operations on the adapter can be done usually only by protocol drivers, but the packet capture driver allows the user-level applications to use this mechanism through an IOCTL function. The driver uses the INTERNAL_REQUEST structure to store the information on a query/set operation requested by the application. To know more about the OID requests see the documentation of PACKET.DLL.

The INTERNAL_REQUEST structure has an important field that is present both in the Windows 95 and in the Windows NT versions:

Variable	Datatype	Description
Request	NDIS_REQUEST	Contains all the data needed to perform a query/set operation on the adapter, like the pointers to the buffers for the information and the number of bytes to read or write (see the NDIS documentation for a detailed description of the NDIS_REQUEST structure).

When the driver needs to read or set a parameter of the network adapter, it must build a proper NDIS_REQUEST structure and send it to the NIC driver through the *NdisRequest* NDIS library function.

The DEVICE_EXTENSION structure

This structure contains information on the packet capture driver, like the pointer to the DRIVER_OBJECT structure that describes the driver, or the ProtocolHandle that NDIS associates to the driver. It is used by the *DriverEntry* and *PacketUnload* functions to initialize or uninstall the driver.

The timeval structure

This structure is used to store the timestamp associated with a packet. It has two fields:

Variable	Datatype	Description
tv_sec	long	Holds the date of the capture in the standard UNIX time format (number of seconds from 1/1/1970).
tv_usec	Long	Holds the microseconds of the capture.

BPF data structures

The packet capture driver uses for the interaction with user-mode applications a set of data structures originally created for BPF in UNIX. These structures are:

- bpf_insn
- bpf_program
- bpf_hdr
- bpf_stat

For a description of these structures see chapter 5.

4.3.4. Functions

This paragraph will describe the procedures of the packet capture driver. All the following functions are present in both the versions of the driver:

- DriverEntry
- PacketOpen
- PacketOpenAdapterComplete
- PacketClose
- PacketCloseAdapterComplete
- PacketResetComplete
- PacketUnload
- bpf_validate
- bpf_filter
- bpf_filter_with_2_buffers
- PacketIoControl
- PacketWrite
- PacketSendComplete
- PacketRead
- PacketMoveMem
- ReadTimeout

- Packet_tap
- PacketTransferDataComplete
- PacketRequestComplete

These functions are present only in the Windows 95 version of the driver:

- PacketReset
- PacketRequest
- PacketGetMacNameList
- QuerySystemTime
- GetDate

The next function is present only in the Windows NT version of the driver:

- PacketCancelRoutine

Notice first of all that usually *PacketXXX* functions have a corresponding *PacketXXXComplete* procedure. This is due to the mechanism that NDIS defines for the interaction between NIC drivers and protocol drivers. This mechanism implies an *asynchronous* interaction, which takes place in two times:

1. The protocol driver asks for a service to the NIC driver invoking the proper function of the NDIS library. For example, the protocol driver can use the *NdisSend* function to send a packet to the network, the *NdisRequest* function to query or set a parameter of the adapter, and so on. Since these functions are non blocking (they return immediately without giving a result), the protocol driver is keeping on working.

2. The NIC driver, after its work is concluded, communicates the end of the operation and sends the results invoking the callback function of the protocol driver associated with that kind of request.

A protocol driver has several callback functions to get the result of the various operations it can perform on the NIC diver. The names of these functions in the packet capture driver end with the word "*Complete*". Therefore *PacketResetComplete* is the callback function called by the NIC driver after the end of a reset operation of the adapter that was started by the *PacketReset* function, *PacketSendComplete* is called after the conclusion of a send operation started by the *PacketWrite* function with a call to *NdisSend*, etc.

The most important callback function of the packet capture driver is *Packet_tap*. It is called by the NIC driver each time a new packet arrives from the network to transfer the packet to the capture driver. This function does the main work in the packet capture driver.

The interaction with NDIS and with the NIC drivers is very similar in Windows 95/98 and in Windows NT because NDIS is a system independent architecture that provides the same interface to the protocol drivers written for all the Win32 platforms. This means that the section of the packet capture driver that interacts with the underlying levels of the network stack is quite similar in the Windows 95 and Windows NT versions.

The interaction with the upper levels, i.e. with the user-level applications, is instead quite different in the two versions. The reason of this is that the interaction between an application and a driver in Windows 95 uses different mechanisms than in Windows NT. In Windows NT the application opens and initializes instances of the packet capture driver, reads packets, writes packets, performs IOCTL calls respectively with the *CreateFile*, *ReadFile*, *WriteFile* and *DeviceIoControl* system calls. The driver has

to manage various types of requests from the application. In Windows 95 all the interaction between the packet capture driver and the applications (included open, close, read and write operations) is done through IOCTL calls using the *DeviceIoControl* system call, therefore the driver has to manage only this type of calls.

DriverEntry

This procedure is called by the operating system when the packet capture driver is loaded and started. It makes all the initializations needed by the driver. In particular this function allocates a NDIS_PROTOCOL_CHARACTERISTICS structure and initializes it with the protocol data (version, name, etc.) and the addresses of all the callback functions of the driver. This structure is then passed to NDIS with a call to *NdisRegisterProtocol*. The Windows NT version of the driver calls also the *IoCreateDevice* function to pass to the operating system the addresses of the handlers for the open/close, read, write and IOCTL requests.

PacketOpen

This function is called when a new instance of the driver is opened. It allocates and initializes variables and buffers needed by the new instance, fills the OPEN_INSTANCE structure associated with it and opens the adapter. The time constants used to obtain the timestamps of the packets during the capture are initialized here. This initialization is really brain damage because it requires the conversion of dates from the internal format of Windows NT (100-nanosecond intervals since January 1, 1601) and Windows 95 (milliseconds since January 1, 1980) to the UNIX one (seconds since January 1, 1970), used by the applications. At this point the network adapter is opened here with a call to *NdisOpenAdapter*.

PacketOpenAdapterComplete

Callback function associated with the *NdisOpenAdapter* function of NDIS library. It is invoked by NDIS when the NIC driver has finished an open operation that was started previously by the *PacketOpen* function with a call to *NdisOpenAdapter*.

PacketClose

This function is called when a running instance of the driver is closed. It stops the capture and buffering process and deallocates the memory buffers that were allocated by the *PacketOpen* function. The network adapter is closed here with a call to *NdisCloseAdapter*.

The corresponding callback function is *PacketCloseAdapterComplete*.

PacketReset

Resets the adapter associated with the current instance, calling the *NdisReset* function of NDIS library. This function is defined only in Windows 95, because the Windows NT version of the driver calls *NdisReset* directly from the *PacketIoControl* function.

The corresponding callback function is *PacketResetComplete*.

PacketUnload

This function is called by the system when the packet capture driver is unloaded. It frees the DEVICE_EXTENSION internal structure, and calls *NdisDeregisterProtocol* to deregister from NDIS.

bpf_validate

This function checks a new filter program, returning true if it is a valid program. See the paragraph on the filtering process for more details. This function is exactly the same on all the platforms.

bpf_filter

The filtering function that is normally used by the capture driver. The syntax of this function is:

```
u_int bpf_filter(register struct bpf_insn *pc,
            register u_char *p,
            u_int wirelen,
            register u_int buflen)
```

pc points to the first instruction of the BPF program to be executed, *p* is a pointer to the packet on which the filter is applied, *wirelen* is the original length of the packet, and *buflen* is the current length of the packet (*wirelen* and *buflen* are different if the filter is invoked before the whole packet has arrived).

bpf_filter returns the number of bytes of the packet to be accepted. If the result is 0 the packet must be discarded. If the result is -1 the whole packet must be accepted.

See the paragraph on the filtering process for more details.

bpf_filter_with_2_buffers

The alternative filtering function, with two input data buffer. The syntax of this function is:

```
u_int bpf_filter_with_2_buffers(register struct bpf_insn *pc,
                                register u_char *p,
                                register u_char *pd,
                                register int headersize,
                                u_int wirelen,
                                register u_int buflen)
```

pd is the pointer to the beginning of packet's data. *headersize* is the size of the header. All the remaining parameters, and the return values are the same of the *bpf_filter* function.

This function is used by the *Packet_Tap* procedure only if the packet's header and data are stored by the NIC driver in two different buffers. This can happen in Windows that does not specifies that the two buffers must be consecutives (even if usually it does). In the case of ATM LAN emulation, in which the Ethernet header is built by the software level and can be separate from data, this could not be verified. In such a case, a filtering function like *bpf_filter_with_2_buffers* is needed by the packet driver. *bpf_filter_with_2_buffers* is not the default filtering function and is used only if needed, because it is noticeably slower than *bpf_filter*. It must in fact perform a check to determine the correct memory buffer every time the filter program needs to access to the packet data. For more details see the description of the Packet tap function.

PacketIoControl

Once the packet capture driver is opened it has to be configured from user-level applications with IOCTL commands using the *DeviceIoControl* system call. This function handles IOCTL calls from the user level applications. This function is particularly important in the Windows 95/98 version of the driver, where it manages read and write requests.

Once the code of the command is obtained, a `switch` case determines the operation to be performed.

Next table summarizes the main IOCTL operations used in both the versions of the driver.

Command	Description
BIOCSETF	This function is used by an application to set a new BPF filter in the driver. The filter is received by the driver in a buffer associated with the IOCTL call. Before allocating any memory for the new filter, the bpf_validate function is called to check the correctness of the filter. If this function returns TRUE, the filter is copied to the driver's memory, its address is stored in the bpfprogram field of the OPEN_INSTANCE structure associated with current instance of the driver, and the filter will be applied to all the incoming packets. Before returning, the function empties the circular buffer used by current instance to store packets. This is done to avoid the presence in the buffer of packets that do not match the filter.
BIOCSETBUFFERSIZE	This function is used by an application to set the new size of the kernel buffer that is used by current instance of the driver. The new size is received by the driver in a buffer associated with the IOCTL call. This function deallocates the old buffer, allocates the new one and resets all the parameters associated with the buffer in the OPEN_INSTANCE structure. Since the old buffer is deallocated, all the currently buffered packets are lost.
BIOCGSTATS	Returns to the application the number of packets received and the number of packets dropped by current instance of the driver. Values passed to the application are the *Received* and *Dropped* fields of the OPEN_INSTANCE structure associated with current instance.
BIOCSMODE	Changes the working mode of current instance of the driver. This function sets the *mode* field of the OPEN_INSTANCE structure. The new working mode is received by the driver in a buffer associated with the IOCTL call. Mode 0

	is normal capture mode and it is the default one. Mode 1 is statistics mode. Unless differently specified, in mode 1 a default 1000 ms timeout is set.
BIOCSRTIMEOUT	Sets the value of the read timeout in milliseconds. This function sets the *TimeOut* field of the OPEN_INSTANCE structure. The new timeout is received by the driver in a buffer associated with the IOCTL call. The read timeout has two different meanings depending if current instance is in mode 0 or in mode 1. If the instance is in mode 0, the timeout indicates the amount of time after which a read system call expires. When the read expires, the user-level application is awaked and receives the content of the kernel buffer, even if it is empty. If the instance is in mode 0, the timeout indicates the length of the time interval after which the instance's statistic counters are passed to the application. When the interval has elapsed, the statistic counters are packet in a *bpf_hdr* structure with a timestamp, and are passed to the application.
BIOCSWRITEREP	Sets the number of times a single write call must be repeated. This function sets the *Nwrites* field of the OPEN_INSTANCE structure, and is used to implement the 'repeated write' feature of the driver.
IOCTL_PROTOCOL_RESET	Resets the adapter associated with current instance of the driver.
IOCTL_PROTOCOL_SET_OID	This call is used to perform a OID set operation on the adapter's NIC driver. The code of the operation and the parameters are received by the driver in a buffer associated with the IOCTL call. The result is returned to the application without changes.
IOCTL_PROTOCOL_QUERY_OID	This call is used to perform a OID query operation on the adapter's NIC driver. The code of the operation and the parameters are received by the driver in a buffer associated with the IOCTL call. The result is returned to the application without changes.

The following IOCTL operations are defined only in the Windows 9x version:

Command	Description
IOCTL_OPEN	Called by the user-level applications when the driver is opened for a new capture. First, this function calls *GetNDISAdapterName* to retrieve the NDIS internal name of the adapter. In fact, in Windows 9x the name that NDIS uses for the an adapter is different from the adapter's device name. At this point *PacketOpen* is invoked to open the adapter and initialize it.
IOCTL_CLOSE	Called by the user-level applications when the driver is closed after a capture. It calls the *PacketClose* function to close the adapter and free the resources allocated by current instance of the driver.
IOCTL_PROTOCOL_READ	Called by user-level applications to read packets from the network. Calls the *PacketRead* function.
IOCTL_PROTOCOL_WRITE	Called by user-level applications to write a packet to the network. Calls the *PacketWrite* function.
IOCTL_PROTOCOL_MACNAME	Called by user-level applications to obtain the names of the NIC drivers on which the packet capture driver can work. Calls the *PacketGetMacNameList* function.

PacketWrite

This function is used to send packets to the network. The packet to send is passed to the driver with the *WriteFile* system call in Windows NTx, and with an IOCTL_PROTOCOL_WRITE IOCTL call in Windows 9x.

In Windows 9x, PacketWrite allocates a NDIS_PACKET structure and associates to it the data received from the application. Then The *NdisSend* function from the

NDIS library is used to send the packet to the network through the adapter associated with the current driver's instance.

In Windows NTx, the behavior of this function is influenced by the *Nwrites* field of the OPEN_INSTANCE structure associated with current instance of the driver. If *Nwrites* is 1, the behavior is exactly the same of the Windows 9x PacketWrite. If *Nwrites* is greater than 1, it indicates the number of time the write must be repeated. In this case, PacketWrite sends sets of 100 packets before waiting any answer from the NIC driver, and repeats this behavior until all the packets are sent. This is done to speed up the generation process, that is fast enough to saturate a 10Mb Ethernet network (with 64 bytes packets) with a low level Pentium machine.

PacketSendComplete

Callback function associated with the *NdisSend* function of NDIS library. It is invoked by NDIS when the NIC driver has finished to send a packet to the network, after *PacketWrite* was called. This function frees the NDIS_PACKET structure that was allocated in the *PacketWrite* function, and awakens the application from the *WriteFile* or *DeviceIoControl* system call.

PacketRead

This function is invoked when the user level application performs a *ReadFile* system call (in Windows NTx), or an IOCTL_PROTOCOL_WRITE IOCTL call (in Windows 95x). In both cases the application must provide a buffer that will be filled by the driver with the packets coming from the network. The behavior of this function is the same when the driver is in mode 0 and when it is in mode 1.

The first operation performed by *PacketRead* is a check on the driver's circular packet buffer associated with the current instance of the driver. There are two possible cases:

1. The packet buffer is empty. This is ALWAYS true if current instance of the driver is in mode 1. The application's request cannot be satisfied immediately, and the system call must be blocked until at least a packet arrives from the net or the timeout on this read expires. PacketRead sets the timeout on this read calling *KeSetTimer* in Windows NTx, and the custom function *SetReadTime-Out* in Windows 9x. After a NDIS_PACKET structure is allocated, the buffer received from the application is mapped in the driver's address space and associated with this structure. The NDIS_PACKET structure is put in the linked list of pending reads. It will be extracted from this list by the *Packet_Tap* function when a new packet will be captured. At this point *PacketRead* returns without awaking the application.

2. The packet buffer contains data: the application's request can be satisfied immediately. First, the driver obtains the length of the buffer passed by the application, to which the packets will be copied. Knowing this value, *PacketRead* tries to copy all the data to the application without further operations. This is possible if the number of bytes present in the driver's circular buffer is smaller than the size of the buffer passed by the application. If the application's buffer is too small to contain all the data present in the driver's packet buffer, a scan on the kernel buffer is performed to determine the amount of bytes to copy. After the scan, *NdisMoveMemory* is invoked to copy the packets. At this point the application is awaked and *PacketRead* returns.

PacketMoveMem

This function is used by all versions of the driver to copy data from the driver's circular buffer to the user-level application's buffer. It copies the data minimizing the accesses to the RAM. This is obtained aligning the memory accesses at the dword. Furthermore, it updates the head of the circular buffer every 1024 bytes copied. In this way the circular buffer is updated during the copy allowing a better use of the circular buffer and a lower loss probability. The function has been written to have a low overhead compared to a normal copy function. For this reason the head is updated no more often than every 1024 bytes.

ReadTimeout

This function is automatically invoked by the kernel when the timeout associated with a read call expires. First of all, the OPEN_INSTANCE structure associated with the current instance of the driver is found. From this structure ReadTimeout determines the working mode of current driver's instance. There are two possibilities.

- Current driver's instance is in mode 0. No packet passed the filter during the timeout period. In fact, if a single packet had passed the filter, the timeout would have been deleted in the packet_tap function. For the same reason, the driver's circular buffer must be empty. The read system call associated with current instance is completed, but an empty buffer is passed to the application.

- Current driver's instance is in mode 1. ReadTimeout builds in the user-level buffer a bpf_hdr structure, and fills it with the current timestamp and with the value 16 for bh_caplen and bh_datalen. After the header, two 64 bit integers are put. The first is the count of the number of packets satisfying the filter (hold in the *Npackets* field of the OPEN_INSTANCE structure associated with current instance), while the second is the amount of bytes satisfying the filter

(hold in the *Nbytes* field of the OPEN_INSTANCE structure). Finally, *Npackets* and *Nbytes* are reset and the read system call associated with current instance is completed.

PacketCancelRoutine

This is the cancel routine set in the PacketRead function with a call to *IoSetCancelRoutine*. It is called by the operating system when the read system call is cancelled, for example when the user-level application is closed during a read on the packet capture driver. PacketCancelRoutine removes the pending IRPs from the queue and completes them. This function was introduced to correct a bug of the old versions of the driver. Without this function, in fact, the driver hangs after the user-level application is closed until at least a packet arrives from the network. This is a Windows NT specific problem, so this function is present only in the Windows NT version of the driver.

Packet_tap

Packet_tap is invoked by the underlying NIC driver when a packet arrives to the network adapter. In Windows 95 and in Windows NT it has the same following syntax:

```
NDIS_STATUS Packet_tap ( NDIS_HANDLE
ProtocolBindingContext,
                NDIS_HANDLE MacReceiveContext,
                PVOID HeaderBuffer,
                UINT HeaderBufferSize,
                PVOID LookAheadBuffer,
                UINT LookaheadBufferSize,
                UINT PacketSize )
```

Parameters are the following:

Parameter	Description
ProtocolBindingContext	Pointer to a OPEN_INSTANCE structure that identifies the instance of the packet capture driver to which the packets are destined.
MacReceiveContext	Handle that identifies the underlying NIC driver that generated the request. This value must be used when the packet is transferred from the NIC driver as a parameter for the *NdisTransferData* NDIS call.
HeaderBuffer	Pointer to the buffer in the NIC driver's memory that contains the header of the packet.
HeaderBufferSize	Size in bytes of the header buffer.
LookAheadBuffer	Pointer to the buffer in the NIC driver's memory that contains the incoming packet's data. During initialization, the packet capture driver performs a OID call that tells to the NIC driver to force the dimension of this buffer to the maximum value allowed.
LookaheadBufferSize	Size in bytes of the lookahead buffer.
PacketSize	Size of the incoming packet, excluded the header.

First of all, the *ProtocolBindingContext* parameter is used to determine if current instance is running in mode 0 or in mode 1.

Then, *Packet_Tap* executes the BPF filter on the packet. The filter is obtained from the OPEN_INSTANCE structure pointed by the *ProtocolBindingContext* input pa-

rameter. To optimize the capture performances and minimize the number of bytes copied by the system, the BPF filter is applied to a packet before copying it, i.e. when it is still in the NIC driver's memory.

Notice that the capture driver receives the incoming packet from NDIS in two buffers: one containing the header and one containing the data. The reason of this subdivision is that normally a protocol driver makes separate uses of the header (used to decode the packet), and the data (sent to the applications). This is not the case of the packet capture driver, that works at link-layer level and needs to threat the whole packet as a unit. However, we noted that in the great part of the cases the packet is stored in the NIC driver's memory in a single buffer (this is the most obvious choice, because the packet arrives to the NIC driver through a single transfer). In these situations *HeaderBuffer* and *LookAheadBuffer* point to two different sections of the same memory buffer, and it is possible to use *HeaderBuffer* as a pointer to the whole packet and use the standard *bpf_filter* function. The packet driver in every case performs a check on the distance between the two buffers: if it is equal to *HeaderBufferSize* (i.e. there is a single buffer), the standard *bpf_filter* function is called, otherwise *bpf_filter_with_2_buffers* is called.

If the filter accepts the packet, there are two possibilities:

- Current instance is working in mode 1. In this case, packet_tap is extremely fast, because it has to do very few operations. The *Npackets* field of the OPEN_INSTANCE structure associated with current driver's instance is incremented. The length of the packet (plus a correction that adds the lenght of the preamble, the SFD and the FCS) is added to the *Nbytes* field of the OPEN_INSTANCE structure. After this, *packet_tap* returns.

- Current instance is working in mode 0. *Packet_Tap* tries to extract an element from the linked list of pending reads. This operation can have two results:
    - The list of pending reads is empty. This means that the user-level application is not waiting for a packet in this moment. The incoming packet must be copied to the circular buffer of the packet capture driver. The address of the circular buffer and the pointers to head and tail are obtained from the OPEN_INSTANCE structure pointed by the *ProtocolBinding-Context* input parameter. If the buffer is full (or, better, if the incoming packet does not fit in the remaining buffer space), the incoming packet is discarded. *Packet_Tap* allocates a NDIS_PACKET structure to receive the packet, and associates to this structure the correct memory position in the circular buffer. At this point, the amount of bytes specified previously by the filter is copied from the NIC driver's memory, and the bpf_hdr structure for this packet is built. Then the head and tail of the buffer are updated and *Packet_Tap* returns.
    - The list of pending reads contains at least one element. This means that at the moment the user-level application is blocked waiting for the result of a read on the packet driver. *Packet_Tap* extracts the first pending read from the list and finds the NDIS_PACKET structure associated with it. This structure is used to receive the packet from the NIC driver. At this point, the amount of bytes specified previously by the filter is copied from the NIC driver's memory, and the bpf_hdr structure for

this packet is built. Finally the application is awaked and *Packet_Tap* returns.

To build the bpf_hdr structure associated with a packet, current value of the microsecond timer must be obtained from the system. In Windows NT this is done by means of a call to the kernel function *KeQueryPerformanceCounter*, in Windows 95 with a call to the packet driver's function *QuerySystemTime*. Since *Packet_Tap* is called directly by the NIC driver, the receive timestamp is closer to the actual reception time.

PacketTransferDataComplete

This function is called by *Packet_Tap* when the packet must be passed directly to the user-level application. *PacketTransferDataComplete* releases the NDIS_PACKET structure and the buffers associated with the packet and awakes the application.

PacketRequest

This function is used to send a OID query/set request to the underlying NIC driver. *PacketRequest* allocates a INTERNAL_REQUEST structure and fills it with the data received from the application. Then The *NdisRequest* function from NDIS library is used to send the request to the NIC driver. This function is available only in Windows 95/98, because in Windows NT/2000 this operation is performed by the *PacketIoControl* function.

PacketRequestComplete

Callback function associated with the *NdisRequest* function of NDIS library. It is invoked by NDIS when the NIC driver has finished an open operation that was started previously either by the *PacketRequest* or *PacketIoControl* function of the packet capture driver.

PacketGetMacNameList

This function returns the names of all the NIC drivers to which the driver is attached. It is invoked by the *PacketIoControl* function when a IOCTL_PROTOCOL_MACNAME command has been received. The names are separated by a 'space' character. The list ends with a \0 character. This function is present only in the Windows 95 version.

GetDate

This assembler function returns the current system date as a 64 bit integer. The low word contains the current time in milliseconds. The high word contains the number of days since January 1, 1980. This function is present only in the Windows 95 version of the driver.

QuerySystemTime

This assembler functions returns the current microsecond system time as a 64 bit integer. This function is present only in the Windows 95 version of the driver, where it is used to get the timestamps of the packets.

5. PACKET.DLL: Packet Driver API

PACKET.DLL is a dynamic link library that interfaces the packet capture driver with user level applications. The DLL implements a set of functions that make the communication with the driver simpler. This avoids using system calls or IOCTLs in user programs. Moreover, it provides functions to handle network adapters, read and write packets from the network, set buffers and filters in the driver, and so on. There are two versions of PACKET.DLL: the first works in Windows 95/98, the second in Windows NT/2000. The two versions export the same programming interface, making easy to write system independent capture applications. Using PACKET.DLL API, the same capture application runs in Windows 95, 98 NT and 2000 without any modification. This feature allows to write a single version of libpcap (and, therefore, of WinDump) for the different families of Windows operating systems. In this manual we will describe how to use PACKET.DLL and the NDIS packet capture driver in an application, giving the details of the API exported by the DLL.

5.1. Packet Driver (PACKET.DLL) vs. Capture Library (libpcap)

If you are writing a capture application, we suggest you to use the packet capture library (*libpcap*) instead of the API described in this chapter. Libpcap uses the functions of PACKET.DLL as well, but provides a more powerful, immediate and easy to use programming environment. With libpcap, operations like capturing a packet, cre-

ating a capture filter or saving a dump on a file are safely implemented and immediate to use. Libpcap is able to provide all the functions needed by a standard network monitor or sniffer. Moreover, the programs written to use libpcap are easily compiled on UNIX because of the compatibility between Win32 and UNIX versions of this library.

However, the PACKET.DLL API offers some possibilities that are not given by libpcap. For example, a network analyzer could need to send packets to the network. Both the NDIS packet capture driver and PACKET.DLL (using function *PacketSendPacket*) allows to send packet; however libpcap doesn't have such a feature. Libpcap was written to be portable and to offer a system-independent capture API therefore it cannot exploit all the possibilities offered by the driver. In that case some functions of PACKET.DLL will be required.

5.2. Data structures

Data structures defined in *packet32.h* are:

- PACKET
- ADAPTER
- bpf_insn
- bpf_program
- bpf_hdr
- bpf_stat
- NetType

The first two are packet driver specific, while the others were originally defined in libpcap. This second set of structures is used to do operations like setting a filter or interpreting the data coming from the driver. The driver in fact uses the same syntax of BPF to communicate with the applications, so the structure used are the same. A further structure, PACKET_IOD_DATA, is defined in the file *ntddpack.h*.

The PACKET structure

The PACKET structure has the following fields:

- OVERLAPPED OverLapped
- PVOID Buffer
- UINT Length
- PVOID Next
- UINT ulBytesReceived
- BOOLEAN bIoComplete

OverLapped is used to handle the asynchronous calls to the driver; an example of its use can be found in the `PacketReceivePacket` function. The *OVERLAPPED* structure is defined in both the Windows 95 and Windows NT DDKs. In particular, it has a *hEvent* field that is reset (calling the ResetEvent function) when the packet must be received asynchronously. The driver sets this event to signal the end of an operation. The application is not obliged to check if the event is set because the Win32 GetOverlappedResult function does it automatically.

Buffer is a pointer to a buffer containing the packet's data, *Length* indicates the size of this buffer.

UlBytesReceived indicates the length of the buffer's portion containing valid data.

The field *bIoComplete* indicates whether the packet contains valid data after an asynchronous call. It is reset by `PacketReceivePacket` and by `PacketSend-Packet`, and is set by `PacketWaitPacket`.

The ADAPTER structure

This structure describes a network adapter. It has two fields:

- HANDLE hFile
- TCHAR SymbolicLink

hFile is a pointer to the handle of the driver: in order to communicate directly with the driver, an application must know its handle. In any case, this procedure is discouraged because PACKET.DLL offers a set of functions to do it.

SymbolicLink is a string containing the name of the network adapter currently opened.

The PACKET_OID_DATA structure

This structure is used to communicate with the network adapter through OID query and set operations. It has three fields:

- ULONG Oid
- ULONG Length
- UCHAR Data

Oid is a numeric identifier that indicates the type of query/set function to perform on the adapter through the `PacketRequest` function. Possible values are defined in the *ntddndis.h* include file. It can be used, for example, to retrieve the status of the error counters on the adapter, its MAC address, the list of the multicast groups defined on it, and so on.

The *Length* field indicates the length of the *Data* field, that contains the information passed to or received from the adapter.

The bpf_insn structure

This structure contains a single instruction for the BPF register-machine. It is used to send a filter program to the driver. It has the following fields:

- USHORT code
- UCHAR jt
- UCHAR jf
- int k

The *code* field indicates the instruction type and addressing modes.

The *jt* and *jf* fields are used by the conditional jump instructions and are the offsets from the next instruction to the true and false targets.

The *k* field is a generic field used for various purposes.

The bpf_program structure

This structure points to a BPF filter program, and is used by the `PacketSetBPF` function to set a filter in the driver. It has two fields:

- UINT bf_len
- struct bpf_insn *bf_insns;

The *bf_len* field indicates the length of the program.

bf_insns is a pointer to the first instruction of the program.

The `PacketSetBPF` function sets a new filter in the driver through an IOCTL call with the control code set to *pBIOCSETF;* a *bpf_program* structure is passed to the driver during this call.

The bpf_hdr structure

This structure defines the header used by the driver in order to deliver a packet to the application. The header is encapsulated with the bytes of the captured packet, and is used to maintain information like length and timestamp for each packet. It is the same used by BPF on UNIX. The bpf_hdr structure has the following fields:

- struct timeval bh_tstamp
- UINT bh_caplen
- UINT bh_datalen
- USHORT bh_hdrlen

bh_tstamp holds the timestamp associated with the captured packet. The timestamp has the same format used by BPF and it is stored in a *TimeVal* structure, that has two fields:

- *tv_sec*: capture date in the standard UNIX time format (number of seconds from 1/1/1970)
- *tv_usec*: microseconds of the capture

bh_caplen indicates the length of captured portion; *bh_datalen* is the original length of the packet.

bh_hdrlen is the length of the header that encapsulates the packet. This field is used only by the UNIX version of BPF. BPF is able to change the length of the header associated with a packet, inserting a variable offset for data alignment. This optimization is not yet implemented in our packet capture driver. For this reason, *bh_hdrlen* contains a fixed value, that is the length of the *bpf_hdr* structure.

The bpf_stat structure

This structure is used by the driver to return the statistics of a capture session. It has two fields:

- UINT bs_recv
- UINT bs_drop

bs_recv indicates the number of packets that the driver received from a network adapter from the beginning of the current capture. This value includes the packets lost by the filter, and should be a count of the packets transmitted by the network on which the adapter is connected to.

bs_drop indicates the number of packets that the driver lost from the beginning of the current capture. Basically, a packet is lost when the the buffer of the driver is full. In this situation the packet cannot be stored and the driver rejects it.

These variables do not account for the packet that are discarded directly by the network interface, due for example to a buffer overrun.

The NetType structure

This structure is used by the `PacketGetNetType` function to get from the driver the information on the current adapter's type. It ha two fields:

- UINT LinkType
- UINT LinkSpeed

Linktype indicates the type of the current network adapter (see PacketGetNetType for more informations).

Linkspeed indicates the speed of the network in Bits per second.

5.3. Functions

PACKET.DLL provides a set of functions that can be used to send and receive a packet, query and set the parameters of a network adapter, open and close an adapter, handle the dynamic allocation of PACKET structures, set a BPF filter, change the size of the driver's buffer and finally retrieve the statistics of the capture session. Available functions are:

- PacketGetAdapterNames
- PacketOpenAdapter
- PacketCloseAdapter
- PacketAllocatePacket
- PacketInitPacket
- PacketFreePacket

- PacketReceivePacket
- PacketWaitPacket
- PacketSendPacket
- PacketResetAdapter
- PacketSetHwFilter
- PacketRequest
- PacketSetBuff
- PacketSetBpf
- PacketGetStats
- PacketGetNetType
- PacketSetReadTimeout
- PacketSetMode
- PacketSetNumWrites

ULONG PacketGetAdapterNames(PTSTR pStr, PULONG BufferSize)

Usually, this is the first function that should be used to communicate with the driver. It returns the names of the adapters installed on the system in the user allocated buffer *pStr*. Starting from version 2.02 it returns, after the names of the adapters, a string that describes each of them.

BufferSize is the length of the buffer.

Warning: the result of this function is obtained by querying the operating system registry and performing OID calls on the packet driver, therefore the format of the result in Windows NT/2000 is different from the one in Windows 95/98. Windows 9x uses the ASCII encoding method to store a string, while Windows NTx uses (usually) UNICODE. After a call to `PacketGetAdapterNames` in Windows 95x, *pStr* contains an ASCII string with the names of the adapters separated by a single ASCII "\0", a double "\0", followed by the descriptions of the adapters separated by a single ASCII "\0" . The string is terminated by a double "\0". In Windows NTx, *pStr* contains the names of the adapters, in UNICODE format, separated by a single UNICODE "\0" (i.e. 2 ASCII "\0"), a double UNICODE "\0", followed by the descriptions of the adapters, in ASCII format, separated by a single ASCII "\0" . The string is terminated by a double ASCII "\0".

LPADAPTER PacketOpenAdapter(LPTSTR AdapterName)

This function receives a string containing the name of the adapter to open and returns the pointer to a properly initialized ADAPTER object. The names of the adapters can be obtained by calling the `PacketGetAdapterNames` function.

Note: as already said, the Windows 95 version of the capture driver works with the ASCII format, the Windows NT version with UNICODE. Therefore, *AdapterName* should be an ASCII string in Windows 95, and a UNICODE string in Windows NT. This difference is not a problem if the string pointed by *AdapterName* was obtained through the `PacketGetAdapterNames` function, because it returns the names of the adapters in the proper format. Problems can arise in Windows NT when the string is obtained from ANSI C functions like `scanf`, because they use the ASCII format. This can be a relevant problem during the porting of command-line applications from

UNIX. To avoid it, we included in the Windows NT version of `PacketOpenAdapter` a routine to convert strings from ASCII to UNICODE. `PacketOpenAdapter` in Windows NT accepts both the ASCII and the UNICODE format. If a ASCII string is received, it is converted to UNICODE by PACKET.DLL before being passed to the driver.

VOID PacketCloseAdapter(LPADAPTER lpAdapter)

This function frees the ADAPTER structure *lpAdapter*, and closes the adapter pointed by it.

LPPACKET PacketAllocatePacket(void)

Allocates a PACKET structure and returns a pointer to it. The returned structure should be properly initialized by calling the `PacketInitPacket` function.

Warning: The *Buffer* field of the PACKET structure is not set by this function. The buffer must be allocated by the programmer, and associated to the PACKET structure with a call to `PacketInitPacket`.

VOID PacketInitPacket(LPPACKET lpPacket, PVOID Buffer, UINT Length)

It initializes a PACKET structure. There are three input parameters:

- a pointer to the structure to be initialized
- a pointer to the user-allocated buffer that will contain the packet data
- the length of the buffer. This is the maximum buffer size that will be transferred from the driver to the application using a single read.

Note: The size of the buffer associated with the PACKET structure is a parameter that can sensibly influence the performances of the capture process. This buffer stores packets received from the the packet capture driver. The driver is able to return several captured packets using a single read call (see the `PacketReceivePacket` *function*). The number of packets transferable to the application in a single call is limited only by the size of the buffer associated with the PACKET structure used to perform the reading. Therefore setting a big buffer with `PacketInitPacket` can decrease the number of system calls, improving the capture speed. Notice also that, when the application performs a `PacketReceivePacket`, it is usually NOT blocked. When this function is invoked, the driver copies the data present in its kernel-level buffer to the application's buffer, but the application is awaked and receives the packets also if there is not enough data to fill its buffer. In this way, the application receives the packets immediately also if it has set a big user-level buffer, or when the data rate on the network is low.

VOID PacketFreePacket(LPPACKET lpPacket)

This function frees the PACKET structure pointed by *lpPacket*.

Warning: The *Buffer* field of the PACKET structure is not deallocated by this function and must be explicitly deallocated by the programmer.

BOOLEAN PacketReceivePacket(LPADAPTER AdapterObject, LPPACKET lpPacket,BOOLEAN Sync)

This function performs the capture of a set of packets. It has the following input parameters:

- a pointer to an ADAPTER structure identifying the network adapter from which the packets must be captured
- a pointer to a PACKET structure that will contain the packets
- a flag that indicates if the operation will be done in a synchronous or asynchronous way. If the operation is synchronous, the function blocks the program until the task is completed (i.e. a packet has been received). If the operation is asynchronous the function does not block the program and the `PacketWaitPacket` procedure must be used to verify the correct completion. **NOTE**: packet driver was created and tested to work with synchronous system call, therefore synchronous mode is the suggested one. Using asynchronous calls is at your own risk.

The number of packets received with this function cannot be known before the call and it is highly variable. It depends on the number of packets actually stored in the driver's buffer, on the size of these packets and on the size of the buffer associated with the *lpPacket* parameter. Figure 3.1 shows the format used by the driver to send packets to the application.

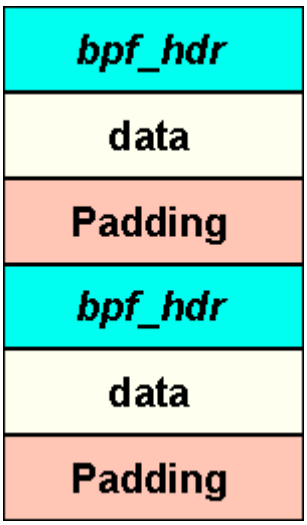

Figure 5.1: method used to encode the packets

Packets are stored in the buffer associated with the *lpPacket* PACKET structure. Each packet has a trailer consisting in a *bpf_hdr* structure that defines its length and holds its timestamp. A padding field is used to word-align the data in the buffer (to increase the speed of the copies). The *bh_datalen* and *bh_hdrlen* fields of the *bpf_hdr* structures should be used to extract the packets from the buffer. Examples can be seen either in the 'TestApp' sample application provided in the developer's pack [15], or in the *pcap_read()* function in the *pcap-win32.c* file (that can be found in the winpcap source code distribution). *Pcap* extracts correctly each incoming packet before passing it to the application, so an application that uses it will not have to do this operation.

It is possible to set a timeout on read calls with the *PacketSetReadTimeout* function. In this case the call returns even if no packets have been captured according to the timeout set by this function.

BOOLEAN PacketWaitPacket(LPADAPTER AdapterObject, LPPACKET lpPacket)

This function is used to verify the completion of an I/O operation on the packet capture driver. It is blocking if the operation has still to be completed by the driver. The return value is TRUE if the operation was successful, FALSE otherwise, and the SDK `GetLastError` function can be used in order to retrieve the error code.

BOOLEAN PacketSendPacket(LPADAPTER AdapterObject, LPPACKET pPacket, BOOLEAN Sync)

This function is used to send a raw packet to the network through the adapter specified with the *AdapterObject* parameter. 'Raw packet' means that the programmer will have to build the various headers because the packet is sent to the network 'as is'.

The user will not have to put a *bpf_hdr* header before the packet. Either the CRC needs not to be calculated and added to the packet, because it is transparently put after the end of the data portion by the network interface.

This function has the same syntax of `PacketReceivePacket`.

The behavior of this function is influenced by the *PacketSetNumWrites* function. With *PacketSetNumWrites*, it is possible to set the number of times a single write must be repeated. If this number is 1, every PacketSendPacket call will correspond to one packet sent to the network. If this number is greater than 1, for example 1000, every raw packet written by the application on the driver's device file will be sent 1000 times on the network. This feature can be used to generate high speed traffic because the overhead of the context switches is no longer present and it is particularly useful to write tools to test networks, routers, and servers. Notice that the ability to write multiple packets is present at the moment only in the Windows NT and Windows 2000 versions of the packet driver. In Windows 95/98 it is emulated at user level in PACKET.DLL. This means that an application that uses the 'repeated' write method will run in Windows 9x as well, but it speed will be very low compared to the one of WindowsNTx.

The optimized sending process is still limited to one packet at a time: for the moment it cannot be used to send a buffer of N packets.

BOOLEAN PacketResetAdapter(LPADAPTER AdapterObject)

It resets the adapter received as input parameter. It returns TRUE if the operation is performed successfully.

BOOLEAN PacketSetHwFilter(LPADAPTER AdapterObject, ULONG Filter)

This function sets a hardware filter on the incoming packets. The constants that define the filters are declared in the file *ntddndis.h*. The input parameters are the adapter on which the filter must be defined, and the identifier of the filter. The value returned is TRUE if the operation was successful. Here is a list of the most useful filters:

- NDIS_PACKET_TYPE_PROMISCUOUS: set the promiscuous mode. Every incoming packet is accepted by the adapter.
- NDIS_PACKET_TYPE_DIRECTED: only packets directed to the workstation's adapter are accepted.
- NDIS_PACKET_TYPE_BROADCAST: only the broadcast packets are accepted.
- NDIS_PACKET_TYPE_MULTICAST: only the multicast packets belonging to the groups of which this adapter is a member are accepted.
- NDIS_PACKET_TYPE_ALL_MULTICAST: every multicast packet is accepted.

BOOLEAN PacketRequest(LPADAPTER AdapterObject,BOOLEAN Set, PPACKET_OID_DATA OidData)

This function is used to perform a query/set operation on the adapter pointed by *AdapterObject*. The second parameter defines if the operation is a set (*set*=1) or a query (*set*=0). The third parameter is a pointer to a PACKET_OID_DATA structure (see the section on the data structures). The return value is true if the function is completed without errors. The constants that define the operations are declared in the file

ntddndis.h. More details on the argument can be found in the documentation provided with the DDK.

NOTE: not all the network adapters implement all the query/set functions. There is a set of mandatory OID functions that is granted to be present on all the adapters, and a set of facultative functions, not provided by all the adapters (see the DDKs to see which functions are mandatory). If you use a facultative function, please be careful to enclose it in an *if* statement to check the result.

BOOLEAN PacketSetBuff(LPADAPTER AdapterObject, int dim)

This function is used to set to a new size the driver's buffer associated with the adapter pointed by *AdapterObject*. *dim* is the new dimension in bytes. The function returns TRUE if successfully completed, FALSE if there is not enough memory to allocate the new buffer. When a new dimension is set, the data in the old buffer is discarded and the packets stored in it are lost.

Note: the dimension of the driver's buffer affects heavily the performances of the capture process. A capture application needs to make operations on each packet while the CPU is shared with other tasks. Therefore the application should not be able to work at network speed during heavy traffic or bursts, especially in presence of high CPU load due to other applications. This problem is more noticeable on slower machines. The driver, on the other hand, runs in kernel mode and is written explicitly to capture packets, so it is very fast and usually does not loose packets. Therefore, an adequate buffer in the driver is able to store the packets while the application is busy, so compensating the slowness of the application and avoiding the loss of packets during bursts or high network activity. The buffer size is set to 0 when an instance of the driver is opened: the programmer must remember to set it to a proper value.

Libpcap calls this function and sets the buffer size to 1MB at the beginning of a capture. Therefore programs written using libpcap usually do not need to cope with this problem.

BOOLEAN PacketSetBpf(LPADAPTER AdapterObject, struct bpf_program *fp)

This function associates a new BPF filter to the adapter *AdapterObject*. The filter, pointed by *fp*, is a set of instructions that the BPF register-machine of the driver will execute on each packet. Details about BPF filters can be found in chapter 4, or in [1]. This function returns TRUE if the driver is set successfully, FALSE if an error occurs or if the filter program is not accepted. The driver performs a check on every new filter in order to avoid system crashes due to bogus or buggy programs, and it rejects invalid filters.

A filter can be automatically created by using the *pcap_compile* function of libpcap. It converts a text filter with the syntax of WinDump (see the manual of WinDump for more details) into a BPF program. If you don't want to use libpcap, but you need to know the code of a filter, you can launch WinDump with the *-d* or *-dd* or *-ddd* parameters to obtain the pseudocode of the filter.

BOOLEAN PacketGetStats(LPADAPTER AdapterObject, struct bpf_stat *s)

With this function, the programmer can know the value of two internal variables of the driver:

- the number of packets that have been received by the adapter *AdapterObject*, starting at the time in which it was opened with *PacketOpenAdapter*.

- the number of packets received by the adapter but that have been dropped by the kernel. A packet is dropped when the user-level application is not ready to get it and the kernel buffer associated with the adapter is full.

The two values are copied by the driver in a *bpf_stat* structure (see section 3 of this manual) provided by the application. These values are very useful to know the state of the network and the behavior of the capture application. They are also very useful to tune the capture stack and to choose the size of the buffers. In fact:

- a high value of the *bs_recv* variable means that there is a lot of traffic on the network. If the application does not need all the packets (for example a monitor application may want to capture only the traffic generated by a particular protocol, or by a single host), it is better to set a selective BPF filter to minimize the number of packets that the application has to process. Since the filter works at kernel level, an appropriate filter increases the performances of the application and decreases the load on the system. In this way a non interesting packet does not need to be transferred from kernel to user space, avoiding the memory copy and the context switch between kernel and user mode.

- If *bs_drop* is greater than zero, the application is too slow and is loosing packets. The programmer can try, as a first solution, to set a greater kernel buffer with the `PacketSetBuff` function. A proper size of the kernel buffer often decreases substantially packets lost. Another solution is to speed up the capture process associating a bigger user buffer to the PACKET structure used in the `PacketReceivePacket` call (see the `PacketInitPacket` function). This decreases the number of system calls required, improving the capture speed.

- If the application keeps on loosing packets, it should be probably rewritten or optimized. The driver is already very fast (by the way: it is not easy to modify the driver) so it is probably better to modify the application than the driver.

BOOLEAN PacketGetNetType (LPADAPTER AdapterObject,NetType *type)

Returns the type of the adapter pointed by *AdapterObject* in a *NetType* structure. The *LinkType* of the type parameter can be set by this function to one of the following values:

- NdisMedium802_3: Ethernet (802.3)
- NdisMedium802_5: Token Ring (802.5)
- NdisMediumFddi: FDDI
- NdisMediumWan: WAN
- NdisMediumLocalTalk: LocalTalk
- NdisMediumDix: DIX
- NdisMediumAtm: ATM
- NdisMediumArcnetRaw: ARCNET (raw)
- NdisMediumArcnet878_2: ARCNET (878.2)
- NdisMediumWirelessWan: Various types of NdisWireless*Xxx* media.

The BPF capture driver at the moment supports NdisMediumWan, NdisMedium802_3, NdisMediumFddi, NdisMediumArcnet878_2 and NdisMediumAtm.

The *LinkSpeed* field indicates the speed of the network in bits per second.

The return value is TRUE if the operation is performed successfully.

BOOLEAN PacketSetReadTimeout (LPADAPTER AdapterObject , int timeout)

This function sets the value of the read timeout associated with the *AdapterObject* adapter. *timeout* indicates the new timeout in milliseconds. 0 means no timeout, i.e. the read never returns if no packet arrives.

This function works also if the adapter is working in mode 1, and can be used to set the time interval between two reports.

BOOLEAN PacketSetMode(LPADAPTER AdapterObject,int mode)

This function sets the adapter *AdapterObject* into mode *mode*. Mode can have two possible values:

- mode 0: standard capture mode, that is set by default after the *PacketOpenAdapter* call.
- mode 1: this is statistics mode, a particular working mode of the BPF capture driver that can be used to perform real time statistics on the network's traffic. The driver does not capture anything when in mode 1 and it limits itself to count the number of packets and the amount of bytes that satisfy the user-defined BPF filter. These values can be requested by the application with the standard *PacketReceivePacket* function, and are received at regular intervals, each time a timeout expires. The default value of this timeout is 1 second, but it can be set to any other value (with a 1 ms precision) with the *PacketSetReadTimeout* function. The counters are encapsulated in a `bpf_hdr` structure before being passed to the application. This allows microsecond-precise

timestamps in order to have the same time scale among the data capture in this way and the one captured using libpcap. Captures in this mode have a very low impact with the system performance.

An application that wants to use mode 1 should perform the following operations:

- open the adapter;
- set it in mode 1 with *PacketSetMode*;
- set a filter that defines the packets that will be counted with *PacketSetBpf*.
- set the correct time interval with *PacketSetReadTimeout*;
- receive the results with *PacketReceivePacket*. This function returns the number of packets and the amount of bytes satisfying the filter received in the last time interval. These values are 64 bit integers and are encapsulated in a *bpf_hdr* structure. Therefore the data returned by *PacketReceivePacket* will be 34 bytes long, and will have the following format:

struct timeval **bh_tstamp**
UINT **bh_caplen**=16
UINT **bh_datalen**=16
USHORT **bh_hdrlen**=18
LARGE_INTEGER **PacketsAccepted**
LARGE_INTEGER **BytesAccepted**

Look at the NetMeter example in the developer's pack [15] to see an example of the use of mode 1.

NOTE: if the interface is working in mode 1, there is no need of the kernel buffer because the statistic values are calculated without storing any data in it. So its dimension can be set to 0 with `PacketSetBuff`.

BOOLEAN PacketSetNumWrites(LPADAPTER AdapterObject,int nwrites)

This function sets to *nwrites* the number of times a single write on the adapter *AdapterObject* must be repeated. See PacketSendPacket for more details.

5.4. Tips: how to write high-performance capture programs

- Set an adequate driver buffer with the `PacketSetBuff` function. As already said, the size of the buffer is a very important parameter for the capture. Remember that the default buffer size if you use PACKET.DLL is 0 (if you use libpcap a 1MB buffer is automatically set), that means VERY poor capture performances. A good size for normal captures is 512KB. WinDump uses the libpcap default, i.e. 1MB buffer.

- Also the size of the buffer associated with the PACKET structure (that is the buffer in which the application receives the packets) is important. A large size means fewer system calls and higher capture speed. If you want best performances, set the size of this buffer to the same dimension of the driver's circular buffer. This ensures that the driver has never to scan the circular buffer to cal-

culate the amount of bytes to be copied, and therefore increases the speed of the copies. Libpcap (and therefore WinDump) defines a fixed 256KB buffer.

- Set the most restrictive filter on the packets needed by the application. A restrictive filter decrease the number of packets buffered by the driver and copied to the capture application. This makes space in the buffer for the needed packets only and decrease the load on the system.

- If the data of the packets is not needed (like in the most part of the capture applications), set a filter that keeps the headers only. For this reason WinDump sets a filter that tells the driver to save only the first 68 bytes of each packet (enough for the most part of the protocols). Type '*WinDump –d*' or '*WinDump –dd*' to see how this kind of filter is defined.

- If you are doing real time analysis, or you want statistics about the network, use mode 1. It uses few processor time, and it does not need any kernel buffer. Therefore, you can set the kernel buffer to 0.

6. The packet capture library (libpcap) for Windows

The Packet Capture library (libpcap) is an API that provides a high level interface to packet capture systems. It was originally written for UNIX where it is used by TcpDump and of other network tools as the low-level part of the capture process. The Win32 version of the library is fully compatible with the UNIX one, and can run indifferently in Windows 95, 98, NT, 2000 .

6.1. Differences between Windows and UNIX versions of libpcap

- libpcap for Win32 has been extended to support the 'statistics mode' of the packet capture driver. It is possible, in a very easy way, to use libpcap to obtain real-time statistics about the packets transiting on the network.
- A pair of new Windows-specific function was created.
    - *pcap_setbuff*: this function is used by the capture application to set the dimension of the buffer in the driver. It exploits the possibility of the capture driver to change the size of the driver's buffer whenever it is needed. This possibility is not present in UNIX, where the buffer is statically allocated by BPF and cannot be changed. Since the dimension of the driver's buffer is a

important parameter for the capture, a function to set or modify it without using directly the calls of PACKET.DLL is very useful.

- *pcap_setmode*: this function can be used to put an adapter in statistics mode. It was added to allow the use of statistics mode of the BPF packet capture driver from libpcap. For more informations about statistics mode, see chapter 4.

- libpcap for Win32 uses, to interact with the hardware, the interface provided by PACKET.DLL. This interface is more complex than the one offered by BPF under the UNIX systems, where the network adapter is seen as a standard file. This is not noticeable in the normal use of libpcap, but can create problems when the programmer wants to access directly to the packet driver. For example, under UNIX it is possible to perform a 'select' system call to know if any packet was captured by an adapter. Under Win32 this action is not possible.

6.2. Some words on the porting

WinDump and libpcap are written using the C language, and the original code was written to be easily portable to the various UNIX versions. Windows, though does not offer all the calls of the POSIX systems, offers some API calls very similar to them. The memory model is comparable (32 bit in Windows and in most UNIX) and the dimension of the integers is similar. The Win32 version of the pcap library is based on the packet capture driver. We developed this driver following the structure and the behavior of BPF for UNIX. This made the porting process quite easy and linear.

During the porting process some problems were encountered with the include files. In fact not all the include files needed to compile the pcap library are present in Windows. Only sometimes it is possible to find the corresponding of a UNIX structure in a Win32 include file, but some structures do not exist in Windows. This problem was solved taking the missing definitions from FreeBSD, a public domain UNIX-like operating system. Since the source code of the entire FreeBSD project is available under the Berkeley license, we were able to obtain the needed include sections from it. The include files containing these definitions are put in the *Win32-Include* directory.

libpcap also needs some UNIX network functions that are not present in Win32 (for example *getnetent*, *getnetbyname*, etc.). These functions, were taken from libraries provided with public-domain C compilers for Free-BSD. For example GNU C for Linux has a library called *libc* whose source code is available and from which we took some functions.

The code for the communication with the network adapter had to be rewritten in order to use the NDIS packet capture driver. According to the libpcap source code structure, the code to interact with the hardware is contained a set of files that are called pcap-XXX.c (there is usually an equivalent pcap-XXX.h), where XXX indicates a particular operating system or a particular capture system. Examples of these files are pcap-nit.c, pcap-bpf.c, pcap-linux.c. The functions in these files are used to:

- Return the statistics from the adapter.
- Open the adapter to capture the packets.
- Read packets from the adapter.
- Set a filter on the incoming packets.

We created a file named pcap-win32.c, that implements these functions for the Win32 operating systems family, using the functions of the PACKET.DLL API to interact with the capture driver. This file has the following functions:

Function	Description
pcap_open_live	This function is used to open an adapter and initialize it to capture packets. First, this function opens the adapter with the PacketOpenAdapter function of PACKET.DLL. If no errors occurred, and if the promisc input parameter is TRUE, the adapter is set in promiscuous mode with a call to PacketSetHwFilter. Then the type of network is determined with a call to PacketGetNetType. The type of network is saved to be used lately with libpcap's pcap_datalink function. Then pcap_open_live allocates a buffer to receive the packets from the driver. Notice that this buffer is allocated once by pcap_open_live, and then is used by all the pcap_read calls. It is allocated only a time to optimize the capture performances. The size of this buffer is 256 KB and is fixed: changing it requires the recompilation of libpcap. Notice that this buffer in libpcap for UNIX must have the same size of the BPF's packet buffer. In Windows, the size of the two buffers can be different, because the copy mechanism of the packet capture driver is independent from the size of the application's buffer. This feature is very useful when the driver's buffer is very big (allocating a corresponding application buffer's wouldn't be possible). At this point a PACKET structure is allocated and initialized. It will be used to read the data from the driver. In previous versions of libpcap for Windows, a PACKET structure was allocated and initialized for every read on the driver. To improve the capture speed, from version 2.01 there is only one global PACKET structure, that is used for all the read calls of libpcap. Finally, pcap_open_live uses the PacketSetBuff function of PACKET.DLL to allocate a 1 MB packet buffer in the capture driver. The user will be able to change the size of this buffer with the pcap_setbuff libpcap's function.
pcap_read	This function reads a group of packets from the packet capture driver and invokes the user's callback function for each of them. The function starts performing a PacketReceivePacket to fill the libpcap's packet buffer with data coming from the driver. At this point pcap_read loops through the packets, invoking for each of them the user defined callback function received as an input parameter.
pcap_setbuff	This function sets a new buffer in the packet capture driver calling the PacketSetBuff procedure of PACKET.DLL. This function is not present in the UNIX versions of libpcap.

pcap_setfilter	This function sets a new packet filter in the capture driver. The filter is received in a bpf_program structure, and is passed to the driver calling the PacketSetBpf procedure of PACKET.DLL.
pcap_stats	This function get the capture statistics from the packet capture driver (number of packets received and number of packets dropped by the driver) calling the PacketGetStats procedure of PACKET.DLL.
pcap_setmode	Sets the working mode of an adapter. This function can be used to put an instance of the driver in statistics mode, and is present only in the Win32 version of libpcap.

The use of packet.dll makes possible to have only a version of libpcap that works in Windows 95 and Windows NT. Pcap-win32.c is not very different from the equivalent file for the UNIX systems (like FREEBSD or Digital UNIX) that have BPF in the kernel. This is due to the similar behavior of the BPF and our capture driver.

As for the rest of the libpcap's source code, we isolated all our changes to the original sources (above all different include files or minor changes) through the use of *#ifdef* and *#ifndef* like in the following example

```
#ifdef WIN32
/* source code for Windows */
#endif
```

In this way, the code of libpcap for Windows is compatible with the code for UNIX, and can be compiled in the UNIX environment in order to generate the correct libpcap binary file.

6.3. libpcap and C++

The pcap library was written to be used from C programs. The reason is that the original version of libpcap was for the UNIX family of operating systems, where C is the most used language. In Windows, C++ (with the MFC class library) is the most used programming language. For this reason we made some efforts to allow a programmer to easily use the libpcap functions from C++. A new include file, called *Pcap-c++.h*, was created. It exports the functions of libpcap in the C++ calling convention, and must be included in a C++ file instead of the standard *pcap.h* file.

6.4. Manual

As we said, the Win32 of libpcap is a superset of the UNIX one and is fully compatible with it. The only differences are statistics mode and the *pcap_setbuff* function. For this reason, in Appendix 1 is provided the UNIX manual of libpcap, modified to include the new function introduced in the Win32 version.

6.5. How to use 'statistics mode'

From version 2.02, libpcap for windows offers the possibility to exploit statistics mode of the packet capture driver. Statistics mode, is a particular working mode of the BPF capture driver that can be used to perform real time statistics on the network's traffic in an easy way and with the minimum impact on the system. When in statistics mode, the driver does not capture anything, but limits itself to count the number of packets and the amount of bytes that satisfy the user-defined BPF filter. This is very useful when a user needs to perform statistics on the network's traffic, but does not

need the data of every packet. Statistics mode allows to obtain statistics values with a VERY low impact on the system compared to the standard capture mode, because minimizes the number of bytes copied and context switches. The counters are encapsulated in a `bpf_hdr` structure before being passed to the application, so interaction with libpcap is quite immediate.

An application that wants to use statistics mode should perform the following operations:

- open the adapter with **pcap_open_live**. Furthermore, with this call the programmer can specify the time interval that will occur between two samples.
- put the adapter in statistics mode calling **pcap_setmode**.
- initialize the adapter in the standard way, calling **pcap_setfilter**, **pcap_loop**, etc

At this point, the capture callback function is invoked every time the timeout defined with **pcap_open_live** expires. This function receives in the third input parameter a pointer to a buffer containing two 64 bit counters. The first holds the number of packets satisfying the filter received in the last interval, the second the amount of bytes satisfying the filter. The second parameter of the callback function is a standard *bpf_hdr* structure, with a valid timestamp. Next table shows the data received from the packet driver.

HEADER
struct timeval **bh_tstamp**
UINT **bh_caplen**=16
UINT **bh_datalen**=16
USHORT **bh_hdrlen**=18
DATA
LARGE_INTEGER **PacketsAccepted**
LARGE_INTEGER **BytesAccepted**

The NetMeter example in the developer's pack shows how to use statistics mode from libpcap.

7. Compilation issues and developer's pack

This chapter contains the steps to follow when compiling the source code of winpcap and WinDump, and the instructions to build applications that use PACKET.DLL or libpcap. Moreover, it is provided a brief explanation of the content of the developer's pack, created to help the work of the programmers, and of the examples included in it.

7.1. Compiling the Packet Driver

The Packet Driver is platform dependent, therefore it MUST be compiled on the platform where you intend to use it.

7.1.1. Compiling on Windows NT/2000

Software requirements:

- Microsoft Windows NT 4.0 operating system
- Microsoft Driver Developer Kit (DDK) for NT 4.0
- Microsoft Software Development Kit (SDK)
- Microsoft Visual C++ 6.0

If your system satisfies the requirements, follow these steps:

1. From the Window NT Start menu, select the folder *Programs* and then *Windows NT DDK*. From here select the voice *Checked Build Environment* if you want to build a debug version, or *Free Build Environment* if you want to build a release version.

2. A DOS shell will be opened. Move to the directory whit the driver's source code and type the command

   compile

   This command will generate the driver (packet.sys). The debug version of the files will be generated in the directory \i386\checked. The release version in \i386\free.

3. To compile the sources of PACKET.DLL, load the project contained in the directory packetNT\dll\project in the Visual C++ 6.0 IDE. Build the project to obtain the PACKET.DLL and *packet.lib* files. The debug version of these files will be generated in the directory packetNT\dll\project\debug, the release version in packetNT\dll\project\release.

Warning: usually, during the first compilation of the driver, a lot of 'last line incomplete' errors are generated. Ignore these errors and let the compilation process continue. These errors usually disappear from the second compilation.

7.1.2. Compiling on Windows 95/98

To compile the driver there are the following software requirements:

- Windows 95/98 operating system
- Driver Developer Kit (DDK) for Windows 95
- Software Development Kit (SDK)
- Visual C++ 6.0

If your system satisfies the requirements, follow these steps:

1. Open a dos shell
2. Go in the VisualC++ BIN directory (for example C:\DEVSTUDIO\VC\BIN) and execute the command

Vcvars32

3. Go in the SDK directory (for example C:\MSSDK) and execute the command

   Setenv sdk_path

   where *sdk_path* is the directory of SDK (for example Setenv C:\MSSDK)

4. Go in the DDK directory (for example C:\DDK) and execute the command

   Ddkenv 32 net

5. Move to the directory whit the driver's source code and type the command

nmake rtl

to obtain a release version, or

nmake

to obtain a debug version.

The release version of *packet.vxd* will be placed in the *retail* directory, the debug version in the *debug* directory.

Warning: we often encountered problems compiling the driver for Windows 95. In particular on some systems the NMAKE utility was not able to launch ADRC2VXD (the driver is generated without the copyright informations), and on other systems the debug version of the driver was not compiled correctly. We don't know the cause of these problems.

7.2. Compiling libpcap

System Requirement:

- Microsoft Visual C++ 6.0 compiler.

Project files are in the directory *libpcap\win32-prj* of the WinDump source code distribution. Load the project from the Visual C++ 6.0 IDE and build the program. The output file *libpcap\lib\libpcap.lib* will be generated.

The project can be compiled indifferently in Windows 95, 98 or NT. The output library file is system-independent.

7.3. Compiling WinDump

Software requirement:

- Microsoft Visual C++ 6.0 compiler.

Project files are in the directory *windump\win32-prj* of the WinDump source code distribution. Load the project from the Visual C++ 6.0 IDE and build the program. The release version of the *WinDump.exe* executable file will be generated in the directory *windump\win32-prj\release.* The debug version of the executable file will be generated in the directory *windump\win32-prj\release.*

The project can be compiled indifferently in Windows 95, 98 or NT. The executable file generated is system-independent.

7.4. How to compile an application that uses directly PACKET.DLL

The creation of an application that uses the capture driver through PACKET.DLL is quite trivial. The following things must be done in order to compile the application:

- Include the file *packet32.h* at the beginning of every source file that uses the functions exported by the DLL. *Packet32.h* is distributed both with the PACKET.DLL source code and the developer's pack and it is platform-independent.
- Set the options of the linker to include the *packet.lib* file. *Packet.lib* is generated compiling the packet driver and can be found in the developer's pack.

The application, doing so, will be able to use the functions exported by the DLL and to use the driver to capture packets.

7.5. How to compile a C application that uses libpcap

The following things must be done to compile an application that uses libpcap:

- Include the file *pcap.h* at the beginning of every source file that uses the functions exported by library.

- Set the options of the linker to include the *libpcap.lib* library file. *libpcap.lib* is generated compiling the libpcap source code and can be found in the developer's pack.

- Set the options of the linker to include the *wsock32.lib* library file. This file is distributed with the C compiler and contains the socket functions for Windows. It is needed by some libpcap functions.

The application, doing so, will be able to use the functions exported by libpcap and use the NDIS packet capture driver to capture packets.

Remember that:

- To add a new library to the project with Microsoft Visual C++ 6.0, you must select *Settings* from the *Project* menu, then select *Link* from the tab control, and then add the name of the new library in the *Objcet/library modules* editbox.

- To add a new path where Microsoft Visual C++ 6.0 will look for the libraries, you must select *Options* from the *Tools* menu, then *Link* from the tab control,

library files from the *show directories for* combobox, and the add the path in the *directories* box.

- To add a new path where Microsoft Visual C++ 6.0 will look for the include files, you must select *Options* from the *Tools* menu, then *Link* from the tab control, *include files* from the *show directories for* combobox, and the add the path in the *directories* box.

Note: when compiling a program using libpcap it is not necessary to include the *packet32.h* and *packet.lib* files to interact with PACKET.DLL, because *libpcap.lib* includes also the code present in *packet.lib*. The pcap library, in fact, uses the PACKET.DLL API, but hides this to the programmer giving a higher level of abstraction and a more powerful interface.

7.5.1. How to port a UNIX application that uses libpcap to Windows

Assuming that you are able to compile the application in Windows (this operation can be very difficult and is cannot be explained here), the only thing you have to do is to link it with libpcap for Windows, following the steps of the previous Section.

7.6. How to compile a C++ application that uses libpcap

The following things must be done to compile an application that uses libpcap:

- Include the file *pcap-c++.h* at the beginning of every source file that uses the functions exported by the library. This file can be found in the *libpcap\Win32-Include* directory of the WinDump source code distribution, or in the developer's pack.

- Set the options of the linker to include the *libpcap.lib* library file. *libpcap.lib* is generated compiling the libpcap source code.

- Set the options of the linker to include the *wsock32.lib* library file. This file is distributed with the compiler and contains the socket functions for Windows. It is needed by some libpcap functions.

- If the linker generates errors like "conflicts with use of other libs", or "Symbol XXX already defined in XXX", force the linker to ignore the multiple definitions. To do this with Microsoft Visual C++ 6.0, select *Settings* from the *Project* menu, select *Link* from the tab control, and type "/force:MULTIPLE" in the *Project Options* edit box. If the link process generates warnings, ignore them.

The application, doing so, will be able to use the functions exported by libpcap and to use the NDIS packet capture driver to capture packets. Note that it is not necessary to include the *packet32.h* and *packet.lib* files to interact with PACKET.DLL, because the code present in *packet.lib* is already present in *libpcap.lib*.

7.7. Developer's pack

A developer's pack was created to help the work of the programmers that want to use winpcap to develop their network tools. It contains all the necessary to build an application that makes use of PACKET.DLL or libpcap, i.e. the include files and the binaries of the libraries. Moreover, it provides a set of examples to show the most common uses of the architecture.

Here I give a short description of these examples. All the examples, once compiled, can run both on Windows 95 and on Windows NT.

7.7.1. TestApp

This is a very simple capture program that shows the use of the *packet capture driver* through the PACKET.DLL API. It is a console application that once complied can be executed under Windows 95, 98 NT and 2000. Testapp.exe, when executed, gives to the user the possibility to choose one of the adapters installed on the machine. Then It captures the packets from the network adapter specified until a key is pressed, dumping the content of each packet on the screen.

7.7.2. PktDump

This example shows how to write and compile an application that uses the packet capture library under the Win32 environment. This example reads the packets from a file or a network adapter, printing on the screen the timestamp, the length and the data of the packets. It was originally written for UNIX (the UNIX makefile is provided), and was compiled in Windows without being modified. It can run, once compiled, in Windows 95, 98 NT and 2000. Notice that this program is very similar to the previous in the behavior and the output is not too different, but the code of the version that uses libpcap is noticeably shorter and simpler, because libpcap offers an higher level programming interface.

7.7.3. Pcap_Filter

This is another example of the use of libpcap. It is more complex than PktDump, and shows, among other things, how to create and set filters and how to save a capture to disk. It can be compiled under Win32 or under UNIX (makefile is provided). Pcap_filter (pf.exe) is a general-purpose packet filter: it receives as input parameters a source of packets (it can be an interface or a file), a filter and an output file. It captures packets from the source until CTRL+C is pressed, applies the filter to every packet, and saves it in the output file if it satisfies the filter. Pcap_filter can be used to capture packets from network according to a particular filter, but also to extract a set of packets from a previously saved file. The format of input and output files is the same of WinDump and TCPdump.

7.7.4. NetMeter

This program shows an alternative and light-weight use of libpcap and the BPF capture driver. NetMeter draws in a window the scrolling diagram of the network's load in bytes per second and in packets per second. The application is written in C++ with MFC, and uses libpcap to interact with the network. In particular, mode 1 of the packet driver is used, so the application is quite simple and very fast. It can run in Windows 95, 98 NT and 2000.

7.7.5. Traffic Generator

This example shows how to use the packet capture driver through PACKET.DLL to send packets to the network. It takes as input parameters the interface that will be used, the number of packets to send and their size. The generated packets will have 1:1:1:1:1:1 as source MAC address, and 2:2:2:2:2:2 as destination address. The 'multiple write' feature of the driver is used to obtain a higher transmit rate, therefore the write performance is better if traffic generator is used in Windows NT or Windows 2000.

8. WinDump

WinDump is the porting to the Windows platform of tcpdump. WinDump is fully compatible with tcpdump but introduces some extensions to work better in the Win32 environments. The WinDump.exe executable file is linked with libpcap for Win32, therefore can run both under Windows 95/98 and under Windows NT/2000. To run WinDump, the correct version of the *BPF packet capture driver* and of the PACKET.DLL library must be installed in the system.

8.1. Manual

Since the differences from WinDump and tcpdump are very few, in Appendix 2 is provided the manual of tcpdump, modified to include our additions.

8.2. Differences between WinDump and tcpdump

Our WinDump project tries to make a clean and complete porting of tcpdump, therefore the use of the two programs is nearly identical. All functions offered by tcpdump are implemented in WinDump, so every operation that can be done by tcpdump can be done in Windows as well, using WinDump. In addition, WinDump offers some characteristics that are not present in tcpdump:

- It is possible to obtain, with the '–D' switch, the list of the network interfaces present in the system and a short description of them. This option was added because the names that Windows gives to the interfaces (i.e. the names of the files associated with the device drivers that manage the adapters) are different from the names used in UNIX. These names are not very easy to obtain in Windows, so a flag that lists them is very handy, particularly if the machine has more than one network adapter. Since in Windows NT and Windows 2000 the name of an adapter can be quite long to type, it is possible also indicate it with its number.

- It is possible to change directly from WinDump the dimension of the circular buffer used by the NDIS packet capture driver to store the packets coming from an interface. This operation can be done through the '-B' switch. The variable-dimension driver's buffer is a feature of NDIS packet capture driver that is not present in BPF, and therefore tcpdump for UNIX does not have this switch. As we say in the documentation of the driver and of the PACKET.DLL API, the size of the driver's buffer influences deeply the capture performances and is a very important parameter of the capture process. A big buffer can allow good captures also on slow machines or on fast networks.

8.3. Some words on the porting

The problems encountered during the porting of WinDump are more or less the same that we had during the porting of the pcap library (see the section 6). We had to import some include files from FreeBSD, and we put them in the *Win32-Include* directory. Moreover, we wrote some Windows specific code to handle things like

Winsock and the Windows NTx UNICODE format. This code is in the file *Win32-Src\w32_fzs.c*. Finally we had to modify t*cpdump.c*, the file containing the *main()* function, to add the new switches of the command line.

However, the porting of tcpdump was easier than the porting of the pcap library. In fact tcpdump, using the functions exported by libpcap, does not interact directly with the system and with the network adapter. This makes it quite easy to port.

We isolated all our changes to the original sources through the use of *#ifdef* and *#ifndef* like in the following example

```
#ifdef WIN32
/* source code for Windows */
#endif
```

Therefore the code of WinDump is compatible with the code of tcpdump and can be compiled under UNIX generating the normal tcpdump executable.

9. Performance and tests

Main goal of a packet capture driver is performance. This means low use of system resources (memory and processor) but also low probability of loosing packets.

The following main parameters influence the performances of the capture process: the efficiency of the filter, size of the packet buffer, the number of bytes copied and the number of system call that needs to be executed by the application.

1. The efficiency of the packet filter is a very important parameter, because the filter must be applied to every incoming packet (i.e. thousands of times per second). The packet capture driver uses the fast and highly optimized BPF filter (for more details about the performances of BPF filter, see [1]), whose virtual-processor architecture is suited for modern computers architectures.

2. More optimized packet filters have been developed after the original BPF. The more interesting for this kind of applications are MPF [13], and BPF+ [12]. The packet capture driver does not offer at the moment the advenced features of these two filters. It could be very useful to include in the driver the possibility to efficiently handle similar filters in a way similar to MPF.

3. Kernel buffer's size is the parameter that influences the number of packet loss during a capture; a bigger buffer means lower loss probability. Since the correct size of the buffer is a very subjective parameter and depends on various factors like network speed or machine characteristics, the packet capture driver offers a dynamic buffer that can be set to any size whenever the user wants to

do that. In this way it is possible to set very big buffers on machines with an huge amount of RAM. Notice however that the buffer is freed when the driver's instance is closed, therefore the memory is used by the driver only during the capture process (i.e. when really needed).

4. Performances are strongly influenced by the number of bytes that need to be copied by the system. This task can absorb a lot of processor time and buffer memory. To overcome the problem, the packet capture driver applies the filter to an incoming packet as soon as it arrives to the system: the packet is filtered when it is still in the NIC driver's memory, without copying it. This means that no copy is needed to filter the packet. The filter tells how many bytes of the packets are needed by the user-level application (for example WinDump needs only the first 68 bytes of each packet). The packet capture driver copies only this amount of bytes (instead of the whole packet) to the circular buffer. This is very important also because reduces the space occupied by the packet in the circular buffer that is used more efficiently. The selected packet is then copied to the user-level application during a read system call. Summarizing, there are two copies of the cut packet, none of the entire packet that is equivalent of the number of copies done by the UNIX version.

5. Each read system call implies a context switch from user-mode (ring 3) to kernel-mode (ring 0) plus another another to return to user-mode. This process is notoriously slow and can influence the capture performances. Since a user-level application might want to look at every packet on the network and the time between packets can be only a few microseconds, it is not possible to do a read system call for each packet. The packet capture driver collects the data from several packets and copies it to the application's buffers in a single read

call. The number of packets copied is not fixed and depends on the dimension of the application's buffer that will receive the packets: the driver detects the size of this buffer, and copies packets to it until it's full. Therefore, it is possible to decrease the number of system calls increasing the size of the application's read buffer.

9.1. Tests

This Section aims at giving some indications about the performance of the capture process on various operating systems. Results obtained under the various Windows platforms have been compared with the ones provided by BPF/libpcap/TCPdump in FreeBSD 3.3 in order to determine the goodness of our implementation.

9.1.1. Testbed

The testbed (shown in next figure) involves two PCs directly connected by means of a Fast Ethernet link. This assures the isolation of the testbed from external sources (our LAN), allowing accurate tests.

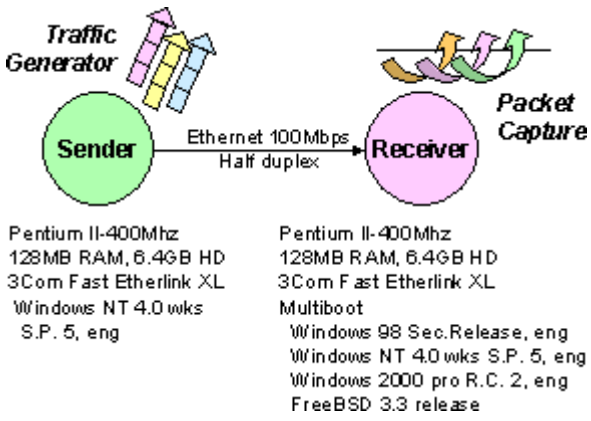

Figure 9.1: testbed

A Windows NT workstation using the 'TG' tool (available into the developer's pack) based on the packet capture device driver generates traffic. This program is able to send data to the network using almost directly NDIS primitives, avoiding the overhead of the upper protocols and assuring the highest transfer rate compared to other traffic generator tools.

Packet sizes have been selected in such way to generate the maximum amount of packets per second, that is usually the worst operating situation for a network analyzer. Packet sizes that maximized the number of packet sent was 101 bytes, as shown in next figure.

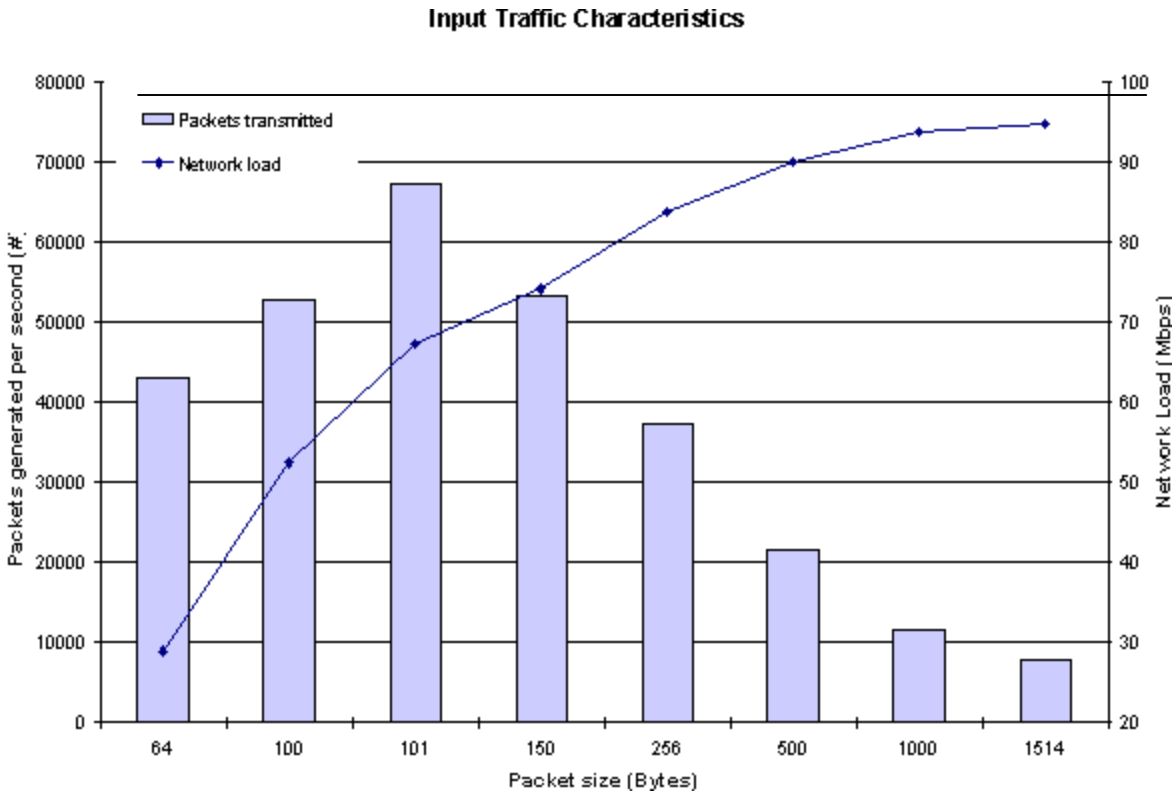

Figure 9.2: traffic generation capability

The generated traffic is usually able to fill all the available bandwidth and there is no other traffic on that link. Tests are repeated several times in order to get accurate results and it has been derived their average value.

Operating Systems under tests are installed in different disk partitions on the same workstation in order to avoid differences due to the hardware. Traffic is sent to a non-existent host in order not to have any interaction between the workstations. The second PC sets the interface in promiscuous mode and captures the traffic using WinDump / TCPdump in various configurations. Depending on the test type, packets captured are either saved to disk, printed on screen or dropped.

The `top` program in FreeBSD, the task manager in Windows NT4/2000 and `cpumeter` [11] in Windows 98 are the programs used to measure the CPU load. First two tools are shipped with the operating system, while the third one is available

on the Internet. WinDump tested was version 2.02; TCPdump was the one included in the FreeBSD 3.3 distribution (TCPdump version 3.4, libpcap version 0.4).

Even if our tests manage to isolate the impact of each subsystem (BPF and filtering, BPF and copying overhead), results are not able to compare exactly the performances of each component. This is due to the different architecture of the various versions, and to the impossibility to isolate each component from interacting one to the others and to the Operating System. In our opinion, the most representative test is test number 3 that measures performances "as a whole", including the packet driver, libpcap, WinDump as well as the operating system overhead (kernel processing, data transfer to disk, etc). The reason is that the "whole system" performance is what the end user is most interested in.

9.1.2. Results

Test 1: Filter performances

This test aims at measuring the impact of the BPF filter machine on the capture process. Packets are received by the network tap and checked by the BPF filter. The filter receives and processes all the packets sent. WinDump/TCPdump is launched with the following command line:

```
windump 'filter'
```

Where 'filter' is a packet filter with the TCPdump syntax. This test was executed with two different filters:

- 'udp': accepts only UDP packets. It is made by 5 instructions.

- 'src host 1.1.1.1 and dst host 2.2.2.2': accepts only packets coming from 1.1.1.1 and going to 2.2.2.2. This filter is a bit more complex, and is made by 13 instructions.

Since no packet satisfying these filters passes on the network (because all the packets are generated by the TG tool), the filters reject all the packets. In this way, there is no copy and no interaction with the application. Only the filter function uses system resources.

The filtering function does not use memory, so what is interesting to see here is the processor usage, shown in next figure.

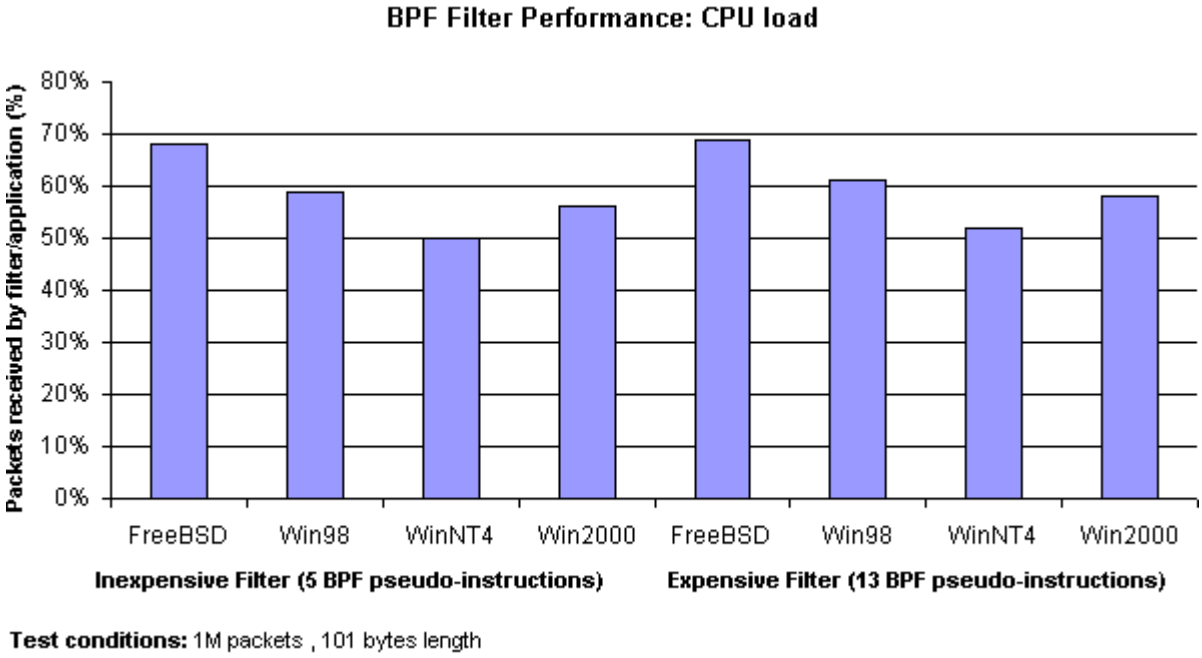

Figure 9.3: results of test 1

The figure shows that differences between different OSs are very limited. This is what we expect and confirms that our choice to create the BPF as a protocol was good enough to compete with original BPF. CPU load varies among different platforms,

remaining however at acceptable levels. Windows platforms have sensibly better results. This is due probably to the fact that NDIS usually invokes the *packet_tap* function before the DMA transfer of the packet is finished, giving a bit of time to BPF. *bpf_tap*, instead, is called after the end of the DMA transfer.

Notice finally that the values are very similar for the two filters. This confirms that the BPF filtering machine is well optimized, and that its efficiency increases with longer filters.

Test 2: Driver's performance

For this second test, a "fake" capture application based on libpcap was created and compiled for Windows and FreeBSD. This application receives ALL the packets from the driver (setting an accept-all filter), but discards them without any further processing, because the libpcap 'callback' function is empty. All the packets are processed by the underlying network levels, then by the packet driver, but there is NO packet processing at user level. The portion of packet to be accepted can be decided by user. This test aims at a global evaluation of efficiency of packet driver, including the copy processes from the interface driver to the kernel buffer and then to the user buffer. There was no filter in these tests, so the filtering function does not influence the results. Next figure shows CPU usage for various combinations "packet length-portion copied".

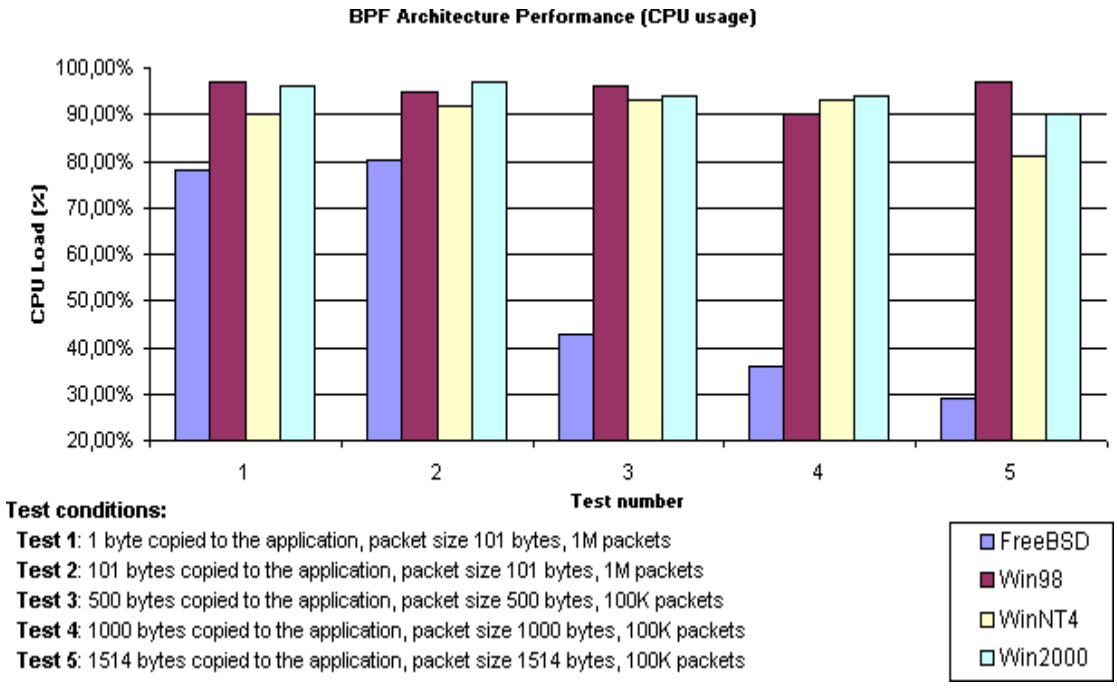

Figure 9.4: results of test 2 (CPU usage)

FreeBSD has better performance than Windows, mainly because the tap function, that in FreeBSD is very simple and fast, in Windows is more complex and slow. For longer packets (i.e. for lower frequencies) the CPU use under FreeBSD decreases, but this does not happen in windows. This results stems from the "delayed write" ability of the UNIX BPF, as explained in Section 1.2.1. For high packets frequencies, the CPU load of the different systems is quite similar. However the system calls frequency (and therefore the CPU load) under UNIX decreases considerably when the size of incoming packets increases (i.e. the frequency is lower), while in Windows it remains stable.

This behavior is not a problem for Windows implementations because it uses more CPU time *only* when it is available. Figure 7, in fact, shows that all systems loose very few packets.

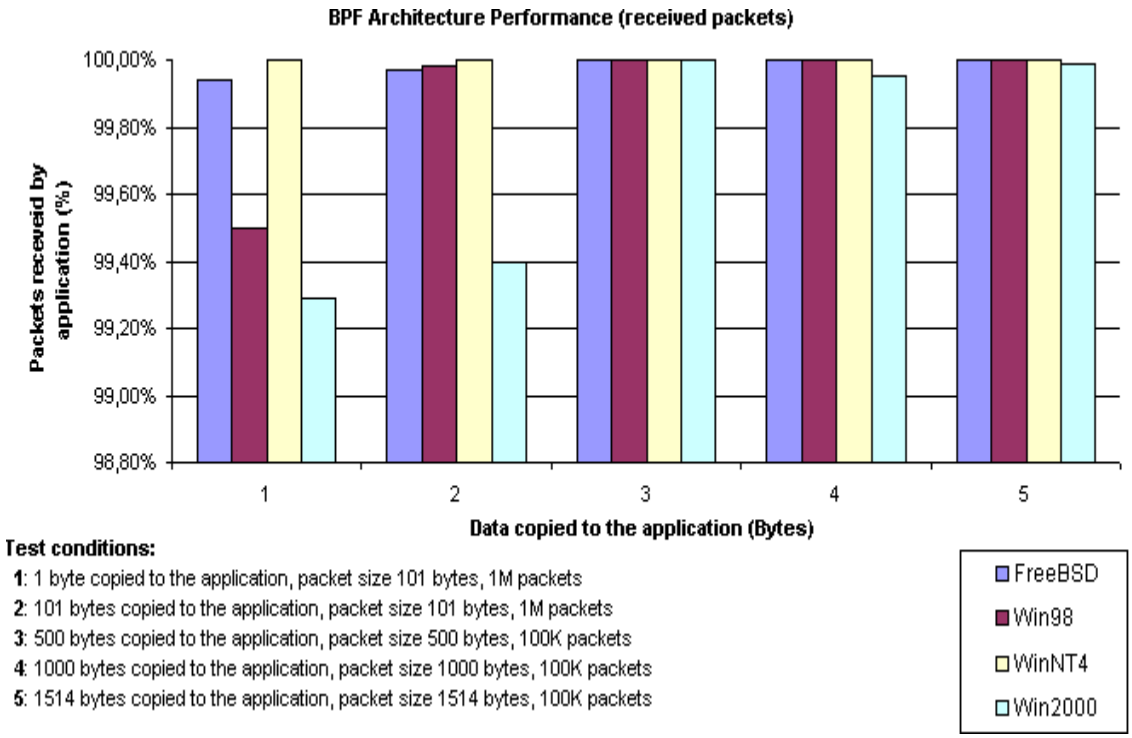

Figure 9.5: results of test 2 (received packets)

Test 3: WinDump capture performance

In our opinion, this is the most important test, because involves the use of WinDump in order to measure the entire capture process. No filter is defined. Packets are captured and stored on disk using the "-w" WinDump option.

Next figure shows the results when, for each packet, a "snapshot" of 68 bytes is saved on file, i.e. when the "windump -w test.acp" is executed.

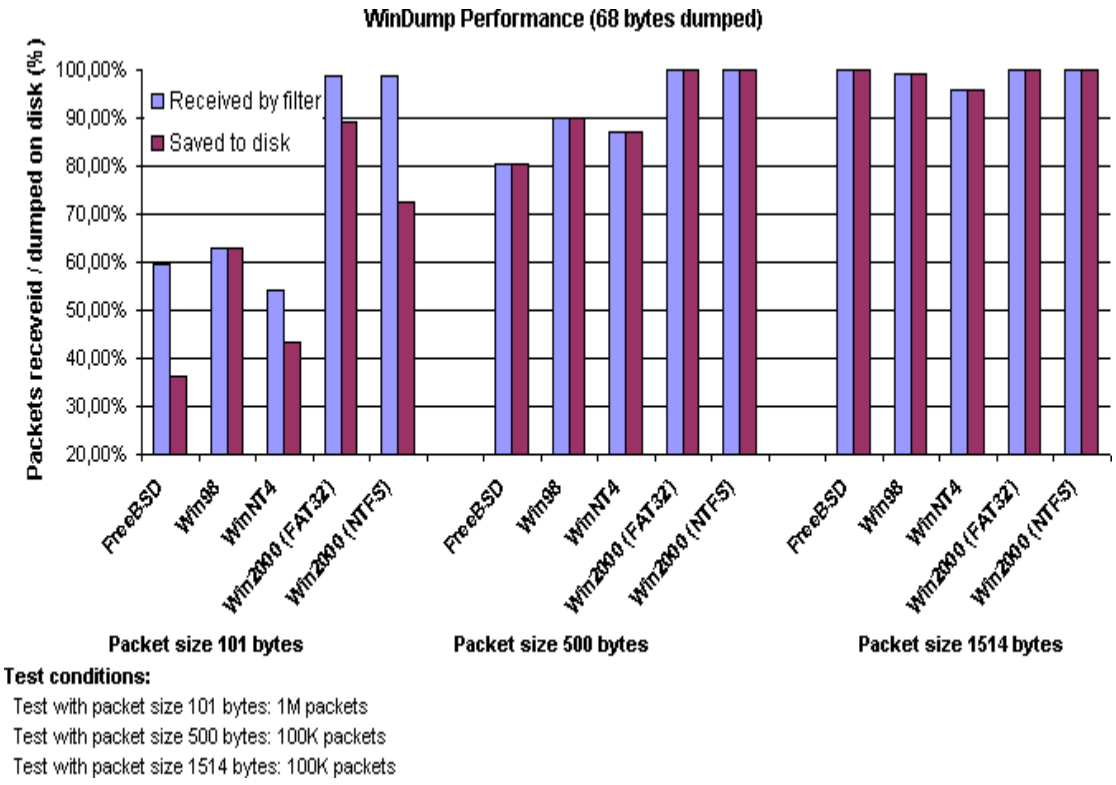

Figure 9.6: results of test 3 (cut packets)

Results are very interesting when the network is overloaded by an high number of packets per second (i.e. packet size 101 bytes, that means about 67000 packets per second). All systems suffer noticeable losses: a certain amount of packets is lost for the lack of CPU time (a new packet arrives while the tap was processing a previous one), while others are dropped because the kernel buffer has no more space to hold them. It can be noted that Windows versions work noticeably better than the FreeBSD one. This is due mainly to the better buffering method of windows versions. Windows NT 4 is able to 'detect' less packets than FreeBSD (i.e. the number of packet received by filter is lower), but saves to disk 20% more packets. Windows 98 has a very good behavior compared to FreeBSD, but the real surprise is the Windows 2000 that is able to save to disk 73% of the packets on a NTFS partition, and 89% on a FAT partition. Since the packet driver for Windows 2000 is very similar to the one for Windows NT 4, the differences are due mainly to the improvements of NDIS and of file

systems brought to Windows 2000. The heaviness of the file system is in fact a very important parameter in a test like this: notice that the same machine can capture under Windows 2000 a larger amount of packets if used with a faster file system like FAT32. This is one of the reasons because Windows 98 is faster than Windows NT 4.

When the dimension of the packets grows (i.e. packet size 500 and 1514 bytes), the situation becomes less critical because the frequency of the packets decreases, but the portion saved is always 68 bytes. The values obtained tend to become more similar, and also the slower systems have good results.

Next figure shows the results when the whole packets are saved to disk, i.e. when the "windump -s 1514 -w test.acp" is executed.

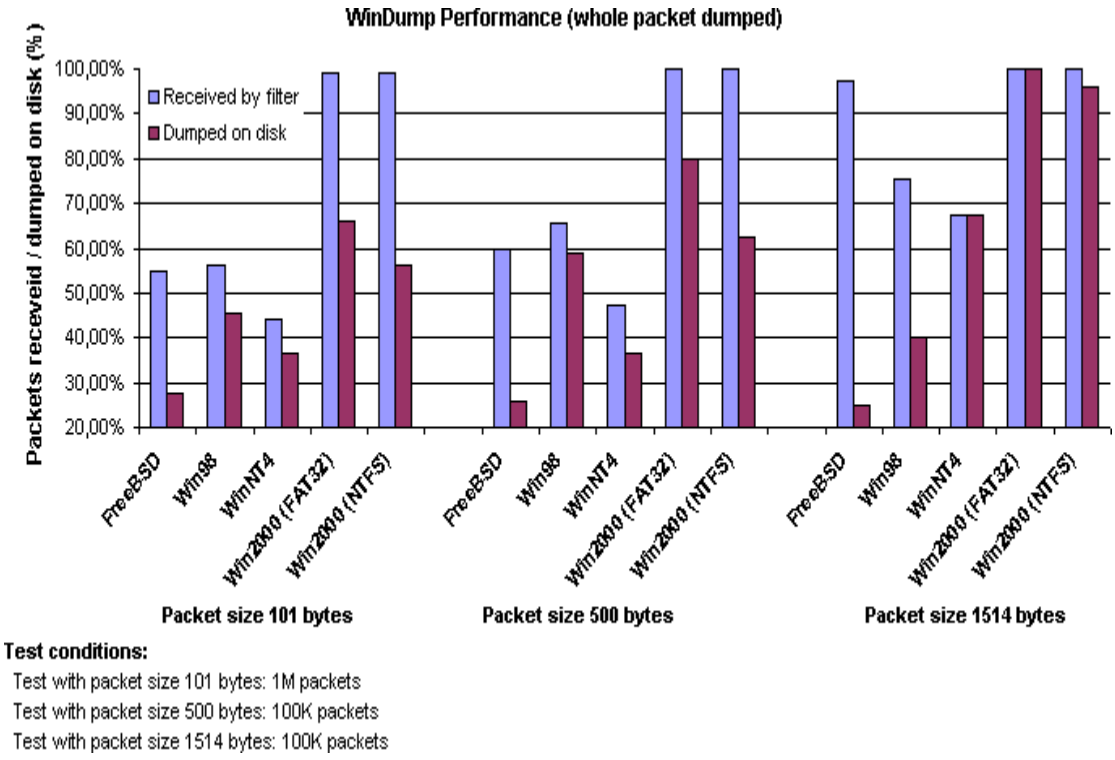

Figure 9.7: results of test 3 (whole packets)

This is quite a hard test, and the kernel buffer is very important. In fact every packet must be entirely stored, and tends to fill the kernel buffer, especially with big

packet sizes. FreeBSD is the system with the most serious problems because of its smaller buffer. Windows 2000 remains the system with better results, above all with long packets where it's the only able to capture without loosing anything.

9.1.3. Discussion

First and second tests show that the Windows implementation of BPF architecture has approximately the same impact on the system than the FreeBSD one and performances are quite similar; differences are located in the CPU load, where FreeBSD is the clear winner. This is due to the fewer code needed to implement the architecture in FreeBSD (because of the possibility to modify the OS sources) compared to the Windows one and to the “delayed write” capability of TCPdump. The results obtained are very important because show that our main goal, i.e. to create a free capture architecture with performance comparable with BPF for UNIX, has been reached. Moreover, the results show that the choice to implement the packet driver at protocol level is good enough to obtain performance comparable with the one of BPF in UNIX.

However the interesting test from the end-user standpoint is the third one, because it shows the behavior of the BPF capture driver in conjunction with the most important tool based on it: WinDump. WinDump for Windows2000 is the clear winner and TCPdump for FreeBSD is the clear looser. While the BPF architecture performs on Windows 2000 like on other systems, Windows 2000 shows the best performances because of its optimized storage management. Packets are quickly saved on file, therefore buffers are freed and the incoming packets can be received with a small number of drops. FreeBSD is the clear looser because of its different buffering architecture that is not able to sustain heavy data rates.

Notice that WinDump has been launched with the standard kernel buffer (1MB); in presence of heavy traffic the size of this buffer can be increased with a simple command line switch, improving further the overall performance of the system. Our conclusions are that BPF architecture for Windows performs well, that the dynamic buffer improves effectively the overall performances and that, among all the Windows flavors, Windows 2000 is the best platform for an high performance network analyzer.

10. Conclusions and future work

First release (1.0) of the network drivers has been released in March 99. This was a preliminary version of this work and included the network driver and WinDump. However the driver was very simple, BPF was not implemented, libpcap was partially implemented and moreover there were several performance problems due to the lack of kernel buffering.

Second release (2.0) was made available in August and it was the first complete package with full implementation of BPF, Libpcap and WinDump.

Release 2.01 (October) fixed some bugs and optimized the capture process. In that occasion the WinDump web site was greatly improved with enhanced documentation and examples.

The present version of winpcap and WinDump is 2.02. This version adds support for Windows 2000, the possibility to open multiple instances on Windows 95/98 (on Windows NT this feature was already supported), statistics mode, optimized write and other general improvements.

From the beginning of the, work a lot of work was done to improve stability and performance, so current version is mature to be used as a real-word development support, and not only as an academic and didactic project.

This fact is confirmed by the interest that the architecture aroused: the number of people that connected to the main web site of the project (http://netgroup-

serv.polito.it/windump) between may 1999 and march 2000 is over 80.000. The number of accesses per day in the last months is over 500. The amount of installed copies of WinDump is quite hard to determine, because the binaries of the driver and the executable of the program are distributed by all the most important security sites on the Internet, and by the mirrors of our web site. We estimate that several tens of thousands of copies are currently installed and used in the world.

Another important fact is that a lot of new network tools for the Windows platforms were developed using winpcap. Some of them are ports of UNIX programs written over libpcap, but the most part was developed for Windows. Among them there are also commercial applications.

The success obtained by this work shows the importance that nowadays network analysis has in the design and maintenance of a network. Moreover, capture capabilities are fundamental in the fields of security and intrusion detection, whose importance, in a world where networking technologies are used everywhere, is growing everyday.

It has to be noted that winpcap is the first complete architecture for network capture under Windows, and since it is completely free, the network developers have accepted it very quickly.

A network analysis program that uses the winpcap API is Analyzer, a public domain tool developed by the NetGroup of the Politecnico di Torino, and distributed freely on the web at http://netgroup-serv.polito.it/analyzer. The whole Analyzer's capture engine was reengineered during this thesis work, in order to use all the features provided by winpcap. This work helped a lot to debug and test the capture architecture.

There are several options concerning future developments because packet capture and network analysis are still an open research area. Moreover there is still a large amount of work deserving to be carried on the architecture described in this book.

Starting from WinDump, it is based on the code of tcpdump 3.4. A new version of tcpdump, 3.5, is going to be released in the next months. This new version will add features like IPv6, SNMPv2/v3, SMB, telnet options decoding and ASCII printing. We received requests from the tcpdump organization (http://www.tcpdump.org) to join the efforts in order to obtain a single version of tcpdump 3.5 able to coherently compile and run both under UNIX and Windows. Since WinDump is not the main target of this research work, after the attainment of such a merged version, the work on the source code of this analysis program will be carried on mainly by tcpdump.org.

The net group of the Politecnico di Torino will probably carry on the evolution of winpcap, because the work on this architecture is still in progress and new features are under development. The most important are:

- A new and simpler interaction with the operating system. This will allow to load the packet driver dynamically when it is needed instead of installing it from the control panel, making its use transparent to the final user and avoiding the need of configurations and reboots.
- The implementation of security mechanisms on Windows NT 4.0 and 2000. This is the last important missing feature compared to the original BPF, because everyone can create its own sniffer capture tools even without belonging to the Administrator group. The intent is to implement a mechanism where the network administrator can set the permission for using the network driver, allowing its use to selected groups only.
- The development of a version of winpcap for Windows CE, with the same interface of the other versions. This will extend the portability of the net-

work tools written for winpcap, and will make it easy to create new tools for CE. Network analysis architecture for palm computers will give the interesting possibility to build pocket network analysis tools.

- The improvement of the filtering engine. The field of packet filtering has recently seen a lot of new works. In particular, BPF+[12] improves the pseudo-code compiler and the optimizer, and makes use of a JIT compiler that coverts pseudo-code into machine code. This speeds the filtering process but only in the case of a single filter. Other architectures have been developed to efficiently handle many filters that are active at the same time, but none of them seems to be suitable to be used in winpcap. The most interesting of them for our purposes seems to be MPF[13]. Its optimization ability is limited to quite simple cases, but it should be possible to extend it in order to make it more general. A new packet filter with the most important features of BPF+ and an extension of the capabilities of MPF should provide high performance and should allow several captures at the same time.
- The extension of winpcap in order to obtain remote capture and analysis. Since one of the goals of libpcap is hardware independence, it is possible to extend its abstraction level in order to capture packets through a remote hardware. This requires the creation of a capture server and an extension of libpcap able to transparently communicate with it. A very interesting feature of winpcap to be used remotely is statistics mode, because it requires the transfer of very low amounts of data on the network, so it can be used in real time without influencing the behavior of the network.

Bibliography

[1] S. McCanne and V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture. Proceedings of the 1993 Winter USENIX Technical Conference (San Diego, CA, Jan. 1993), USENIX.

[2] V. Jacobson, C. Leres and S. McCanne, TCPdump, Lawrence Berkeley Laboratory, Berkeley, CA, June 1989. Available at http://www-nrg.ee.lbl.gov/

[3] Gary R. Wright, W. Richard Stevens, TCP-IP illustrated Volume 2, chapter 31. Addison-Wesley professional computing series.

[4] Microsoft Software Development Kit and Driver Development Kit Examples, Microsoft Corporation.

[5] Lew Perin, Bugs in the NT DDK Packet Protocol Driver Sample, Internet page. Available at http://www.panix.com/~perin/packetbugs.html

[6] Simpson, W., Editor, The Point-to-Point Protocol (PPP), RFC 1548, Daydreamer, December 1993.

[7] Microsoft Corporation, 3Com Corporation, NDIS, Network Driver Interface Specification, May 1988

[8] Microsoft Windows 95, Windows 98, Windows NT and Windows 2000 Driver Development Kit documentation, Microsoft Corporation.

[9] Peter G. Viscarola, W. Anthony Mason, Windows NT Device Driver Development, Macmillan Technical publishing.

[10] Microsoft MSDN Library, Microsoft Corporation, August 1999.

[11] Ricardo Thompson (icardoth@interserver.com.ar), Cpumeter, available on the Internet at http://www.winsite.com/info/pc/win95/sysutil/cpumet12.zip/, 1997

[12] A. Begel, S. McCanne, S.L.Graham, BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture, 1999.

[13] M. Yuhara, B. Bershad, C. Maeda, J.E.B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In Proceedings of the 1994 Winter USENIX Technical Conference, pages 153-165, San Francisco, CA, January 1994.

[14] Windump web site, available at http://netgroup-serv.polito.it/windump

[15] winpcap web site, available at http://netgroup-serv.polito.it/winpcap