UNIVERSITÀ DI PISA

Corso di Laurea in Ingegneria Informatica

Indirizzo Sistemi ed Applicazioni Informatici

Sviluppo di un sistema di schedulazione del traffico ed emulazione di rete per Windows

Autore:

Francesco Magno

Relatori:

Prof. Luigi Rizzo

Ing. Giovanni Stea

Anno Accademico 2009-2010

Indice

Elenco delle figure

Elenco dei listati

Sommario

L'obiettivo di questo lavoro è lo sviluppo di un sistema di schedulazione del traffico ed emulazione di rete per Windows. Per ottenere il risultato desiderato abbiamo scelto di effettuare il porting a Windows di *dummynet*[3, 4], un software già esistente che implementa molte delle funzionalità richieste.

Dummynet è uno strumento creato originariamente per il test dei protocolli di rete, divenuto col tempo una piattaforma completa di emulazione, che consente di effettuare *bandwidth shaping*, inserire ritardi, filtrare selettivamente il traffico, simulare situazioni di congestione, perdita di pacchetti, routing asimmetrico.

Sviluppato inizialmente per FreeBSD, ne è stato realizzato un porting per Linux, MacOS, e scopo di questa tesi è stata la realizzazione della versione per Windows. È stato quindi creato sia un modulo che implementa le funzionalità di filtraggio e di scheduling dei pacchetti, sia il programma utente *ipfw* necessario a configurarlo.

Essendo il codice FreeBSD originario ampiamente testato, è stato scelto di ridurre al minimo le differenze tra le varie piattaforme, utilizzando uno strato di compatibilità che si interfaccia con il sistema operativo secondo le specifiche imposte dalla Microsoft, e col codice originale utilizzando le strutture dati derivate direttamente dal kernel di Linux e FreeBSD, in questo modo è stata minimizzata la possibilità di errore, e si è ristretta notevolmente la porzione di codice su cui effettuare il debug.

Il porting del programma utente *ipfw* è stato effettuato su Cygwin, un ambiente di sviluppo che fornisce un'emulazione di molte delle API dei sistemi basati sullo standard POSIX, quali Linux e FreeBSD. Cygwin fornisce inoltre il compilatore standard GCC e una buona quantità di header compatibili. Il lavoro in spazio utente è consistito sostanzialmente nell'adattamento di alcuni header, e nel rimappaggio dei meccanismi di comunicazione kernel-userspace.

Per quanto riguarda il driver invece, lo sviluppo è stato effettuato utilizzando il *Driver Development Kit* fornito dalla Microsoft, un pacchetto che comprende, oltre al compilatore vero e proprio e agli header necessari alla realizzazione di un driver

Windows, una serie di tool proprietari per la gestione del progetto, la configurazione del compilatore, la verifica statica del codice e gli strumenti di debug.

Il porting del driver è stato realizzato in tre fasi:

- Creazione di header *ad-hoc*. L'ambiente di sviluppo Microsoft non contiene gli header necessari, si sono dovute quindi ricostruire tutte le strutture dati non proprietarie utilizzate da *dummynet*.

- Strato di compatibilità e rimappaggio delle funzioni. È stato creato lo strato di compatibilità che reimpacchetta il flusso di dati proveniente dal kernel di Windows nelle strutture dati originarie di FreeBSD, spostando la logica di queste conversioni in apposite funzioni, utilizzando quindi un *design pattern* di tipo proxy. Sono state quindi individuate e rimappate le funzioni del kernel di FreeBSD che hanno un analogo Windows (sincronizzazione tra threads e protezione delle sezioni critiche), mentre sono state implementate *ex novo* le funzioni non presenti o non compatibili (timer ricorrenti, intercettazione di pacchetti, emissione di pacchetti da e verso le interfacce di rete).

- Realizzazione del driver vero e proprio. Sono state realizzate le funzioni proprietarie di un driver Windows, il caricamento e lo scaricamento, il posizionamento nello stack di rete, l'aggancio al protocollo e alle interfacce di rete.

Si è scelto di sviluppare il modulo come un *NDIS intermediate filter driver*, un oggetto che si colloca nello stack di rete a metà strada tra il protocollo TCP/IP e le interfacce, intercetta tutti i pacchetti in entrata e in uscita dal sistema, li confeziona nelle opportune strutture dati e li passa al firewall, che può lasciarli passare, bloccarli, oppure conservarli in una coda interna per poi reimmetterli quando necessario.

Dummynet può essere immaginato come una scatola nera di codice intrinseco, con diversi ingressi e uscite, che lavora principalmente su strutture dati proprietarie e su liste. È evidente che, una volta forniti al compilatore gli header di queste strutture dati, e le definizioni dei tipi non presenti in ambiente Microsoft, il *core* di *dummynet* può essere compilato senza alcuna difficoltà.

Dobbiamo quindi focalizzare l'analisi su questi ingressi ed uscite, che sono le funzioni con cui questa scatola nera comunica con il resto del sistema operativo.

- Meccanismi di comunicazione tra userland e kernel: nel capitolo 1 vengono trattate le tecniche utilizzate per l'iniezione delle regole di filtraggio ed emulazione, e la lettura delle statistiche;
- Emulazione della funzione `sysctl`: nel capitolo 2 viene spiegata l'implementazione di un meccanismo di *fine tuning* di alcuni paramentri interni di funzionamento del modulo, derivato direttamente dai sistemi POSIX;
- Hook di intercettazione e di emissione spontanea dei pacchetti: nel capitolo 3 viene mostrata la struttura interna del modulo, come viene intercettato, elaborato e reinserito il traffico di rete;
- Timer e letture del system time: nel capitolo 4 vengono documentate le tecniche di sincronizzazione e temporizzazione del modulo;
- Semafori, funzioni standard di libreria, allocazione e deallocazione della memoria: nel capitolo 5 viene elencata l'utilizzazione delle cosiddette *kernel API*, cioè di quelle funzioni che ogni sistema operativo mette a disposizione del programmatore per effettuare operazioni di basso livello, in ambiente privilegiato;
- Adattamento dell'ambiente di sviluppo e test: nel capitolo 6 viene descritto l'ambiente di sviluppo utilizzato, i test finali del codice, e le estensioni per le differenti versioni di Windows a 64 bit;
- Istruzioni per l'utilizzo: infine nel capitolo 7 viene spiegato come installare il modulo, e vengono forniti esempi per le configurazioni tipiche di utilizzo.

Nonostante le limitazioni imposte da un sistema operativo non orientato alla gestione e alla manipolazione del traffico di rete, quale è Windows, si è ottenuto un port completo dell'applicazione originale.

Capitolo 1

Meccanismi di comunicazione tra userland e kernel

"È giunto il momento," disse il tricheco, "di parlare di molte cose."

L. Carroll

Convenzionalmente lo spazio di memoria dei sistemi operativi è suddiviso in uno strato utente, in cui risiedono gli applicativi, e uno strato kernel, dove risiedono le strutture dati del sistema (lo stack di rete, ad esempio) ed i moduli. Vista l'importanza e la delicatezza di queste strutture dati, l'area di memoria del kernel è separata, e all'utente è negata la possibilità di accedervi direttamente, se non tramite opportune system call. Le system call dedicate alla comunicazione con un device harware, tipicamente prendono il nome di *Device Input and Output Control (IOCTL)*.

Nel codice originale l'interfaccia di controllo avviene mediante la creazione, da parte dell'applicativo utente, di un socket, che viene gestito tramite le system call `getsockopt()` e `setsockopt()`. Il modulo del kernel registra gli hook necessari affinchè venga chiamata la funzione corretta per gestire le richieste da parte dell'utente.

Sotto Windows invece viene utilizzato il meccanismo degli *IOCTL*.

1.1 Operazioni sul lato kernel

Un driver Windows che voglia permettere la comunicazione con lo spazio utente deve quindi effettuare le seguenti operazioni:

- Assegnarsi un nome nello spazio dei device, visibile solo a livello kernel, e presente nella directory `\Device\` del filesystem virtuale del sistema.
- Assegnarsi un nome nello spazio dei device visibili dall'utente, presenti nella directory `\DosDevices\` del filesystem virtuale.
- Creare un link simbolico che colleghi virtualmente questi due nomi.
- Inizializzare l'array delle dispatch routine, chiamate *IRP Major Fuction*[10], con puntatori alle relative funzioni handler, ed implementare almeno i gestori per le richieste di tipo `IRP_MJ_CREATE`, `IRP_MJ_CLOSE`, `IRP_MJ_CLEANUP`, e `IRP_MJ_DEVICE_CONTROL`. Tipicamente, per un driver che non accede a periferiche fisiche, è sufficiente una funzione vuota per i primi tre handler, mentre il quarto deve gestire la richiesta vera e propria.
- Registrarsi con una funzione della famiglia `RegisterDevice()` per confermare queste operazioni e rendersi disponibile alle richieste.

1.2 Operazioni sul lato utente

Un programma utente che voglia creare un canale di comunicazione con un modulo del kernel deve effettuare le seguenti operazioni:

- Allocare un `HANDLE`. Un `HANDLE` è operativamente un puntatore a void, definito in `Windows.h`. Da un punto di vista logico, un `HANDLE` è un oggetto opaco che serve al programmatore per relazionarsi con una serie di entità di cui non si deve conoscere la struttura interna, e che non vanno gestite direttamente. Saranno sempre le WIN32 API a inizializzare un `HANDLE`, a verificarne la sua validità, e a gestirlo correttamente a seconda della funzione chiamata.
- Aprire il device utilizzando il nome visibile all'utente, presente nella directory `\\\\.\\`, che è un alias della directory `\DosDevices\` del filesystem virtuale. Questa operazione, in caso di successo, inizializza l'`HANDLE` precedentemente allocato, che sarà utilizzato per inviare i dati nel canale di comunicazione.

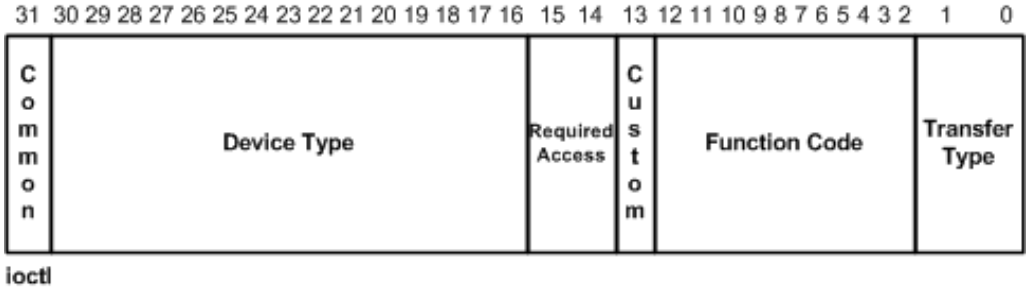

Figura 1.1: Struttura di un I/O control code

- Effettuare la comunicazione, utilizzando la funzione `DeviceIoControl()`[6], fornendo il codice di controllo dell'operazione, l'indirizzo e la lunghezza dei buffer di ingresso e uscita precedentemente allocati, necessari a contenere i dati relativi alla richiesta e alla relativa risposta da parte del device. È possibile utilizzare lo stesso buffer sia per l'input che per l'output.

1.3 Definizione dei codici di controllo

La Microsoft mette a disposizione una serie di codici di controllo predefiniti, per funzioni generiche di driver generici. È tuttavia possibile, ed opportuno, definire dei codici di controllo specifici per il driver in sviluppo, utilizzando la macro `CTL_CODE` definita in `ntddk.h`[5].

Un codice di controllo è un valore a 32 bit la cui struttura è mostrata in figura 1.1. I campi `DeviceType` e `FunctionCode` permettono di identificare univocamente il codice di controllo, e sono scelti in modo arbitrario nel range di valori non riservati alla Microsoft. Il campo `RequiredAccess` è un parametro di sicurezza usato per limitare gli eventuali permessi di lettura e scrittura, e prevenire chiamate illegali.

Il campo `TransferType` indica il metodo che verrà usato dal sistema operativo per gestire i buffer di I/O e prevede tre possibili valori:

- `METHOD_BUFFERED`. È il metodo utilizzato quando la comunicazione prevede il trasferimento di piccole quantità di dati, è il più sicuro ma anche il meno efficiente. Il sistema operativo alloca un buffer di ingresso/uscita la cui dimensione è uguale al massimo tra quella del buffer di ingresso e quella del buffer di uscita forniti nella chiamata di `DeviceIoControl()`. In questa zona di memoria viene effettuata la copia dei dati presenti nel buffer di ingresso dell'utente, il driver quindi provvede all'elaborazione della richiesta, ed even-

tualmente sovrascrive i dati e notifica la quantità di bytes che devono essere ricopiati nel buffer di uscita dell'utente.

- `METHOD_IN_DIRECT` o `METHOD_OUT_DIRECT`. È il metodo più efficiente poichè non effettua alcuna copia, ma semplicemente rimappa gli indirizzi userspace dei buffer forniti dal chiamante in indirizzi validi in spazio kernel, sui quali verrà effettuata direttamente l'elaborazione. Si utilizza `IN` nelle richieste che non necessitano di una risposta da parte del driver, `OUT` altrimenti.

- `METHOD_NEITHER`. Con questo metodo il sistema operativo non fornisce alcun buffer, e non effettua alcun remapping. Al driver viene fornito l'indirizzo userspace non modificato, ed è sua cura accertarsi che sia valido, accessibile, e che sia possibile effettuare operazioni di lettura e scrittura.

1.4 Implementazione

Per implementare la comunicazione tra userspace e kernel abbiamo realizzato uno strato di compatibilità che in *ipfw* rimappa le funzioni di creazione e gestione del socket, e nell'handler del `IRP_MJ_DEVICE_CONTROL` effettua le opportune trasformazioni sui dati prima di richiamare l'elaborazione di *dummynet*. In questo modo utilizziamo il sistema *IOCTL* appena descritto per incapsulare le strutture dati originali.

1.4.1 La `struct sockopt`

Il listato 1.1 mostra la definizione della struttura `sockopt` utilizzata per emulare una chiamata di `setsockopt()` o `getsockopt()` da parte del client.

Listato 1.1: La `struct sockopt`

```
enum sopt_dir { SOPT_GET, SOPT_SET };
struct  sockopt {
        enum    sopt_dir sopt_dir;
        int     sopt_level;
        int     sopt_name;
        void   *sopt_val;
        size_t  sopt_valsize;
        struct  thread *sopt_td;
};
```

Il valore `sopt_dir` discrimina una *get* da una *set*, i quattro valori successivi memorizzano i quattro parametri attuali delle funzioni `sockopt`, l'ultimo memorizza informazioni relative al processo chiamante, ma non viene utilizzato in Windows.

1.4.2 Le chiamate in userspace

Sul lato utente, vengono mostrate nei listati 1.2 e 1.3 le righe più significative del wrapping. Tramite opportune direttive di preprocessing vengono rimappate le funzioni `socket()` e `getsockopt()` su `mysocket()` e `wnd_getsockopt()` rispettivamente.

Listato 1.2: Creazione del socket

```
int my_socket(int domain, int ty, int proto)
{
        TCHAR *pcCommPort = TEXT("\\\\.\\Ipfw");
        HANDLE _dev_h = INVALID_HANDLE_VALUE;
        _dev_h = CreateFile (pcCommPort,
                GENERIC_READ | GENERIC_WRITE,
                0, NULL,
                OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        return (int)_dev_h;
}
```

Listato 1.3: Wrapping di `getsockopt`

```
int
wnd_getsockopt(int s, int level, int sopt_name, void *optval,
                socklen_t *optlen)
{
    size_t len = sizeof (struct sockopt) + *optlen;
    struct sockopt *sock;
    DWORD n;
    BOOL result;
    HANDLE _dev_h = (HANDLE)s;

    sock = malloc(len);
    if (sock == NULL)
        return -1;

    sock->sopt_dir = SOPT_GET;
    sock->sopt_name = sopt_name;
    sock->sopt_valsize = *optlen;
    sock->sopt_val = (void *)(sock+1);

    memcpy (sock->sopt_val, optval, *optlen);

    result = DeviceIoControl (_dev_h, IP_FW_GETSOCKOPT, sock, len,
                sock, len, &n, NULL);
```

```
    *optlen = sock->sopt_valsize;
    memcpy (optval, sock->sopt_val, *optlen);
    free (sock);
    return (result ? 0 : -1);
}
```

Alla chiamata di `socket` viene creato un `HANDLE` usato per aprire il canale di comunicazione col driver tramite `CreateFile()`. Alla chiamata di `getsockopt` vengono forniti dal chiamante:

- il descrittore del socket, che nel nostro caso è l'`HANDLE` restituito dalla prima funzione;
- il parametro `level`, che non viene utilizzato;
- il parametro `sopt_name` che contiene il tipo di operazione da effettuare (aggiungere una regola per il firewall, configurare i parametri di una pipe...),
- i parametri `optval` e `optlen` che contengono rispettivamente l'indirizzo e la dimensione del buffer contenente i dati da inviare, o sui quali si attende la risposta.

La funzione alloca una zona di memoria contigua grande a sufficienza da contenere la struttura `sockopt` ed il buffer del chiamante, vengono quindi copiati i parametri attuali nei corrispondenti campi della struttura, e il puntatore `sock->sopt_val` viene fatto puntare al primo byte successivo alla struttura `sockopt`.

Viene chiamata la `DeviceIoControl()` specificando come buffer unico di ingresso/uscita l'indirizzo di questa zona contigua di memoria. Al ritorno, il campo `sock->sopt_valsize` contiene la dimensione in bytes dei dati sovrascritti dal driver, i quali vengono ricopiati nel buffer fornito dal chiamante.

La funzione `setsockopt()` effettua le stesse operazioni, ma non restituisce alcun dato al chiamante.

1.4.3 Il gestore della richiesta

La funzione di inizializzazione di un driver Windows è chiamata `DriverEntry()`, e deve provvedere, nel caso specifico di un *NDIS intermediate filter driver*, a riempire le strutture *characteristics* con i puntatori a funzione relativi a tutte le possibili chiamate del modulo: inizializzazione e cleanup dei suoi diversi componenti, e le callback.

Nel listato 1.4 vengono mostrate le righe relative alle procedure utilizzate per inizializzare il gestore delle richieste dell'utente, presenti nella `DriverEntry()` e nella `PtRegisterDevice()`. Si eseguono esattamente le procedure precendetemente descritte, creazione dei nomi nel filesystem virtuale, inizializzazione dei puntatori a funzione per le dispatch, registrazione del device. La dispatch relativa alle richieste `IRP_MJ_CREATE`, `IRP_MJ_CLEANUP` e `IRP_MJ_CLOSE` si limita a ritornare uno stato successful; la dispatch per la richiesta `IRP_MJ_DEVICE_CONTROL` viene mostrata nel listato 1.5 e richiede una trattazione più approfondita.

Listato 1.4: Inizializzazione del gestore

```
DispatchTable[IRP_MJ_CREATE] =          PtDispatch;
DispatchTable[IRP_MJ_CLEANUP] =         PtDispatch;
DispatchTable[IRP_MJ_CLOSE] =           PtDispatch;
DispatchTable[IRP_MJ_DEVICE_CONTROL] =  DevIoControl;

#define LINKNAME_STRING     L"\\DosDevices\\Ipfw"
#define NTDEVICE_STRING     L"\\Device\\Ipfw"
NdisInitUnicodeString(&DeviceName, NTDEVICE_STRING);
NdisInitUnicodeString(&DeviceLinkUnicodeString, LINKNAME_STRING);

Status = NdisMRegisterDevice(
                    NdisWrapperHandle,
                    &DeviceName,
                    &DeviceLinkUnicodeString,
                    &DispatchTable[0],
                    &ControlDeviceObject,
                    &NdisDeviceHandle
                    );
```

Listato 1.5: Gestore `DevIoControl`

```
NTSTATUS
DevIoControl(
    IN PDEVICE_OBJECT    pDeviceObject,
    IN PIRP              pIrp
    )
{
        struct sockopt *sopt;
        int ret = 0;

        pIrpSp = IoGetCurrentIrpStackLocation(pIrp);
        sopt = pIrp->AssociatedIrp.SystemBuffer;

        FunctionCode = pIrpSp->Parameters.DeviceIoControl.IoControlCode;

        len = sopt->sopt_valsize;

        switch (FunctionCode)
        {
```

```
        case IP_FW_SETSOCKOPT:
            ret = do_ipfw_set_ctl(NULL, sopt->sopt_name, sopt+1, len);
            break;

        case IP_FW_GETSOCKOPT:
            ret = do_ipfw_get_ctl(NULL, sopt->sopt_name, sopt+1, &len);
            sopt->sopt_valsize = len;
            BytesReturned = len + sizeof(struct sockopt);
            break;
        }

        pIrp->IoStatus.Information = BytesReturned;
        pIrp->IoStatus.Status = NtStatus;
        IoCompleteRequest(pIrp, IO_NO_INCREMENT);
        return tStatus;
}
```

L'argomento significativo che viene passato al gestore `DevIoControl()` è un puntatore ad un *IRP*, o *I/O Request Packet*. Un *IRP* è una struttura parzialmente opaca, che si può sintetizzare come implementata da un *MDL*, o *Memory Descriptor List*, cioè una lista di descrittori a zone di memoria potenzialmente non contigue, seguita da metadati di controllo. Una trattazione completa della gestione della memoria e del modello di I/O nel kernel di Windows esula dai nostri scopi e pertanto si rimanda a [9, 12, 14].

Tralasciando i sanity check che non sono stati riportati per brevità, si vede che il gestore acquisisce l'indirizzo del buffer di I/O e il comando associato alla richiesta. Questo comando è il control code che è stato definito precedentemente in un header in comune tra modulo e programma utente, e non la funzionalità richiesta da *ipfw*. Siamo ancora nel layer più esterno dell'incapsulamento della richiesta e pertanto questo valore discrimina solo l'emulazione di una *set* piuttosto che di una *get*.

A questo punto la funzione si può interfacciare con *dummynet* passando la richiesta di *ipfw*, memorizzata in `sopt->sopt_name`, ed il relativo buffer di memoria con la sua lunghezza. Particolare attenzione va posta sul fatto che nella struttura `sockopt` ricevuta, il valore del puntatore `sopt_val` non è l'indirizzo corretto da passare a *dummynet*. Nella nostra implemetazione abbiamo scelto il `METHOD_BUFFERED`, poichè l'irrisoria quantità di dati scambiati non giustificava l'utilizzo di un metodo più efficiente ma meno sicuro, pertanto quello che arriva alla dispatch routine è una copia del buffer di ingresso del chiamante, e l'indirizzo presente in `sopt_val` si riferisce alla zona dati allocata dal chiamante, nello spazio degli indirizzi destinati all'utenza, che quindi non è un indirizzo valido nel contesto del driver, ma che, rimanendo immutato durante tutta la procedura, sarà di nuovo valido alla fine della

dispatch, quando ci sarà il ritorno della `getsockopt()`.

L'elaborazione della richiesta va effettuata sulla zona dati del buffer dell'*IRP* fornito dal sistema operativo, che quindi si trova all'indirizzo `sopt+1`. Soltanto alla fine dell'elaborazione della dispatch sarà il sistema operativo a ricopiare il buffer di I/O contenuto nell'*IRP* sul buffer di uscita fornito dal chiamante.

Nel caso di una *get* completiamo la procedura scrivendo in `sopt->sopt_valsize` la lunghezza dei dati che torneranno a *ipfw*, e in `pIrp->IoStatus.Information` la lunghezza dei dati presenti nel buffer di I/O dell'*IRP* che il sistema operativo dovrà ricopiare nel buffer del chiamante in userspace, e che ovviamente è uguale alla somma del precedente valore e della dimensione della `struct sockopt` di incapsulamento.

Capitolo 2

Emulazione della funzione `sysctl`

La perfezione si raggiunge solamente al momento del collasso

C.N. Parkinson

2.1 Le funzioni `sysctl`

La funzione `sysctl`[1] fornisce, su piattaforme FreeBSD e Linux, un meccanismo per esaminare, ed eventualmente modificare, dei parametri del kernel. In FreeBSD questi parametri vengono impostati mappando le zone di memoria in cui risiedono queste variabili globali in indirizzi accessibili dallo spazio utente tramite apposite *system calls*; in Linux invece, seguendo la filosofia *"everything is a file"*, vengono implementate all'interno dei filesystem virtuali `/sys` e `/proc`. Esempio tipico dell'utilizzo di `sysctl` è l'abilitazione dell'*ip forwarding* settando a 1 il parametro `net.inet.ip.forwarding`. In Windows un meccanismo del genere è presente solo nel registro, di cui è ben nota sia la delicatezza, sia la scarsa intuitività di utilizzo; inoltre le restrizioni per un driver che voglia interagire con il registro sono abbastanza pesanti; un approccio a *callback* sarebbe stato difficile da implementare, un approccio a *polling* assolutamente improponibile in termini di overhead; inoltre, l'accesso al pannello di configurazione di un driver di rete, è un'operazione decisamente macchinosa e poco intuitiva

È stato quindi opportuno creare una soluzione *ad hoc*, che:

- fornisse uno strato di compatibilità per le macro di FreeBSD presenti nel codice originale, garantendo quindi possibilità di espansione futura in modo completamente trasparente al programmatore;
- implementasse un meccanismo di comunicazione bidirezionale tra userland e kernel;
- estendesse le funzionalità di *ipfw* per utilizzare questo meccanismo quando presente.

È stato quindi ovvio scegliere di veicolare la comunicazione attraverso il meccanismo delle `sockopt` già descritto nel capitolo 1. Inoltre tutto il codice relativo all'emulazione della `sysctl` si pone al di sopra dello strato di compatibilità utilizzato per il porting, e utilizza le funzioni di generazione di richieste interne di *ipfw*, pertanto il meccanismo è assolutamente indipendente dalla specifica implementazione Windows, e può essere riutilizzato su altre piattaforme senza alcuna modifica.

2.2 Manipolazione delle macro di FreeBSD

Nel listato 2.1 viene mostrata la sintassi di una macro FreeBSD utilizzata per definire un parametro da esportare in spazio utente.

Listato 2.1: Le macro `sysctl` in FreeBSD

```
SYSCTL_INT(_net_inet_ip_dummynet, OID_AUTO, hash_size,
    CTLFLAG_RW, &dn_cfg.hash_size, 0, "Default hash table size");
```

Si notano gli argomenti che determinano il nome della variabile, il controllo di accesso, l'indirizzo della variabile e la sua descrizione. Il preprocessore C, per quanto potente, non permette una manipolazione tale da eliminare il punto e virgola alla fine delle macro, per cui l'unica soluzione possibile è stata di tradurle in statement. La posizione di queste dichiarazioni, nello spazio delle variabili globali, ha quindi richiesto la definizione di altre macro, `SYSBEGIN` e `SYSEND`, per racchiuderle in una funzione, in cui ogni statement esegue un *pushback* della singola entry in un array definito globalmente, in modo analogo al `valarray::push_back()` della C++ Standard Library.

Listato 2.2: Rimappaggio delle macro

```
#define SYSBEGIN(x) void sysctl_addgroup_##x() {
#define SYSEND }

#define SYSCTL_INT(a,b,c,d,e,f,g)                                   \
        sysctl_pushback(STRINGIFY(a) "." STRINGIFY(c) + 1,          \
                (d) | (SYSCTLTYPE_INT << 2), sizeof(*e), e)
```

Nel listato 2.2 si vede come a partire dai parametri `a` e `c` venga ricostruito il nome del valore, che tuttavia nel codice originale presenta underscore al posto dei punti, viene quindi passato il flag che determina il tipo ed il controllo di accesso, l'indirizzo della variabile e la sua dimensione. L'indirizzo della stringa statica ricostruita viene incrementato di uno, per eliminare l'underscore iniziale, mentre gli altri saranno sostituiti da un'apposita funzione in fase di inizializzazione. Il parametro `d`, invece, prevede l'utilizzo dei due bit meno significativi per il controllo d'accesso, e i restanti per determinare il tipo della variabile, che in questo caso particolare è `int`. Il meccanismo è già predisposto per l'estensione ad altri tipi di valore, anche se al momento *dummynet* esporta principalmente interi signed o unsigned a 32 bit.

2.3 Le strutture dati

Come si è già visto, l'unica strada percorribile è stata quella di utilizzare una struttura dati globale con al suo interno un array statico, che viene riempito in fase di inizializzazione con opportune entry. Nel listato 2.3 vengono riportate le strutture dati della *Global Sysctl Table* e delle singole entry. In `count` viene conservata la quantità di entry valide inserite dell'array, in `totalsize` la dimensione in memoria dello stesso, che viene calcolata in fase di inizializzazione tenendo conto della lunghezza dei nomi, e dei dati; questo valore non cambia mai durante la vita del driver, è quindi utile calcolarlo al caricamento, per evitare inutile overhead durante la comunicazione con *ipfw*.

Ciascuna entry è composta da una `struct sysctlhead`, definita in comune tra kernel e spazio utente, che contiene la lunghezza del blocco di memoria, la lunghezza del nome della variabile esportata, i flag di tipo e di accesso, e la lunghezza dei dati; queste sono tutte le informazioni coinvolte nel trasferimento in spazio utente. La definizione dell'entry vera e propria è invece locale al kernel, e contiene anche i puntatori alle zone di memoria contenenti la stringa e il dato, questi puntatori non vengono trasferiti in spazio utente, poichè soffrirebbero degli stessi problemi

evidenziati nel capitolo 1: sarebbero locali allo spazio di indirizzamento del kernel, e quindi non validi.

Listato 2.3: Le strutture dati del meccanismo `sysctl`

```
#define GST_HARD_LIMIT 100

struct sysctlhead {
        uint32_t blocklen; //total size of the entry
        uint32_t namelen; //strlen(name) + '\0'
        uint32_t flags; //type and access
        uint32_t datalen;
};

struct sysctlentry {
        struct sysctlhead head;
        char* name;
        void* data;
};

struct sysctltable {
        int count; //number of valid tables
        int totalsize; //total size of valid entries of al the valid tables
        void* namebuffer; //a buffer for all chained names
        struct sysctlentry entry[GST_HARD_LIMIT];
};
```

2.4 Inizializzazione delle strutture dati al caricamento del modulo

Nel listato 2.4 viene mostrato come viene tradotta la macro del codice originale, e la funzione di *pushback* richiamata dalla macro.

Listato 2.4: La funzione di *pushback*

```
void
sysctl_addgroup_f1()
{
        sysctl_pushback("_net_inet_ip_dummynet" "." "hash_size" + 1,
                    (CTLFLAG_RW) | (SYSCTLTYPE_INT << 2),
                    sizeof(*&dn_cfg.hash_size), &dn_cfg.hash_size);
}

void
sysctl_pushback(char* name, int flags, int datalen, void* data)
{
        struct sysctlentry *entry;
        if (GST.count >= GST_HARD_LIMIT) {
                printf("WARNING: global sysctl table full, "
```

```
                        "this entry will not be added,"
                        "please recompile the module "
                        "increasing the table size\n");
                return;
        }
        entry = GST.entry[GST.count];
        entry->head.namelen = strlen(name)+1; //add space for '\0'
        entry->name = name;
        entry->head.flags = flags;
        entry->data = data;
        entry->head.datalen = datalen;
        entry->head.blocklen =
                ((sizeof(struct sysctlhead) + entry->head.namelen +
                        entry->head.datalen)+3) & ~3;
        GST.totalsize += entry->head.blocklen;
        GST.count++;
}
```

Viene creata una `sysctl_addgroup_XX()` per ciascuno dei file del progetto che vuole dichiarare delle variabili accessibili tramite `sysctl`, e queste funzioni vengono chiamate in sequenza in fase di inizializzazione. Viene calcolata la lunghezza del nome, incrementata di uno per il terminatore di stringa, per i restanti campi viene fatto un semplice assegnamento. La dimensione del blocco viene calcolata come la somma della dimensione della struttura, la dimensione della stringa comprensiva di terminatore e la dimensione dei dati (dati e stringa risiedono in memoria al di fuori dell'array), e il risultato viene arrotondato ad un multiplo di 32 bit, per compatibilità con le piattaforme che non riescono a gestire trasferimenti non mod4. Alla fine di ciascun inserimento viene incrementato il contatore e la dimensione complessiva della memoria da trasferire alla richiesta dell'utente.

Alla fine di tutti gli inserimenti viene richiamata la funzione `formatnames()` che copia tutte le stringhe statiche generate dal preprocessore in una zona di memoria allocata dinamicamente, e converte gli underscore in punti per uniformarsi allo standard dei nomi delle piattaforme POSIX, quindi aggiorna i puntatori `name` delle singole entry alla locazione corretta. Questo passo è necessario, poichè le stringhe generate in fase di compilazione sono considerate costanti, e un tentativo di modificarle direttamente si traduce in un kernel panic.

Alla fine della fase di inizializzazione, la struttura dati completa, nella memoria del kernel, si presenta come in figura 2.1.

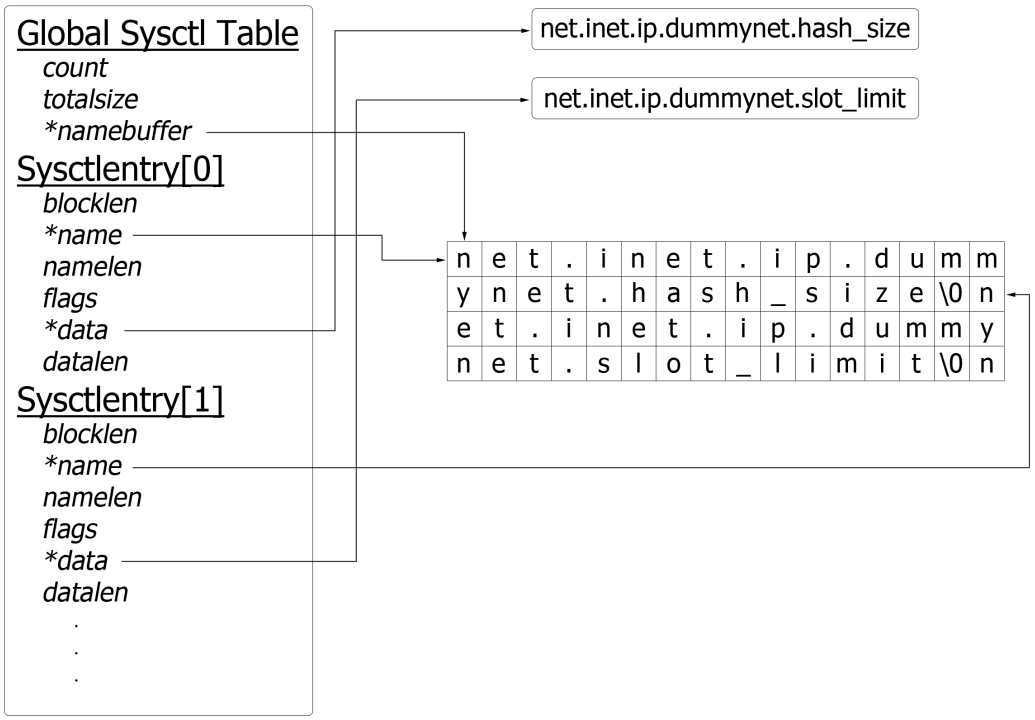

Figura 2.1: Global Sysctl Table

2.5 Le richieste dallo spazio utente

In spazio utente abbiamo reimplementato la funzione `sysctlbyname()` in modo da utilizzare l'emulazione `sysctl` per impostare e recuperare le variabili esportate nella memoria del kernel se chiamata con la sintassi tradizionale POSIX, e da fornire un servizio di stampa a video se chiamata con una sintassi speciale dal `main()` di *ipfw* se l'utente specifica l'apposito subcomando `./ipfw sysctl`.

La dichiarazione della funzione è mostrata nel listato 2.5. Se chiamata con `newp!=NULL` e `newlen!=0` viene interpretata come una *set* di un valore in memoria del kernel; se `oldp!=NULL` e `oldlenp!=NULL` viene interpretata come una *get*; se invece anche questi due puntatori sono `NULL`, viene interpretata come la speciale richiesta di una *print*, in cui `name` contiene una wildcard per la selezione della/e variabile/i da stampare. È possibile, opzionalmente, passare in `oldp` un descrittore di stream di tipo `FILE*` per reindirizzare l'output in uno stream diverso da `stdout`.

Listato 2.5: La dichiarazione di `sysctlbyname`

```
int sysctlbyname(const char *name, void *oldp, size_t *oldlenp,
        void *newp, size_t newlen);
```

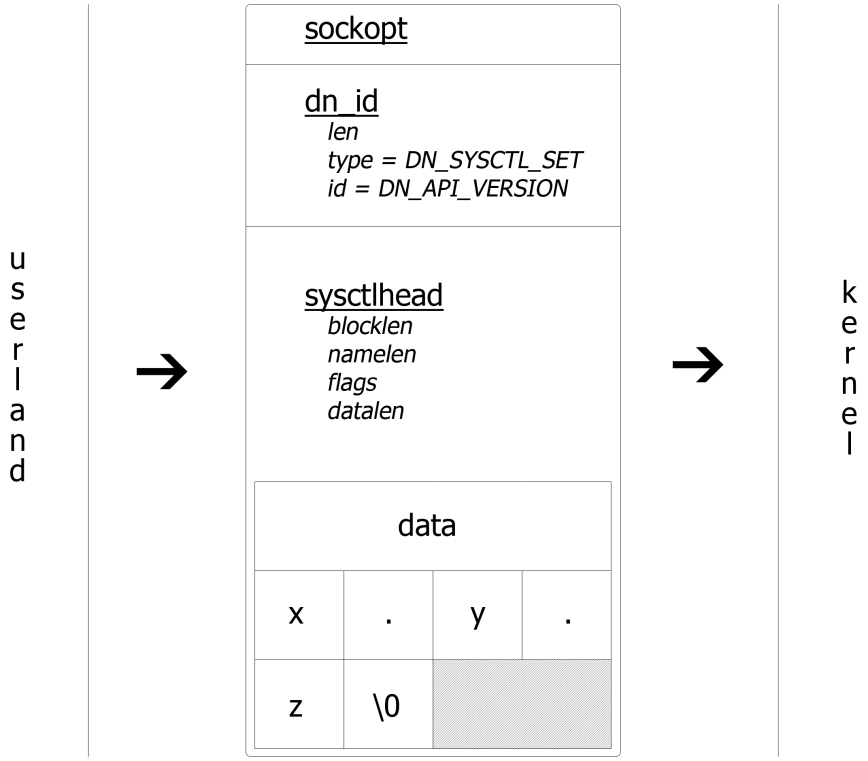

Figura 2.2: Trasferimento dati in emulazione `sysctl`

Durante un trasferimento di dati tra userland e kernel, indipendentemente dalla direzione, la quantità di dati di una singola entry dipende dalla lunghezza del nome e dei dati, e quindi non è nota a priori dal programma utente.

Come si è detto, viene riutilizzato il meccanismo delle `sockopt` per incapsulare la richiesta; il buffer, mostrato in figura 2.2, è quindi formato dalla sequenza di:

- una `struct dn_id`, che è la struttura generica utilizzata da *ipfw* per iniettare regole e comandi nel modulo;
- una `struct sysctlhead`, la cui definizione è stata fornita nel listato 2.3, che è la struttura utilizzata dal modulo per conservare informazioni sulle variabili esportate;
- il valore della variabile;
- il nome della variabile, seguito dal terminatore `\0`;
- un eventuale padding per rendere la dimensione del blocco multipla di 32 bit.

2.5.1 Il caso *set*

Nel caso di una *set* il programma utente acquisisce da riga di comando il valore da impostare e lo converte in un intero a 32 bit tramite la funzione di libreria `strtol()`, quindi costruisce la richiesta, come mostrato nel listato 2.6.

Viene inizializzata la struttura `dn_id` con la lunghezza complessiva del buffer, il subcomando e la versione delle API, quindi si inizializza la struttura `sysctlhead` riempiendo i campi relativi alle lunghezze, tenendo conto dei terminatori, e dell'eventuale padding. Vengono riempite le zone di memoria relative ai dati e al nome, e quindi viene inviata la richiesta tramite la `do_cmd()`, utilizzando il comando `IP_DUMMYNET3`.

Listato 2.6: `sysctlbyname` in una *set*

```
oid->len = l;
oid->type = DN_SYSCTL_SET;
oid->id = DN_API_VERSION;

entry = (struct sysctlhead*)(oid+1);
pdata = (unsigned char*)(entry+1);
pstring = pdata + newlen;

entry->blocklen = ((sizeof(struct sysctlhead) + strlen(name)+1
                   + newlen) + 3) & ~3;
entry->namelen = strlen(name)+1;
entry->flags = 0;
entry->datalen = newlen;

bcopy(newp, pdata, newlen);
bcopy(name, pstring, strlen(name)+1);

do_cmd(IP_DUMMYNET3, oid, (uintptr_t)l);
```

Sul lato kernel, all'interno della funzione che gestisce l'esecuzione dei comandi utente, la `do_config()`, viene intercettato il caso speciale di comando `IP_DUMMYNET3` e subcomando `DN_SYSCTL_SET`, e servita la richiesta:

- viene effettuato un parsing della struttura `GST`, per trovare un match sul nome della variabile;
- se trovato, vengono fatti dei *sanity check* sulla dimensione del dato;
- viene effettuato il controllo di accesso, per verificare che il dato non sia di tipo read-only;
- viene aggiornata la variabile in memoria del kernel.

2.5.2 Il caso *get* e *print*

Nel caso di una *get* il programma utente non può sapere, a priori, il numero e il tipo delle variabili esportate dal modulo, per questo invia una *probe request* utilizzando la stessa tecnica che viene impiegata per le altre richieste di tipo `IP_DUMMYNET3`. Viene costruito un buffer contente solo una `struct dn_id` riempita con il subcomando `DN_SYSCTL_GET`, la lunghezza, e la versione delle API, alla quale il kernel risponde intercettando nella `dummynet_get()` comando e subcomando relativi, e settando il campo `id` alla dimensione complessiva dei dati da trasferire, prima di restituire il buffer allo spazio utente. Questo valore è stato precedentemente calcolato e si trova nel campo `totalsize` della *Global Sysctl Table*.

Viene quindi allocata memoria sufficiente a contenere i dati, e viene effettuata una seconda richiesta, come mostrato nel listato 2.7. Essendo la dimensione della *GST* tipicamente piccola (nell'ordine dei 2kbyte), è più conveniente trasferire l'intero blocco in spazio utente e lasciare al client il compito di parsare i risultati, piuttosto che perdere tempo all'interno del kernel.

Listato 2.7: `sysctlbyname` in una *get*

```
oid->len = l;
oid->type = DN_SYSCTL_GET;
oid->id = DN_API_VERSION;

ret = do_cmd(-IP_DUMMYNET3, oid, (uintptr_t)&l);
l=oid->id;

oid->len = l;
oid->type = DN_SYSCTL_GET;
oid->id = DN_API_VERSION;

ret = do_cmd(-IP_DUMMYNET3, oid, (uintptr_t)&l);
```

Quando al modulo arriva una richiesta di tipo `DN_SYSCTL_GET`, il cui buffer è sufficientemente grande da contenere l'intera *GST* (la dimensione è ricavabile dall'header `sockopt`), viene costruita una lista delle entry, utilizzando la struttura già mostrata in figura 2.2. *Ipfw* non può sapere quante entry sono contenute nel buffer ritornato, viene quindi inserito un terminatore costituito da una `struct sysctlhead` in cui `blocklen` è settato a 0.

Il client, come mostrato nel listato 2.8 discrmina il tipo di operazione richiesta dai parametri attuali, e inizia il parsing del buffer:

- nel caso di una *get*
  - al match esatto del nome viene ricopiato il valore nel buffer fornito dal chiamante di `sysctlbyname()`;
- nel caso di una *print*
  - viene effettuato il match con il nome fornito dal chiamante e vengono stampate le entry in cui combaci il prefisso, in caso di match parziale, o l'unica entry in caso di match completo, oppure
  - vengono stampate tutte le entry se il nome è ‘‘`-a`’’, per mantenere la coerenza con la sintassi del comando di FreeBSD.

  La stampa viene effettuata tenendo conto della tipologia del dato, il cui valore è ricavato dai 30 bit più significativi del campo flag, e viene segnalato all'utente se il campo è read-only, in base al valore dei 2 bit meno significativi.

Listato 2.8: Parsing dei risultati in `sysctlbyname`

```
if(name != NULL && oldp != NULL && *oldlenp > 0)
{ //this is a get
    if(strcmp(name,pstring) == 0)
    { //match found, sanity chech on len
        if(*oldlenp < entry->datalen)
        {
            printf("%s error: buffer too small\n",__FUNCTION__);
            return -1;
        }
    *oldlenp = entry->datalen;
    bcopy(pdata, oldp, *oldlenp);
    return 0;
    }
}
else
{ //this is a print
    if( name == NULL )
        goto print;
    if ( (strncmp(pstring,name,strlen(name)) == 0) &&
       ( pstring[strlen(name)]=='\0' || pstring[strlen(name)]=='.' ) )
        goto print;
    else
        goto skip;
print:
```

```
    fprintf(fp, "%s: ",pstring);
    switch( entry->flags >> 2 )
    {
        case SYSCTLTYPE_LONG:
            fprintf(fp, "%li ", *(long*)(pdata));
            break;
        case SYSCTLTYPE_UINT:
            fprintf(fp, "%u ", *(unsigned int*)(pdata));
            break;
        case SYSCTLTYPE_ULONG:
            fprintf(fp, "%lu ", *(unsigned long*)(pdata));
            break;
        case SYSCTLTYPE_INT:
        default:
            fprintf(fp, "%i ", *(int*)(pdata));
    }
    if( (entry->flags & 0x00000003) == CTLFLAG_RD )
        fprintf(fp, "\t(read only)\n");
    else
        fprintf(fp, "\n");
skip:
}
```

Capitolo 3

Intercettazione del traffico, il driver *Passthru*

Le ottimizzazioni premature sono la fonte di tutti i mali

D. Knuth

D'altra parte non si può trascurare l'efficienza

J. Bentley

3.1 Struttura dei pacchetti nel kernel di Windows

In Windows i pacchetti vengono rappresentati da un oggetto parzialmente opaco di tipo `NDIS_PACKET`[11], definito nel file `ndis.h`, la cui gestione è completamente affidata alle funzioni ed alle macro della famiglia `NdisXxx`. Esplorando la sua definizione si nota la presenza di una parte privata, non direttamente accessibile, e di una sezione riservata ai *Miniport* ed una ai *Protocol*, che verranno discusse più avanti.

Le parti salienti della struttura interna di un pacchetto vengono mostrate in figura 3.1. Il `NDIS_PACKET` è un descrittore di pacchetto, che contiene al suo interno un puntatore ad una lista di `NDIS_BUFFER`, oltre a puntatori a dati *Out Of Band* che non tratteremo specificamente; ogni `NDIS_BUFFER` è in realtà un *MDL*, per la cui descrizione si veda [14], che contiene la lunghezza e l'indirizzo della zona di memoria

in cui si trovano i dati veri e propri, ed ovviamente un puntatore al `NDIS_BUFFER` successivo. I pacchetti IP generati localmente sono solitamente composti da quattro buffer, uno per i dati, ed uno per ciascun livello di incapsulamento, come mostrato in figura 3.1; quelli provenienti dalla scheda di rete sono invece generalmente composti da un solo buffer.

Per l'ispezione di un pacchetto, le principali funzioni da utilizzare sono le seguenti:

- `NdisQueryPacket` fornisce informazioni relative al pacchetto descritto da un particolare `NDIS_PACKET`
  - `BufferCount` contiene il numero dei `NDIS_BUFFER` collegati al descrittore di pacchetto,
  - `FirstBuffer` contiene l'indirizzo del primo `NDIS_BUFFER` della lista, che è memorizzato nel campo `Head` della parte privata del descrittore di pacchetto,
  - `TotalPacketLength` contiene la lunghezza totale, in byte, del pacchetto;
- `NdisQueryBuffer` fornisce informazioni relative al singolo `NDIS_BUFFER`, e va chiamata in sequenza per ciascun buffer della lista
  - `VirtualAddress` contiene l'indirizzo di partenza della zona di memoria descritta dal *MDL*,
  - `Length` contiene la lunghezza della zona di memoria descritta dal *MDL*;
- `NdisGetNextBuffer` ritorna il puntatore al successivo `NDIS_BUFFER` della lista, che è memorizzato nel capo `Next` di ciasun *MDL*.

3.2 Scelta della tipologia di driver

A partire da Windows 2000, vengono messe a disposizione del programmatore due tipologie di *Hook Driver* per implementare un firewall:

- i *Filter-Hook Driver*,
- i *Firewall-Hook Driver*.

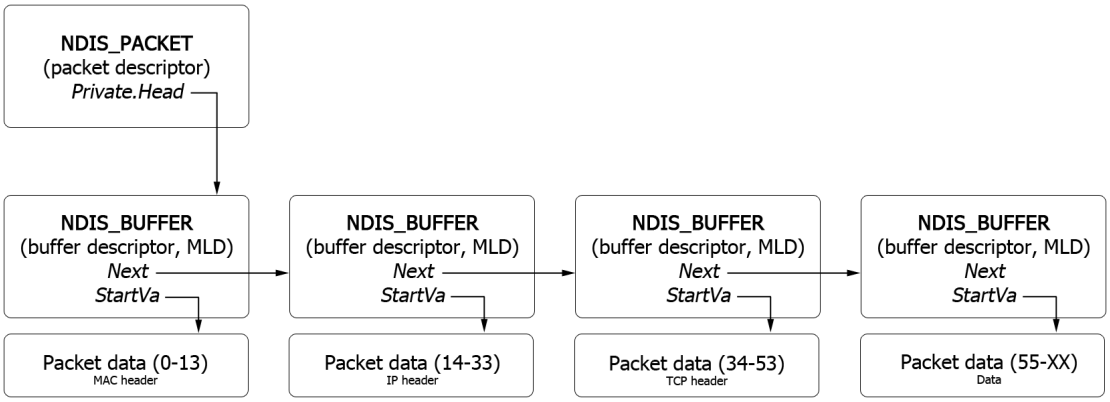

Figura 3.1: Struttura interna di un NDIS_PACKET

È divertente notare che la seconda categoria è completamente non documentata, ne viene sconsigliato l'utilizzo da parte della stessa Microsoft, tuttavia il firewall integrato di Windows XP è basato proprio su questa tipologia di driver. Per quanto riguarda il primo tipo, viene descritto come un kernel driver che estende le funzionalità dell'*IP filter driver* già presente, al fine di filtrare il traffico di rete. Questo tipo di driver, utilizzato nella fase iniziale del progetto e scelto per la sua semplicità realizzativa e per la sua efficienza, ha tuttavia rivelato subito la sua inadeguatezza per i nostri scopi:

- solo una istanza di un *Filter-Hook Driver* può essere presente sulla macchina, questo renderebbe inutilizzabile la presenza di altri firewall sullo stesso sistema, essendo *dummynet* non solo un firewall ma anche uno schedulatore di pacchetti, non si vuole impedire all'utente di utilizzare altre soluzioni per il solo filtraggio;

- a partire dal Service Pack 2 di Windows XP, è stata eliminata la possibilità di emissione spontanea di pacchetti, sia per i programmi utente che per i driver di alto livello (per motivi di sicurezza, essendo Windows un sistema notoriamente molto usato dagli hacker di tutto il mondo); in sostanza un *Filter-Hook Driver* può soltanto lasciar passare o droppare un pacchetto, ma non ha la possibilità di reimmettere in un secondo momento un pacchetto catturato da *dummynet*.

La scelta è ricaduta quindi su un *NDIS intermediate filter driver*, un driver di più basso livello, che si inserisce nello stack di rete tra il *Protocol*, il protocollo IP, ed i *Miniport*, le schede di rete. Una rappresentazione grafica delle due tipologie di driver è mostrata nelle figure 3.2 e 3.3.

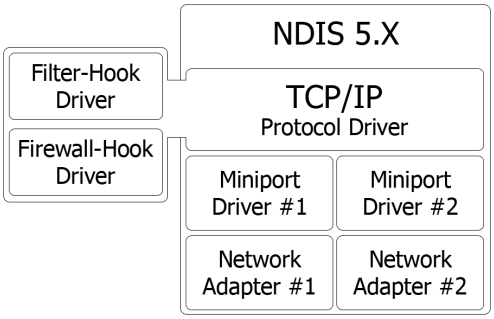

Figura 3.2: Un *filter-hook driver*

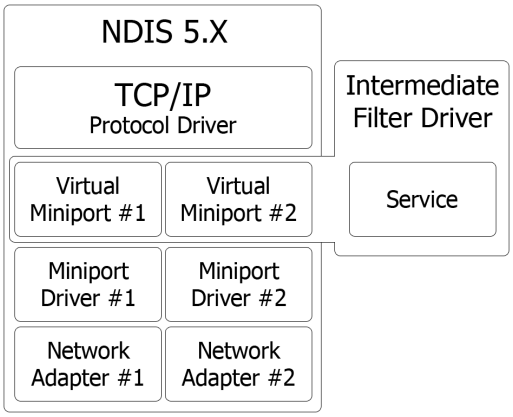

Figura 3.3: Un *intermediate driver*

Viene creata una singola istanza del driver, che viene vista dal sistema operativo come un *Service*, l'equivalente Windows di un demone di Unix, esterno allo stack di rete e inserito nel filesystem virtuale dei device generici come spiegato nel capitolo 1, più un *Miniport* virtuale per ciascuna scheda di rete installata nel sistema, i cui *binding* nello stack vengono gestiti automaticamente del sistema *NDIS* all'interno della relativa sezione del registro. Un *NDIS intermediate filter driver* non ha bisogno di settare nessun *hook*, poichè è la sua posizione nello stack che rende automatica l'intercettazione di tutto il traffico proveniente da entrambe le direzioni.

Lo scheletro di un *NDIS intermediate filter driver* viene fornito come esempio nel *Windows Driver Kit*, ed è chiamato *Passthru driver* poichè si limita a lasciar passare tutto il traffico senza esaminarlo nè modificarlo.

3.3 Il driver *Passthru*

È opportuno spiegare il funzionamento del driver *Passthru*, per capire quali sono state le modifiche apportate.

3.3.1 Impostazione delle *characteristics* del driver

Nella sezione 1.4.3 abbiamo già descritto quali sono i compiti della funzione `DriverEntry()` e le operazioni da effettuare per registrare un'interfaccia *IOCTL* per la comunicazione con lo spazio utente; di seguito, nel listato 3.1 vengono mostrate le *characteristics* necessarie alla gestione del traffico.

Listato 3.1: *Characteristics* per la gestione del traffico

```
MChars.SendHandler = MPSend;
MChars.SendPacketsHandler = NULL;
PChars.SendCompleteHandler = PtSendComplete;

PChars.ReceiveHandler = PtReceive;
PChars.ReceivePacketHandler = PtReceivePacket;
PChars.ReceiveCompleteHandler = PtReceiveComplete;
MChars.ReturnPacketHandler = MPReturnPacket;
```

Per i pacchetti in uscita dal sistema vengono assegnati i seguenti handler:

- `SendHandler`, richiamato quando il protocollo notifica la presenza di un pacchetto pronto per la spedizione;
- `SendPacketsHandler`, richiamato quando il protocollo notifica la presenza di un array di pacchetti pronti, questa funzione è opzionale e non viene utilizzata in quanto le primitive utilizzate da *dummynet* per l'emissione non prevedono questa possibilità, pertanto viene lasciato `NULL`;
- `SendCompleteHandler`, richiamato in maniera asincrona al completamento di un'operazione di *send*, per effettuare il cleanup dei descrittori utilizzati.

Per i pacchetti in ingresso al sistema vengono assegnati i seguenti handler:

- `ReceivePacketHandler`, richiamato quando una scheda di rete notifica la presenza di un pacchetto in entrata;
- `ReceiveHandler`, richiamato quando la scheda di rete notifica la presenza di dati generici in entrata, che non sono specificamente un pacchetto; questa condizione può verificarsi in concomitanza di una severa carenza di risorse, in presenza di traffico non IP, o in presenza di schede di rete particolarmente vecchie, o che sono montate su bus particolarmente lenti, dove può essere notificato un pacchetto parziale, o dove può essere utile che il protocollo determini se è interessato ai dati dal solo esame dell'header, per evitare il trasferimento del payload; tipicamente in un computer moderno questo handler non viene richiamato mai, almeno per il traffico che è di interesse di *dummynet*;
- `ReceiveCompleteHandler`, richiamato in maniera asincrona dopo notifiche effettuate tramite `ReceiveHandler`, solo in presenza di pacchetti parziali, o da schede di rete particolarmente vecchie.

- `ReturnPacketHandler`, richiamato in maniera asincrona al completamento di un'operazione di *receive*, per effettuare il cleanup dei descrittori utilizzati.

3.3.2 Sequenza di operazioni per un pacchetto in uscita

In figura 3.4 viene mostrata una schematizzazione grafica del comportamento del driver e della sequenza delle operazioni nel caso di un pacchetto in uscita dal sistema.

Notifica da parte del protocollo

Quando il protocollo notifica la presenza di un pacchetto pronto per l'invio, il sistema *NDIS* richiama il `SendHandler`, che è stato assegnato alla funzione `MPSend()`. Il protocollo fornisce al sistema *NDIS* un pacchetto, nella struttura già descritta nella sezione 3.1. Sia il descrittore di pacchetto, sia il descrittore di buffer, sia le zone di memoria contenenti i dati restano di proprietà del protocollo sovrastante, e durante tutta l'operazione di *send* rimangono di proprietà del protocollo, non devono essere in alcun modo modificate, nè tantomeno liberate.

Operazioni del `SendHandler`

Il `SendHandler` deve produrre un nuovo descrittore di pacchetto. Esiste la possibilità, nelle versioni più recenti del *NDIS*, di riutilizzare il vecchio descrittore, chiamata *Packet stacks*, ma non è supportata dai driver delle schede di rete meno recenti, e non è facilmente adattabile alle esigenze di *dummynet*, per cui non viene utilizzata. Viene allocato un nuovo descrittore di pacchetto, e tramite apposite macro (ricordiamo che un pacchetto è un oggetto opaco e i suoi membri non vanno acceduti direttamente), viene agganciato alla catena di buffer forniti dal protocollo, vengono infine ricopiate le informazioni *Out Of Band*. L'*intermediate driver* sta per passare dei dati alla scheda di rete sottostante, quindi in questo momento si comporta come un *Protocol*, e può utilizzare la sezione riservata al protocollo per memorizzare delle informazioni, nella fattispecie viene memorizzato un puntatore al pacchetto originale, che servirà nel momento della chiamata alla routine di cleanup. Terminate queste operazioni, effettua la *send* vera e propria. Un *intermediate driver* è un driver di tipo *deserialized* per definizione, quindi l'operazione di *send* è sempre asincrona, e la routine di cleanup verrà chiamata sempre in maniera asincrona.

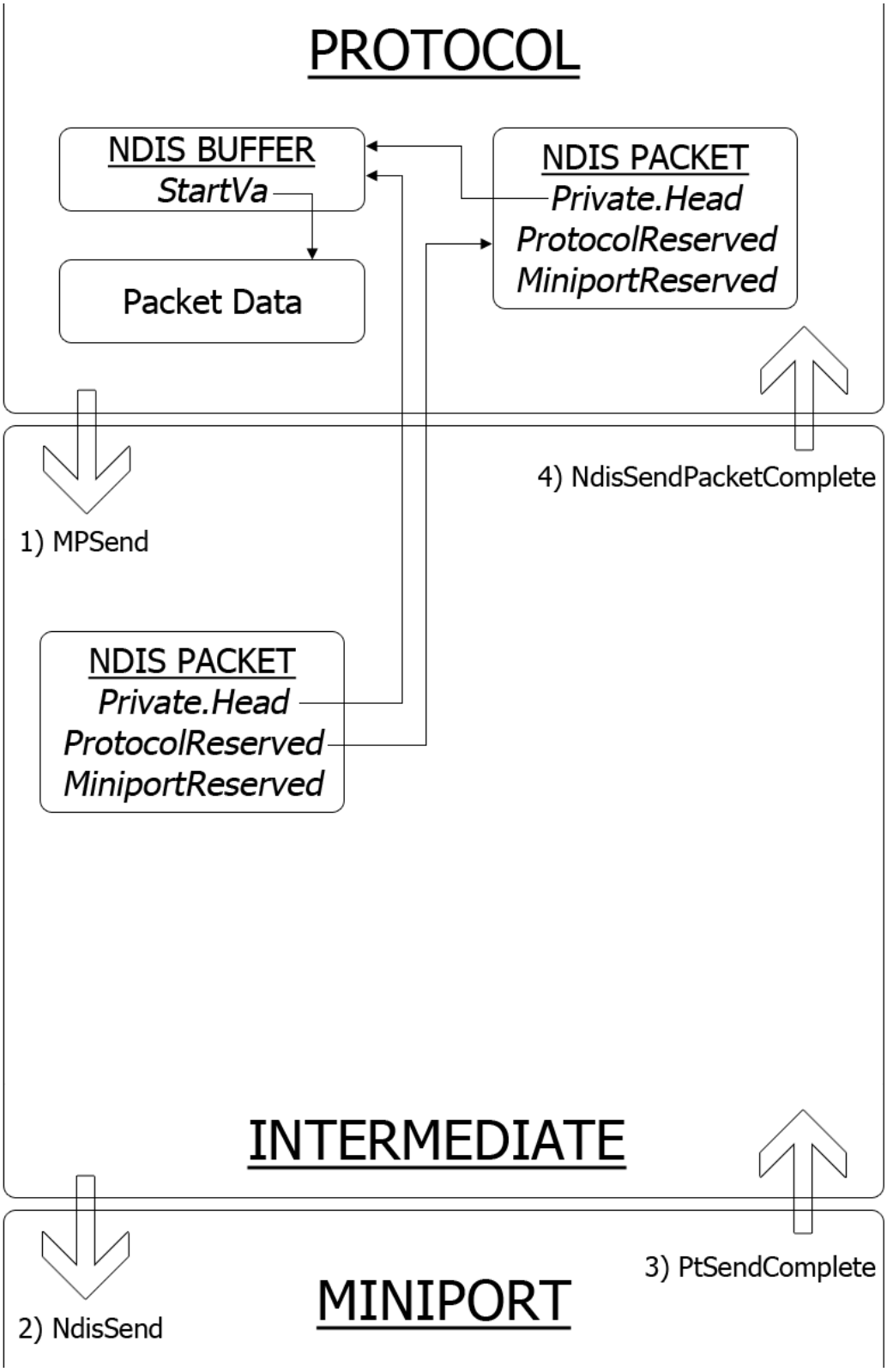

Figura 3.4: Il driver *Passthru*

Inoltre, nella struttura dati che rappresenta l'*adapter*, è presente un *ref count* dei pacchetti in uscita, il `SendHandler` si occupa di incrementare il contatore prima dell'effettiva *send* e di decrementarlo dopo, in modo da prevenire lo scaricamento del driver della scheda di rete fintanto che ci sono pacchetti pendenti.

Operazioni asincrone di cleanup

Quando la scheda di rete ha ricevuto i dati da inviare, viene richiamato dal sistema *NDIS* il `SendCompleteHandler`. Si noti che non c'è nessuna garanzia sul fatto che il pacchetto sia partito, nè che effettivamente partirà, tuttavia a questo punto le responsabilità dell'*intermediate driver* nei confronti del pacchetto sono terminate, e si può effettivamente procedere alla pulizia. Si può deallocare il descrittore di pacchetto creato nella `MPSend()`, recuperando il puntatore al descrittore originario che era stato memorizzato nella sezione riservata, e si può notificare al protocollo sovrastante che la *send* è terminata, in questo modo il protocollo potrà effettuare l'effettiva deallocazione dei suoi descrittori e della zona dati, poichè, ricordiamo nuovamente, questi oggetti restano di sua proprietà per tutta la durata della *send*.

3.3.3 Sequenza di operazioni per un pacchetto in ingresso

Notifica da parte della scheda di rete

Quando la scheda di rete notifica la presenza di dati in ingresso, il sistema *NDIS*, a seconda della tipologia di driver della scheda, della tipologia di dato ricevuto e della quantità di risorse di sistema disponibili, richiama il `ReceivePacketHandler` o il `ReceiveHandler`.

Operazioni del `ReceivePacketHandler`

Il `ReceivePacketHandler` opera in maniera duale rispetto al `SendHandler`, non utilizza il *Packet stacks*, alloca un nuovo descrittore di pacchetto, lo aggancia alla catena dei buffer forniti dal driver della scheda di rete, copia le informazioni *Out Of Band*, e notifica al protocollo sovrastante l'avvenuta ricezione. Questa volta l'*intermediate driver* comunica con un *Protocol*, per cui si comporta come un *Miniport*, ed utilizza la relativa zona riservata per memorizzare il puntatore al descrittore di pacchetto originario. Dopo la notifica, se la scheda di rete ha espresso urgenza di riottenere il descrittore, effettua la pulizia inline liberando il descrittore di pac-

chetto precedentemente allocato; come succedeva nella *send* infatti, il descrittore di pacchetto originario e le zone dati sono di proprietà del driver della scheda di rete.

Operazioni asincrone di cleanup

Quando il protocollo ha ricevuto correttamente i dati in ingresso, viene richiatato dal sistema *NDIS* il `ReturnPacketHandler`, che ripristina il puntatore al descrittore di pacchetto originario, effettua la liberazione del descrittore allocato dall'*intermediate driver*, e notifica alla scheda di rete che le operazioni di *receive* sono terminate.

Operazioni del `ReceiveHandler`

Quando viene richiamato il `ReceiveHandler`, siamo in presenza di una situazione anomala: driver particolarmente vecchio, carenza di risorse, dati incompleti, traffico non IP. Se c'è carenza di risorse da parte della scheda di rete, e quello che abbiamo ricevuto è effettivamente un pacchetto, la procedura è analoga al caso precedente: viene allocato un descrittore, copiate le informazioni aggiuntive, ed effettuata la notifica al protocollo. Essendo certi di essere in shortage, la pulizia viene effettuata inline. In caso contrario non è possibile creare un `NDIS_PACKET`, e la notifica avviene specificando il *medium*, cioè la tipologia della scheda di rete (Ethernet, Token Ring, FDDI), i puntatori all'header e al payload del pacchetto, e la lunghezza totale del pacchetto. Se la lunghezza totale del pacchetto è maggiore della lunghezza dell'header e del payload, viene richiamato automaticamente il `TransferDataHandler` che si occupa di ricopiare la parte mancante. Al termine del trasferimento dei dati, viene richiamato il `ReceiveCompleteHandler`, che notifica al protocollo che i dati sono completi e si può procedere alla loro analisi. In questo caso non c'è alcuna pulizia da effettuare, in quanto nessun descrittore di pacchetto è stato allocato dall'*intermediate driver*.

3.4 Il driver *Passthru* modificato

Esaminiamo ora come è stato modificato ed esteso il driver *Passthru* per realizzare la collaborazione con il *core* di *dummynet*.

3.4.1 Estensione della struttura `mbuf`

La `struct mbuf` è la struttura dati utilizzata da *dummynet* per memorizzare il contenuto dei pacchetti e le informazioni addizionali necessarie alla loro gestione. Il listato 3.2 mostra l'estensione della struttura su Windows.

Listato 3.2: Estensione della struttura `mbuf`

```
int             direction;
NDIS_HANDLE     context;
PNDIS_PACKET    pkt;
```

- `direction` viene utilizzato per discriminare la direzione di un pacchetto, in Windows le interfacce di rete sono oggetti piuttosto complessi, inoltre a questo livello dello stack sarebbe stato troppo oneroso associare i puntatori alle strutture dati che rappresentano gli *adapter* ai loro indirizzi IP, questo campo viene quindi usato come flag per ridurre l'overhead del modulo;
- `context` è un `HANDLE` della famiglia *NDIS* utilizzato per memorizzare il puntatore all'*adapter* fisico da cui il pacchetto proviene o verso cui è diretto;
- `pkt` è un puntatore ad un `NDIS_PACKET` utilizzato per memorizzare l'indirizzo del descrittore di pacchetto generato nell'*intermediate driver*.

3.4.2 Sequenza modificata di operazioni per un pacchetto in uscita

In figura 3.5 viene mostrata una schematizzazione di come è stata modificata la sequenza delle operazioni del driver *Passthru* nel caso di un pacchetto in uscita dal sistema, quando viene intercettato da *dummynet*.

- Il protocollo notifica la presenza di un pacchetto pronto per l'invio, viene creato il descrittore di pacchetto e i descrittori di buffer, viene allocata la memoria per i dati, e viene richiamata la `MPSend()`.
- L'*intermediate driver* compie tutte le operazioni necessarie alla creazione di un nuovo descrittore di pacchetto, ma prima di effettuare la *send*, viene richiamato il *queue handler*, il quale costruisce un `mbuf`, lo riempie con le informazioni necessarie, e ricopia i dati; si ribadisce che questa copia è necessaria, poichè la zona dati originale è di proprietà del protocollo e deve essere restituita

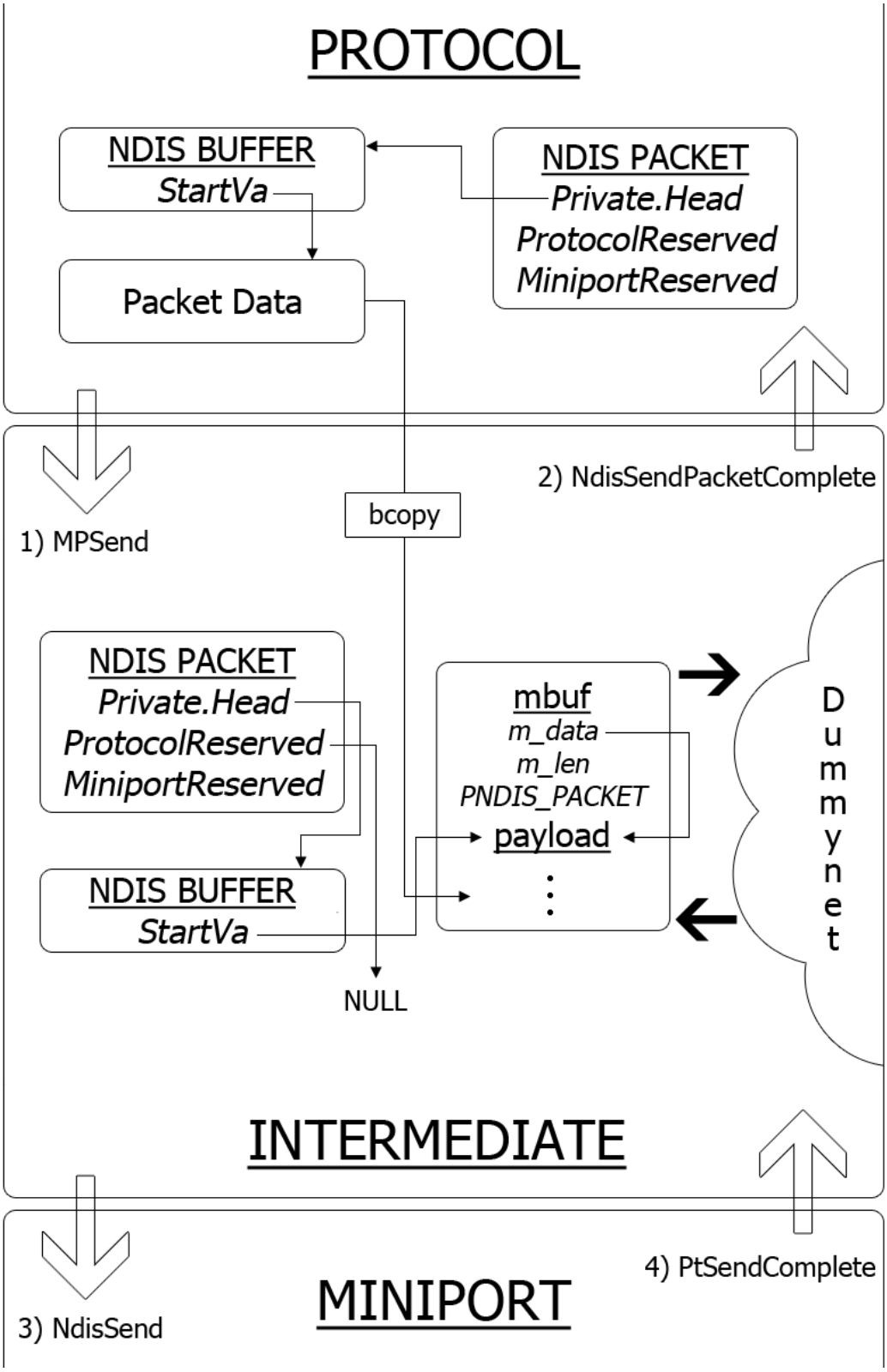

Figura 3.5: Il driver *Passthru* modificato

al protocollo per la deallocazione alla fine delle operazioni dell'*intermediate driver*.

- Il *queue handler* memorizza all'interno del `mbuf` l'indirizzo del descrittore di pacchetto allocato dall'*intermediate driver*, servirà per la corretta liberazione della memoria, quando il pacchetto sarà effettivamente partito.

- *Dummynet* notifica al *queue handler* che è interessato al pacchetto, viene quindi simulato un invio corretto, e si notifica al protocollo sovrastante che l'operazione di *send* è andata a buon fine; non viene decrementato il *ref count*.

- Dopo un certo numero di millisecondi, *dummynet* reimmette il pacchetto nella rete, restituendo il `mbuf` precedentemente salvato.

- Viene utilizzata la sezione riservata al *Protocol* per marcare il pacchetto come *reinjected*.

- Viene recuperato il descrittore di pacchetto, il cui indirizzo era salvato nel `mbuf`, viene ricostruito il descrittore di buffer e fatto puntare alla zona dati del `mbuf`, e con questi parametri viene effettuata la *send*.

- Si effettua la pulizia inline, se la scheda di rete lo richiede. Altrimenti, nell'esecuzione asincrona del `SendCompleteHandler` viene riconosciuto che il pacchetto è *reinjected*, vengono deallocate le zone di memoria relative, e si decrementa il *ref count*.

Le operazioni per un pacchetto in entrata sono assolutamente identiche, ad eccezione delle operazioni di cleanup finali. Essendo l'*intermediate driver* visto come un *Miniport* dal protocollo sovrastante, può simulare la carenza di risorse di una scheda di rete ed effettuare una notifica diretta senza descrittore, in questo modo il cleanup può essere fatto sempre inline.

3.4.3 Modifiche alle funzioni del driver *Passthru*

Modifiche alla `MPSend()`

Il listato 3.3 mostra la chiamata al *queue handler* durante la `MPSend()`, evidenziato dal blocco `#if`. Il codice originale crea il nuovo descrittore di pacchetto `MyPacket`, ed effettua le opportune operazioni di inizializzazione. La chiamata si

effettua subito prima dell'effettiva *send*, passando come parametri il descrittore appena creato, la direzione ed il puntatore all'*adapter*. I possibili valori di ritorno del *queue handler* sono:

- `PASS`: il firewall consente il passaggio del pacchetto, in questo caso si riprende ad eseguire il codice originale;
- `DROP`: il firewall non consente il passaggio del pacchetto, in questo caso si ritorna con un codice di errore, che consente al protocollo di capire che il pacchetto è stato rifiutato;
- `DUMMYNET`: *dummynet* ha conservato il pacchetto nelle sue code e lo reimmetterà in seguito, in questo caso si simula un invio andato a buon fine e si ritorna subito.

Listato 3.3: Modifica della `MPSend()`

```
#if 1 /* IPFW: query the firewall */
          /* if dummynet keeps the packet, we mimic success.
           * otherwise continue as usual.
           */
        {
                int ret = ipfw2_qhandler_w32(MyPacket, OUTGOING,
                                MiniportAdapterContext);
                if (ret != PASS) {
                        if (ret == DROP)
                                return NDIS_STATUS_FAILURE;
                        else {  //dummynet kept the packet
                                return NDIS_STATUS_SUCCESS;
                        }
                } //otherwise simply continue
        }
#endif  /* end of IPFW code */
```

Modifiche alla `PtReceivePacket()`

Il listato 3.4 mostra la chiamata al *queue handler*, del tutto analoga alla precedente, durante la `PtReceivePacket()`. Si noti che qui non c'è distinzione tra il caso `DROP` e il caso `DUMMYNET`, in quanto non ha senso notificare alla scheda di rete che il pacchetto è stato droppato.

Listato 3.4: Modifica della `PtReceivePacket()`

```
if (pAdapt->MiniportHandle != NULL)
{
#if 1   /* IPFW: query the firewall */
    int ret;
    ret = ipfw2_qhandler_w32(MyPacket, INCOMING,
          ProtocolBindingContext);
    if (ret != PASS)
        return 0; //otherwise simply continue
#endif /* end of IPFW code */
    NdisMIndicateReceivePacket(pAdapt->MiniportHandle, &MyPacket, 1);
}
```

Modifiche alla `PtReceive()`

La funzione si divide in due rami, il primo viene percorso in caso di shortage di risorse da parte della scheda di rete, in questo caso si procede in maniera analoga alla `PtReceivePacket()`; nel secondo ramo ci troviamo con una segnalazione *old-style* da parte di un driver non recente, e si chiama una versione modificata del *queue handler*, che effettua le stesse operazioni ma lavorando su buffer di header e payload, piuttosto che su `NDIS_PACKET`.

Listato 3.5: Modifica della `PtReceive()`

```
    case NdisMediumWan:
#if 1   /* IPFW: query the firewall */
    {
        int ret = ipfw2_qhandler_w32_oldstyle(INCOMING, ProtocolBindingContext,
            HeaderBuffer, HeaderBufferSize,
            LookAheadBuffer, LookAheadBufferSize,
            PacketSize);
        if (ret != PASS)
            return NDIS_STATUS_SUCCESS;
    }
#endif /* end of IPFW code */
        NdisMEthIndicateReceive(pAdapt->MiniportHandle,
                    MacReceiveContext,
                    HeaderBuffer,
                    HeaderBufferSize,
                    LookAheadBuffer,
                    LookAheadBufferSize,
                    PacketSize);
    break;
```

Modifiche alla `PtSendComplete()`

All'inizio della `PtSendComplete()`, viene esaminata la parte riservata del descrittore, e se si identifica il pacchetto come *reinjected*, viene richiamata l'apposita funzione per liberarne la memoria, e si ritorna immediatamente.

Listato 3.6: Modifica della `PtSendComplete()`

```
    PSEND_RSVD SendRsvd;
    SendRsvd = (PSEND_RSVD)(Packet->ProtocolReserved);
    Pkt = SendRsvd->OriginalPkt;

#if 1   // IPFW
    if (Pkt == NULL) { //this is a reinjected packet, with no 'father'
        CleanupReinjected(Packet, SendRsvd->pMbuf, pAdapt);
        return;
    }
#endif /* end of IPFW code */
```

Modifiche alle sezioni riservate del `NDIS_PACKET`

L'*intermediate driver* si comporta come un *Protocol* quando inoltra alla scheda di rete pacchetti in uscita, e come un *Miniport* quando inoltra al protocollo IP pacchetti in ingresso. Ha quindi a disposizione le due sezioni riservate del `NDIS_PACKET`, ma ne può usare solo una per volta, a seconda della direzione. La documentazione[11] assicura ai *Miniport* una zona di memoria di dimensione fissa pari a 8 byte su un sistema a 32 bit, e 16 byte su uno a 64 bit, mentre per i *Protocol* uno spazio rispettivamente di 16 o 32 byte. Le zone riservate sono ridefinibili a piacere da un *intermediate driver*, purchè non si superi la dimensione consentita, nella nostra implementazione abbiamo aggiunto un secondo puntatore a `mbuf`, utilizzato dalle routine di cleanup.

Listato 3.7: Modifiche alle sezioni riservate del `NDIS_PACKET`

```
typedef struct _SEND_RSVD
{
    PNDIS_PACKET    OriginalPkt;
    struct mbuf*    pMbuf; // IPFW extension, reference to the mbuf
} SEND_RSVD, *PSEND_RSVD;

typedef struct _RECV_RSVD
{
    PNDIS_PACKET    OriginalPkt;
    struct mbuf*    pMbuf; // IPFW extension, reference to the mbuf
} RECV_RSVD, *PRECV_RSVD;
```

La funzione `MPReturnPacket()` invece non viene modificata in alcun modo, poichè, come si è già visto, il cleanup per i pacchetti in entrata viene sempre effettuato inline.

3.4.4 Il *queue handler*

Il *queue handler* è la funzione dello strato di compatibilità che si occupa di trasformare un `NDIS_PACKET` in un `mbuf`, e interrogare *dummynet* su cosa fare del pacchetto. Di seguito viene mostrata e descritta l'implementazione della funzione.

Listato 3.8: *queue handler*: operazioni iniziali

```
static char _if_in[] = "incoming";
static char _if_out[] = "outgoing";

int ipfw2_qhandler_w32(PNDIS_PACKET pNdisPacket, int direction,
        NDIS_HANDLE Context)
{
        unsigned int            BufferCount = 0;
        unsigned                TotalPacketLength = 0;
        PNDIS_BUFFER            pCurrentBuffer = NULL;
        PNDIS_BUFFER            pNextBuffer = NULL;
        struct mbuf*            m;
        unsigned char*          payload = NULL;
        unsigned int            ofs, l;
        unsigned short          EtherType = 0;
        unsigned int            i = 0;
        int                     ret = 0;
        PNDIS_BUFFER            pNdisBuffer, old_head, old_tail;
        NDIS_HANDLE             PacketPool;
        PADAPT                  pAdapt;
        NDIS_STATUS             Status;

        NdisQueryPacket(pNdisPacket, NULL, &BufferCount,
                &pCurrentBuffer, &TotalPacketLength);
        m = malloc(sizeof(struct mbuf) + TotalPacketLength, 0, 0 );
        if (m == NULL) //resource shortage, drop the packet
                goto drop_pkt;

        payload = (unsigned char*)(m + 1);
        m->m_len = m->m_pkthdr.len = TotalPacketLength-14;
        m->m_pkthdr.rcvif = (void *)((direction==INCOMING) ? _if_in : NULL);
        m->m_data = payload + 14; /* past the MAC header */
        m->direction = direction;
        m->context = Context;
        m->pkt = pNdisPacket;
```

Dummynet utilizza puntatori all'interfaccia tramite cui transita il pacchetto per differenziare il traffico in entrata da quello in uscita, in Windows questa operazione

sarebbe stata troppo onerosa, vengono quindi definite due stringhe statiche ed il loro indirizzo viene usato per emulare due interfacce virtuali "incoming" ed "outgoing".

Nel corpo della funzione, si interroga il pacchetto per conoscerne la dimensione totale ed il numero dei buffer di cui è composto. Si alloca quindi la memoria sufficiente a contenere un `mbuf` e l'intero pacchetto linearizzato. Viene quindi riempito il `mbuf` con:

- il puntatore all'inizio del pacchetto;
- l'indirizzo dell'interfaccia virtuale se il pacchetto è in entrata;
- il puntatore all'inizio dell'header layer 3;
- il flag che differenzia la direzione del pacchetto; questa è effettivamente duplicazione di un'informazione già presente, tuttavia, come già spiegato nella sezione 3.4.1, il campo `direction` viene utilizzato all'interno dello strato di compatibilità per semplificare le operazioni dello stesso, mentre *dummynet* continua ad utilizzare altri campi per la classificazione interna dei pacchetti;
- il puntatore all'*adapter*, che nella terminologia di Windows assume il nome di *contesto*;
- il puntatore al `NDIS_PACKET` generato nell'*intermediate driver*.

Listato 3.9: *queue handler*: linearizzazione del pacchetto

```
for (i=0, ofs = 0; i < BufferCount; i++) {
        unsigned char* src;
        NdisQueryBufferSafe(pCurrentBuffer, &src, &len,
                NormalPagePriority);
        bcopy(src, payload + ofs, len);
        ofs += len;
        NdisGetNextBuffer(pCurrentBuffer, &pNextBuffer);
        pCurrentBuffer = pNextBuffer;
}
```

Viene quindi effettuata la copia del pacchetto in una zona di memoria contigua, ed adiacente al `mbuf`. Tipicamente un pacchetto generato localmente è distribuito su quattro buffer distinti, uno per l'header MAC, uno per l'header IP, uno per l'header layer 4, ed uno per il payload.

Listato 3.10: *queue handler*: ispezione del protocollo layer 3

```
EtherType = *(unsigned short*)(payload + 12);
EtherType = RtlUshortByteSwap(EtherType);
if (EtherType != 0x0800) {
        free(m, 0);
        return PASS;
}
```

Si ispezionano gli ultimi due bytes dell'header MAC, se non risulta un pacchetto IP, viene fatto semplicemente passare, poichè *dummynet* al momento gestisce solo traffico IP.

Listato 3.11: *queue handler*: allocazione del `NDIS_BUFFER`

```
pAdapt = Context;
PacketPool = direction == OUTGOING ?
        pAdapt->SendPacketPoolHandle : pAdapt->RecvPacketPoolHandle;
NdisAllocateBuffer(&Status, &pNdisBuffer,
        PacketPool, payload, m->m_pkthdr.len+14);
if (Status != NDIS_STATUS_SUCCESS)
        goto drop_pkt;

pNdisBuffer->Next = NULL;
old_head = NDIS_PACKET_FIRST_NDIS_BUFFER(pNdisPacket);
old_tail = NDIS_PACKET_LAST_NDIS_BUFFER(pNdisPacket);
NdisReinitializePacket(pNdisPacket);
NdisChainBufferAtFront(pNdisPacket, pNdisBuffer);
```

Viene allocato un descrittore di buffer dall'opportuno *pool* di risorse, ogni scheda di rete ha infatti due *pool* distinti, uno per l'ingresso e uno per l'uscita. Come è stato già sottolineato, anche il descrittore di buffer appartiene al protocollo sovrastante o alla scheda di rete sottostante, per i pacchetti in transito può essere riutilizzato indipendentemente dall'utilizzo o meno del *Packet stacks*, ma per i pacchetti trattenuti da *dummynet* verrà deallocato dal suo proprietario nel momento in cui simuliamo di aver effettuato il trasferimento, bisogna quindi creare una copia locale anche del buffer da utilizzare al momento del *reinject*.

Listato 3.12: *queue handler*: interrogazione del firewall

```
if (direction == INCOMING)
        ret = ipfw_check_hook(NULL, &m, NULL, PFIL_IN, NULL);
else
        ret = ipfw_check_hook(NULL, &m, (struct ifnet*)_if_out,
                              PFIL_OUT, NULL);

if (m != NULL) {
        NdisReinitializePacket(pNdisPacket);
        NDIS_PACKET_FIRST_NDIS_BUFFER(pNdisPacket) = old_head;
        NDIS_PACKET_LAST_NDIS_BUFFER(pNdisPacket) = old_tail;
```

```
                NdisFreeBuffer(pNdisBuffer);
                m_freem(m);
                return PASS;
        } else if (ret == 0) {
                return DUMMYNET;
        } else {
                return DROP;
        }
drop_pkt:
        NdisFreePacket(pNdisPacket);
        return DROP;
}
```

Finite le operazioni iniziali si può effettivamente interrogare il firewall. Si noti che il traffico in ingresso viene contrassegnato dalla presenza del puntatore all'interfaccia virtuale nel `mbuf`, quello in uscita dal terzo parametro della funzione `ipfw_check_hook()`. Se il firewall lascia passare il pacchetto, il puntatore al `mbuf` resta valido, e si procede al ripristino della chain di `NDIS_BUFFER` originaria, e alla liberazione delle strutture dati temporanee create durante la procedura. Altrimenti *dummynet* ha conservato il pacchetto, oppure ha deciso di dropparlo e ha già provveduto alla liberazione della memoria, si restituisce quindi il relativo valore di ritorno.

3.4.5 Il *queue handler* alternativo

Per i pacchetti in ingresso provenienti dalla `PtReceive()`, viene richiamata una versione modificata del *queue handler*, che gestisce non il normale `NDIS_PACKET`, ma un pacchetto distribuito sempre su due buffer, uno contenente l'header MAC, e l'altro il resto dei dati. Nel listato 3.13 viene mostrata la differenza di parametri formali e di codice.

Listato 3.13: Versione modificata del *queue handler*

```
int ipfw2_qhandler_w32_oldstyle(int direction,
        NDIS_HANDLE         ProtocolBindingContext,
        unsigned char*      HeaderBuffer,
        unsigned int        HeaderBufferSize,
        unsigned char*      LookAheadBuffer,
        unsigned int        LookAheadBufferSize,
        unsigned int        PacketSize)
{

        /*
         * Stesse operazioni del normale queue handler
         */
```

```
        bcopy(HeaderBuffer, payload, HeaderBufferSize);
        bcopy(LookAheadBuffer, payload+HeaderBufferSize, LookAheadBufferSize);
}
```

3.4.6 Il *reinject* dei pacchetti

Dummynet utilizza due funzioni per reimmettere i pacchetti nella rete, una per quelli in ingresso ed una per quelli in uscita. Vengono entrambe rimappate sulla funzione `netisr_dispatch()`, la direzione viene ricavata dal `mbuf`.

Listato 3.14: *reinject*: controllo dello stato del *Protocol* e del *Miniport*

```
void netisr_dispatch(int num, struct mbuf *m)
{
        unsigned char*  payload = (unsigned char*)(m+1);
        PADAPT          pAdapt = m->context;
        NDIS_STATUS     Status;
        PNDIS_PACKET    pPacket = m->pkt;
        PNDIS_BUFFER    pNdisBuffer;
        NDIS_HANDLE     PacketPool;

        NdisAcquireSpinLock(&pAdapt->Lock);
        if (m->direction == OUTGOING) {
                if (pAdapt->PTDeviceState > NdisDeviceStateD0) {
                        pAdapt->OutstandingSends--;
                        NdisReleaseSpinLock(&pAdapt->Lock);
                        goto drop_pkt;
                }
        } else {
                if (!pAdapt->MiniportHandle ||
                    pAdapt->MPDeviceState > NdisDeviceStateD0) {
                        NdisReleaseSpinLock(&pAdapt->Lock);
                        goto drop_pkt;
                }
        }
        NdisReleaseSpinLock(&pAdapt->Lock);
```

Il listato 3.14 mostra le operazioni iniziali della procedura, con le quali si controlla lo stato del protocollo sovrastante o della scheda di rete sottostante, infatti dall'immissione del pacchetto nella coda di *dummynet* al suo *reinject* l'hardware può essere entrato in power saving, o il sistema operativo può aver iniziato la procedura di shutdown. Prima di effettuare l'inoltro bisogna quindi eccettarsi che il destinatario del pacchetto sia in condizione di riceverlo, controllando sotto lock gli opportuni campi della struttura `ADAPT`, relativa all'istanza di *Miniport* virtuale in cui ci si trova. Se il destinatario non è pronto, si abortisce, decrementando anche il *ref count* nel caso di un pacchetto uscente.

Listato 3.15: *reinject*: invio del pacchetto

```
if (m->direction == OUTGOING) {
                PSEND_RSVD      SendRsvd;
                SendRsvd = (PSEND_RSVD)(pPacket->ProtocolReserved);
                SendRsvd->OriginalPkt = NULL;
                SendRsvd->pMbuf = m;

                NdisSend(&Status, pAdapt->BindingHandle, pPacket);
                if (Status != NDIS_STATUS_PENDING) {
                        PtSendComplete(m->context, m->pkt, Status);
                }
                return;
        } else {
                ULONG Proc = KeGetCurrentProcessorNumber();
                pAdapt->ReceivedIndicationFlags[Proc] = TRUE;
                NdisMEthIndicateReceive(pAdapt->MiniportHandle,
                        NULL, payload, 14, payload+14, m->m_len, m->m_len);
                NdisMEthIndicateReceiveComplete(pAdapt->MiniportHandle);
                pAdapt->ReceivedIndicationFlags[Proc] = FALSE;
        }
drop_pkt:
        if (m->pkt != NULL)
        {
                NdisUnchainBufferAtFront(m->pkt, &pNdisBuffer);
                NdisFreeBuffer(pNdisBuffer);
                NdisFreePacket(m->pkt);
        }
        m_freem(m);
}
```

Segue quindi, come mostrato nel listato 3.15, l'invio vero e proprio del pacchetto.

- Per un pacchetto uscente, in cui il cleanup sarà eseguito asincronamente, si utilizza la sezione riservata al *Protocol* per marcare il pacchetto come "orfano", impostando a `NULL` il puntatore al pacchetto originale, e per salvare l'indirizzo del `mbuf`, anche questo verrà deallocato asincronamente.

- Per un pacchetto entrante, si notifica semplicemente al protocollo l'avvenuta ricezione. Si utilizza la funzione `NdisMEthIndicateReceive()` passando direttamente il payload del `mbuf` piuttosto che il pacchetto completo, per aggirare un bug presente nel *Passthru* originale, che non ne consentiva il corretto funzionamento con alcune schede di rete.

3.4.7 La funzione di cleanup dei pacchetti *reinjected*

Il cleanup delle strutture dati create nell'*intermediate driver* viene effettuato, come si è visto, sempre inline per i pacchetti in ingresso, e sempre in maniera asin-

crona per i pacchetti in uscita. Il listato 3.16 mostra la funzione di cleanup asincrona richiamata dal `SendCompleteHandler` relativamente ad un pacchetto reimmesso. Si interroga il pacchetto per recuperare il puntatore al buffer, si sgancia il buffer dal pacchetto, si dealloca il buffer, il pacchetto, e il `mbuf` con la funzione `win_freem()`, che è un semplice wrapping all'analoga funzione FreeBSD che si occupa di liberare non solo il `mbuf` ma anche tutte le zone di memoria dinamica ad esso collegate.

Listato 3.16: Cleanup asincrono

```
void CleanupReinjected(PNDIS_PACKET Packet, struct mbuf* m, PADAPT pAdapt)
{
        PNDIS_BUFFER pNdisBuffer;
        NdisQueryPacket(Packet, NULL, NULL, &pNdisBuffer, NULL);
        NdisUnchainBufferAtFront(Packet, &pNdisBuffer);
        NdisFreeBuffer(pNdisBuffer);
        win_freem(m);
        NdisFreePacket(Packet);
        ADAPT_DECR_PENDING_SENDS(pAdapt);
}
```

Capitolo 4

Timer e letture del system time

La ragione per cui le strutture dati e gli algoritmi possono funzionare assieme senza problemi è che non sanno nulla gli uni degli altri

A. Stepanov

4.1 Le funzioni della famiglia `callout`

Lo schedulatore interno di *Dummynet* è una funzione che viene richiamata ogni millisecondo, mentre un secondo timer viene utilizzato per la gestione delle regole dinamiche, a granularità di un secondo. Vengono utilizzate la struttura dati e le funzioni della famiglia `callout`, che non prevedono la possibilità di inizializzare un timer ricorrente, per questo motivo nel codice originale, durante il caricamento del modulo viene creato un timer *one-shot* che richiama la funzione di scheduling, ed è cura di questa stessa, alla fine della sua esecuzione, reinserire il timer nella coda di sistema.

In Windows i timer ricorrenti sono disponibili[13], tuttavia, nell'ottica di mantenere il codice quanto più possibile vicino all'originale, piuttosto che escludere le `callout` con delle direttive di preprocess e settare manualmente un timer ricorrente durante l'esecuzione del driver, abbiamo preferito implementare le `callout` in modo che fossero compatibili e completamente trasparenti per il codice originale.

Viene quindi ridefinita la `struct callout` come mostrato nel listato 4.1. Si utilizza un `KTIMER`, il timer generico di più basso livello tra quelli forniti dal kernel, un `KDPC`, un oggetto opaco che è collegato alla *Deferred Procedure Call* che verrà eseguita allo scadere del timer, una flag `dpcinitialized` che viene impiegata per discriminare la prima chiamata del timer in cui va inizializzato l'oggetto `KDPC` dalle seguenti, e un intero a 64 bit per contenere il valore del timer espresso in *system time unit* da 100 nanosecondi.

Listato 4.1: La `struct callout`

```
struct callout {
        KTIMER thetimer;
        KDPC timerdpc;
        int dpcinitialized;
        LARGE_INTEGER duetime;
};
```

Vengono quindi definite le *DPC* legate ai due timer come mostrato nel listato 4.2. Oltre al puntatore all'oggetto `KDPC`, la funzione riceve dei generici puntatori a void, che nel kernel di Windows vengono spesso utilizzati associandoli al concetto di *contesto*. Questi puntatori forniscono alla funzione chiamata una serie di parametri attuali, di cui però solo il primo può essere fornito in fase di inizializzazione della *DPC*, gli altri due vengono passati dalla funzione che inserisce la *DPC* nella coda di esecuzione del processore; nel caso di una *CustomTimerDPC* l'inserimento nella coda viene effettuato automaticamente dalle funzioni timer, non è quindi possibile utilizzarli. Il corpo della *DPC* è, banalmente, una chiamata alla rispettiva funzione di *dummynet* associata al timer.

Listato 4.2: Le *Deferred Precedure Calls*

```
VOID dummynet_dpc(
    __in struct _KDPC  *Dpc, __in_opt PVOID  DeferredContext,
    __in_opt PVOID  SystemArgument1, __in_opt PVOID  SystemArgument2
    )
{
        dummynet(NULL);
}

VOID ipfw_dpc(
    __in struct _KDPC  *Dpc, __in_opt PVOID  DeferredContext,
    __in_opt PVOID  SystemArgument1, __in_opt PVOID  SystemArgument2
    )
{
        ipfw_tick(DeferredContext);
}
```

Il listato 4.3 mostra invece l'implementazione delle quattro funzioni relative ai timer utilizzate da *dummynet*. Esattamente come nelle `callout` originali, la `callout_init()` si limita a inizializzare l'oggetto di tipo `KTIMER`; le due funzioni di stop, che in FreeBSD hanno una semantica leggermente diversa, vengono entrambe rimappate sulla `callout_drain()`, che è la versione bloccante, ed è l'unica che garantisca, tramite un meccanismo di sincronizzazione interno, che il timer sia effettivamente fermo e che la funzione chiamata abbia terminato la sua esecuzione; questa precauzione è necessaria poichè `dummynet()` provvede a rischedularsi alla fine delle sue operazioni, e quindi potrebbe rientrare nella coda della cpu anche dopo il ritorno della `callout_stop()`, causando potenziali deadlock o accessi alla memoria non validi. L'implementazione quindi richiama ciclicamente la `KeCancelTimer()`, che ritorna `TRUE` solo quando riesce a fermare il timer prima che la *DPC* venga messa in coda.

La funzione di reset si comporta in maniera differente a seconda del timer a cui viene associata, poichè i due timer hanno esigenze diverse. Se viene chiamata per il timer di `dummynet()`, inizializza la *DPC* alla prima invocazione, e solo alla prima, come è normale fare; inoltre tramite la funzione `KeSetTargetProcessorDpc()` costringe il thread che eseguirà lo schedulatore a girare sulla prima cpu del sistema (che è sempre presente, quindi questo codice è valido tanto per le piattaforme mono core, che per le multi core), poichè, come spiegato nella sezione 4.3, il calcolo dell'ora di sistema viene effettuata andando a leggere il valore del *performance counter* di una specifica cpu, e senza questa restrizione si potrebbero avere letture che "saltano avanti e indietro nel tempo".

Se viene chiamata invece per il timer di `ipfw_tick()`, provvede a reinizializzare ogni volta la *DPC*, utilizzando quindi il puntatore a contesto per passare l'argomento al chiamato. La funzione `KeInitializeDpc()` viene utilizzata per costringere il programmatore a utilizzare il `KDPC` come oggetto opaco, ma di fatto si limita a fare un assegnamento, per cui richiamarla più del necessario non è un problema dal punto di vista prestazionale, tenendo anche conto che la granularità del relativo timer è piuttosto bassa.

Il valore del timeout deve essere passato attraverso un `LARGE_INTEGER`, che è una struttura fornita per retrocompatibilità con i compilatori che non sanno gestire nativamente gli interi a 64 bit, per cui dobbiamo convivere con lo spreco di spazio nella `struct callout` e di tempo nell'assegnamento. Il valore è espresso, come si è detto, in *system time unit* da 100 nanosecondi, quindi viene effettuata la conversione

dal valore originale in millisecondi, e viene cambiato il segno: un tempo positivo viene interpretato come assoluto, e quindi è suscettibile ai cambiamenti di orario (che possono avvenire anche senza l'intervento umano in presenza di sincronizzazioni del clock tramite *NTP* o meccanismi simili), un tempo negativo è invece relativo al momento del set, e quindi preferibile.

Listato 4.3: Le funzioni `callout`

```
static __inline int
callout_reset(struct callout *co, int ticks, void (*fn)(void *), void *arg)
{
        if(fn == &dummynet) {
                if(co->dpcinitialized == 0) {
                        KeInitializeDpc(&co->timerdpc, dummynet_dpc, NULL);
                        KeSetTargetProcessorDpc(&co->timerdpc, 0);
                        co->dpcinitialized = 1;
                }
        } else {
                KeInitializeDpc(&co->timerdpc, ipfw_dpc, arg);
        }
        co->duetime.QuadPart = (-ticks)*10000;
        KeSetTimer(&co->thetimer, co->duetime, &co->timerdpc);
        return 0;
}

static __inline void
callout_init(struct callout* co, int safe) {
        printf("%s: initializing timer at %p\n",__FUNCTION__,co);
        KeInitializeTimer(&co->thetimer);
}

static __inline int
callout_drain(struct callout* co) {
        BOOLEAN canceled = KeCancelTimer(&co->thetimer);
        while (canceled != TRUE) {
                canceled = KeCancelTimer(&co->thetimer);
        }
        printf("%s: stopping timer at %p\n",__FUNCTION__,co);
        return 0;
}

static __inline int
callout_stop(struct callout* co) {
        return callout_drain(co);
}
```

4.2 Lo schedulatore di Windows

Indipendentemente dal valore richiesto nella `KeSetTimer()`, lo schedulatore di Windows va in esecuzione tipicamente ogni 10-15 millisecondi circa (è *platform-dependant*), è quindi impossibile, utilizzando le impostazioni di default del sistema operativo, ottenere una granularità superiore a questo valore sia per quanto riguarda la precisione del system time, sia per quanto riguarda la frequenza con cui viene richiamato l'algoritmo di schedulazione di *dummynet*. Lo schedulatore interno di *dummynet* è strutturato in maniera tale da essere abbastanza indipendente dalla periodicità con cui viene richiamato, è infatti in grado di calcolare il discostamento dalla frequenza ideale di 1KHz, tenendo conto anche della somma dei residui generati dall'imprecisione dei timer, e di compensare. Tuttavia bisognerebbe accettare una imprecisione sulla reimmissione dei pacchetti, che può arrivare fino ad un massimo di un interrupt timer.

Fortunatamente esiste una funzione del kernel, `ExSetTimerResolution()`, che consente di modificare il valore dell'interrupt timer e quindi dello schedulatore di sistema. Il valore minimo impostabile su una cpu moderna è circa 1 millisecondo, il che rende i timer sufficientemente precisi per i nostri scopi.

4.3 Il system time

Lo schedulatore interno di *dummynet* si appoggia al system time per effettuare i calcoli, è stata quindi definita la struttura `timeval` e implementata la funzione `do_gettimeofday()` affinchè utilizzi la precisione al microsecondo come nella versione POSIX.

La funzione kernel equivalente in Windows per acquisire il time of the day, `KeQuerySystemTime()`, resituisce un valore con una granularità ai 100-nsec, molto fine, tuttavia il valore viene aggiornato ogni 10-15 millisecondi, indipendentemente dall'aver ridotto o meno il valore dell'interrupt timer. Effettuando la chiamata ad ogni schedulazione di *dummynet*, quindi ogni millisecondo, avremmo ottenuto per 10-15 volte lo stesso valore, come se il tempo fosse fermo, e questo avrebbe creato ancora più imprecisione. È stato quindi necessario studiare una soluzione diversa.

Un possibile workaround sarebbe stato quella di implementare direttamente in assembler una routine come quella mostrata nel listato 4.4, ma sarebbe stato rischio-

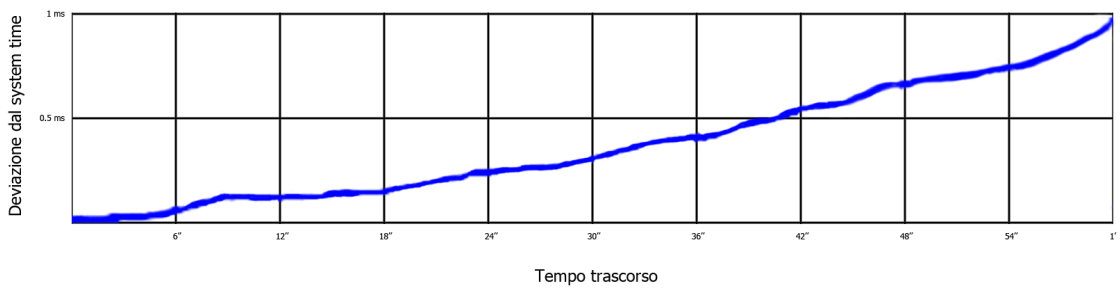

Figura 4.1: Deviazione dal sytem time

so e poco portabile. Inoltre nelle cpu moderne, che consentono l'adeguamento della frequenza al carico attuale, si sarebbero potute avere letture errate.

Listato 4.4: Una possibile implementazione in assembler

```
__asm {
        cpuid           ; forza il flush della pipeline
        rdtsc           ; legge time stamp counter
        mov time, eax   ; scrive il valore nel registro di ritorno eax
}
```

Per gli applicativi che necessitano di misurare intervalli di tempo più piccoli di un interrupt time non modificato, la documentazione della Microsoft suggerisce di utilizzare i *performance counter*, una serie di funzioni di basso livello che consentono di ottenere temporizzazioni legate al clock del processore. Si deve sottolineare che queste funzioni sono basate sull'*Hardware Abstraction Layer*, e quindi la loro implementazione varia a seconda del sistema sul quale vengono eseguite, tuttavia tengono conto della possibile variazione di frequenza, sono portabili, e proprio perchè si affidano all'*HAL* garantiscono di ottenere i valori più precisi possibile relativamente alla macchina su cui viene eseguito il codice.

Resta il problema però che la frequenza, anche quando non viene cambiata volutamente dai meccanismi per il risparmio energetico, è un dato il cui valore nominale si discosta sempre da quello reale. Come si può vedere dalla figura 4.1, affidarsi esclusivamente ad un *performance counter* per la misurazione dei tempi assoluti porta ad un discostamento progressivo del tempo misurato rispetto al tempo di sistema. L'entità del discostamento dipende ovviamente dalla frequenza, e sulla nostra macchina di prova è stato quantificato in circa 1 ms ogni minuto.

L'idea quindi è di effettuare una risincronizzazione periodica ad intervalli di tempo prestabiliti. Ma anche in questo caso ci sono due inconvenienti: l'impossibilità di decidere a priori un intervallo di tempo ottimale dopo cui effettuare il resynch, e

l'impossibilità di effettuare una misurazione assoluta precisa al momento del resynch. Infatti se andassimo a leggere il system time una volta ogni minuto, non potremmo mai sapere in quale punto del segmento temporale di 10-15 ms tra due aggiornamenti ci troviamo.

La nostra implementazione effettua il resynch ad ogni cambiamento del system time, come mostrato nel listato 4.5. Anche se piuttosto oneroso dal punto di vista delle commutazioni di contesto, questa soluzione ci consente di:

- non discostarci mai dal system time reale, di una quantità (sulla macchina di test) superiore a 1/60 di ms, cioè 17 microsecondi;
- di effettuare misurazioni precise all'interno del segmento temporale tra due aggiornamenti del system time;
- di ottenere un errore alla risincronizzazione inferiore al millisecondo.

Listato 4.5: Implementazione di `do_gettimeofday()`

```
void
do_gettimeofday(struct timeval *tv)
{
        static LARGE_INTEGER prevtime; //system time in 100-nsec resolution
        static LARGE_INTEGER prevcount; //RTC counter value
        static LARGE_INTEGER freq; //frequency

        LARGE_INTEGER currtime;
        LARGE_INTEGER currcount;
        if (prevtime.QuadPart == 0) { //first time we ask for system time
                KeQuerySystemTime(&prevtime);
                prevcount = KeQueryPerformanceCounter(&freq);
                currtime.QuadPart = prevtime.QuadPart;
        } else {
                KeQuerySystemTime(&currtime);
                currcount = KeQueryPerformanceCounter(&freq);
                if (currtime.QuadPart == prevtime.QuadPart) {
                        //time HAS NOT changed, calculate time
                        //using ticks and DO NOT update
                        LONGLONG difftime = 0; //difference in 100-nsec
                        LONGLONG diffcount = 0; //clock count difference
                        diffcount = currcount.QuadPart - prevcount.QuadPart;
                        diffcount *= 10000000;
                        difftime = diffcount / freq.QuadPart;
                        currtime.QuadPart += difftime;
                } else {
                        //time HAS changed, update and return SystemTime
                        prevtime.QuadPart = currtime.QuadPart;
                        prevcount.QuadPart = currcount.QuadPart;
                }
```

```
        }
        currtime.QuadPart /= 10; //convert in usec
        tv->tv_sec = currtime.QuadPart / (LONGLONG)1000000;
        tv->tv_usec = currtime.QuadPart % (LONGLONG)1000000;
}
```

Capitolo 5

Semafori, gestione della memoria dinamica e funzioni della *Run Time Library*

Ci si aspetta che per fare 'n+1' piccoli lavori
venga impiegato lo stesso tempo che per farne 'n'

Paradosso di Gray sulla programmazione

5.1 Semafori e sincronizzazione

In Windows esistono due tiplogie di semafori utilizzabili nel kernel per le sincronizzazioni: gli *spin lock*[8] ed i *mutex*.

Per capire la differenza tra le due tipologie, bisogna spiegare brevemente il concetto di *Interrupt Request Level (IRQL)*. Ad ogni processore viene associato un `KIRQL`, un oggetto opaco (in realtà una normalissima DWORD) per rappresentare l'*IRQL* corrente, che il thread in esecuzione può alzare o abbassare; le interruzioni che hanno un livello di privilegio maggiore dell'*IRQL* corrente fanno *preemption*, quelle a livello minore o uguale vengono mascherate. Bisogna puntualizzare che l'*IRQL* non ha niente a che vedere nè con la priorità di un processo, nè con il ring dei processori: Windows usa soltanto il ring 0 per il kernel ed il ring 3 per i programmi utente.

L'*IRQL* è una astrazione software presente solo nel kernel di Windows, il codice utente non ha consapevolezza di cosa sia, nè vi può accedere direttamente, anche se naturalmente può modificarlo temporaneamente tramite le *system call*.

I `KIRQL` che interessano nella nostra trattazione sono i primi tre:

- 0: `PASSIVE_LEVEL`, tutto il codice utente gira a `PASSIVE_LEVEL`, ed anche buona parte del codice kernel, e nessuna interruzione è mascherata;
- 1: `APC_LEVEL`, è il livello delle *Asynchronous Procedure Call*, funzioni che vengono eseguite in modo asincrono nel contesto di esecuzione di un thread, ogni thread ha la sua coda di *APC*;
- 2: `DISPATCH_LEVEL`, è il `KIRQL` più alto tra i *Software IRQL*, lo stesso schedulatore di Windows gira a questo livello;
- valori più alti di 2 sono usati per gestire le interruzioni hardware.

I *mutex* sono semafori efficienti utilizzabili per le routine che vengono eseguite a `KIRQL <= APC_LEVEL`, quando un thread cerca di acquisire un *mutex* già acquisito, la sua esecuzione è sospesa fino a che il *mutex* non viene rilasciato.

Gli *spin lock* sono semafori meno efficienti, utilizzabili per le routine che vengono eseguite a `KIRQL >= DISPATCH_LEVEL`. La loro inefficienza nasce dal fatto che acquisire uno *spin lock* porta automaticamente il livello di privilegio del thread a `DISPATCH_LEVEL`, rendendo quindi lo stesso non interrompibile da tutte le altre interruzioni software e nei sistemi multi-core può creare attesa attiva durante l'acquisizione.

Nel kernel di Windows, le *DPC* (cfr. sezione 4.1) vengono eseguite sempre a `KIRQL = DISPATCH_LEVEL`, quindi siamo obbligati ad usare gli *spin lock*, tuttavia il problema dell'attesa attiva non sussiste, poichè non possono mai essere schedulate contemporaneamente due *DPC* dello stesso tipo, quindi non esisterà mai un'istanza di `dummynet()` che mette in busy-stall la cpu in attesa del rilascio dello *spin lock* da parte di una precedente istanza.

Il rimappaggio delle funzioni viene mostrato nel listato 5.1, ed è particolarmente semplice in quanto le sintassi delle primitive di Windows sono compatibili con quelle di FreeBSD, si effettua quindi con delle direttive di preprocessing.

Listato 5.1: Rimappaggio degli *spin lock*

```
#define DEFINE_SPINLOCK(x)              NDIS_SPIN_LOCK x
#define mtx_init(m,a,b,c)               NdisAllocateSpinLock(m)
#define mtx_lock(_l)                    NdisAcquireSpinLock(_l)
#define mtx_unlock(_l)                  NdisReleaseSpinLock(_l)
#define mtx_destroy(m)                  NdisFreeSpinLock(m)
```

5.2 Allocazione e deallocazione della memoria dinamica

Vengono ridefinite la `malloc()` e la `free()` con le funzioni di allocazione e deallocazione di memoria utilizzabili nel kernel, come mostrato nel listato 5.2.

Listato 5.2: Rimappaggio di `malloc()` e `free()`

```
#define malloc(_size, _type, _flags) my_alloc(_size)
#define free(_var, type) ExFreePool(_var)

void *
my_alloc(int size)
{
        void *_ret = ExAllocatePoolWithTag(NonPagedPool, size, 'wfpi');
        if (_ret)
                memset(_ret, 0, size);
        return _ret;
}
```

Vengono ignorati i parametri `type` e `flags`, si alloca la memoria dinamica con la `ExAllocatePoolWithTag` e si ritorna l'indirizzo della memoria allocata. Da notare l'utilizzo della versione `WithTag`, che consente di marcare la memoria allocata con un identificatore univoco relativo al driver, in questo modo tramite appositi programmi di test è possibile verificare che il driver sia privo di *memory leak*. Inoltre la zona di memoria viene completamente cancellata, poichè il codice assume che lo sia, e questo previene potenziali errori su piattaforme in cui la memoria dinamica non è inizializzata.

La `free()` viene semplicemente rimappata sulla `ExFreePool()`, ignorando il parametro `type`.

5.3 Funzioni della *Run Time Library*

Un modulo del kernel non può ovviamente utilizzare le funzioni di libreria tipiche di un programma in spazio utente. Esiste tutta una serie di funzioni accessibili da un driver, chiamate *Kernel-Mode Support Routines*[7], alcune delle quali forniscono una versione kernel delle funzioni classiche, e sono quelle della famiglia `RtlXxx`.

Il listato 5.3 mostra l'implementazione delle funzioni che convertono interi a 16 e 32 bit da *network order* a *host order*, vengono semplicemente rimappate sulle funzioni di libreria del kernel che effettuano il *byte swap* su oggetti di lunghezza approriata.

Listato 5.3: Conversione tra *network order* e *host order*

```
#define htons(x) RtlUshortByteSwap(x)
#define ntohs(x) RtlUshortByteSwap(x)
#define htonl(x) RtlUlongByteSwap(x)
#define ntohl(x) RtlUlongByteSwap(x)
```

Il listato 5.4 mostra invece l'implementazione della funzione di generazione di numeri casuali. Viene creato un seed statico, alla prima chiamata della funzione viene inizializzato utilizzando la parte meno significativa del system time, che ne garantisce l'unicità, i dati casuali vengono quindi generati tramite la `RtlRandomEx()` e complementati con l'apposita maschera per far rientrare il risultato nel range desiderato.

Listato 5.4: Generazione di numeri casuali

```
int random(void)
{
        static unsigned long seed;
        if (seed == 0) {
                LARGE_INTEGER tm;
                KeQuerySystemTime(&tm);
                seed = tm.LowPart;
        }
        return RtlRandomEx(&seed) & 0x7fffffff;
}
```

Capitolo 6

Adattamento dell'ambiente di sviluppo e test

Quod scripsi, scripsi

Pilato

6.1 Adattamento dell'ambiente di sviluppo

L'ambiente di sviluppo di un driver Windows, fornito dal *Driver Development Kit*, è una normale shell MS-DOS in cui vengono settate opportunamente le variabili d'ambiente relative al path degli eseguibili, alle directory di inclusione e di libreria, e alla piattaforma per cui si vuole effettuare la compilazione (è anche possibile crosscompilare per qualsiasi versione di Windows a partire da XP, sia a 32 che a 64 bit). Viene inoltre fornita un'utility *make* proprietaria, che utilizza una particolare sintassi per configurare la compilazione di progetti distribuiti su più file e directory.

Data la limitatezza della shell MS-DOS, e la necessità di una shell *bash* con relative utility per compilare il programma utente *ipfw*, si sono studiate le meccaniche dell'ambiente di sviluppo Microsoft, e si è creato un *Makefile* per lo *GNU make* in grado di compilare sia la parte utente, utilizzando il GCC e gli header di Cygwin, sia la parte kernel utilizzando il compilatore Microsoft e gli header del *DDK*. Come

nei normali software UNIX, sono previsti i consueti target di compilazione, pulizia dei sorgenti, e creazione di un archivio *tar* distribuibile.

È stata anche fornita la possibilità di compilare il client con il *Tiny C Compiler*, ottenendo in questo modo un binario più leggero, indipendente dalla libreria dinamica Cygwin, e distribuibile sotto licenza BSD, come il modulo.

6.2 Ricostruzione e organizzazione degli header

Per ovviare alla mancanza di alcuni header utilizzati da *dummynet*, è stato necessario ricostruire svariate definizioni mancanti, la lista seguente spiega in che modo sono state organizzate.

- Nella sottodirectory `include` sono presenti i file ricavati direttamente dai sistemi UNIX, contenenti le definizioni delle principali costanti numeriche, i codici di errore, gli header IP e degli altri principali protocolli, e naturalmente gli header delle strutture proprietarie utilizzate da *dummynet*. Alcuni semplici rimappaggi, come quelli relativi alle funzioni `malloc()` e `free()` sono stati effettuati direttamente qui.

- Nella sottodirectory `include_e` sono presenti quei file che sono dipendenze da header presenti nella cartella precedente, o che contengono definizioni relative a funzioni del software originario non implementate su Windows. Per non modificare inutilmente i file originali, e per non gravare sulla loro leggibilità con ulteriori direttive di preprocessing, si è scelto di creare un albero di file vuoti, che vengono creati dal *make* al momento della compilazione, e distrutti durante il *clean* dei sorgenti.

- Il file `glue.h`, posto nella cartella principale della distribuzione, contiene le definizioni delle strutture dati condivise tra modulo e programma utente, quali le `sockopt`, e più in generale tutto ciò che è coinvolto nella comunicazione con lo spazio utente. Sono presenti inoltre alcune costanti estrapolate direttamente da header UNIX, contenute in file che non era necessario includere per intero.

- Il file `missing.h`, nella sottodirectory dei sorgenti del modulo, contiene definizioni presenti su FreeBSD, ma assenti sia su Linux che su Windows.

- Il file `winmissing.h`, anch'esso nella sottodirectory dei sorgenti del modulo, contiene ulteriori remapping, definizioni e costanti necessarie esclusivamente al porting Windows. In questo file sono contenuti anche i *typedef* che consentono di tradurre i tipi ISO-C presenti nel sorgente originario (del tipo `uint16_t`, `uint32_t`, ...) nei corrispettivi tipi predefiniti utilizzati dal compilatore Microsoft (`USHORT`, `UINT`, ...).

I file `winmissing.h`, `missing.h` e `glue.h` vengono inclusi nella compilazione non con la classica sintassi `#include`, ma attraverso un'opportuna opzione del compilatore, che li inserisce all'inizio del testo da passare al preprocessore, nell'ordine prefissato. In questo modo, oltre a non modificare, ancora una volta, i sorgenti originali, è possibile controllare in maniera diretta in che modo le funzioni vengono rimappate, percorrendo in senso inverso la catena delle inclusioni. Nell'ottica, sempre utilizzata in questo progetto, di *nested compatibility layers*, è possibile quindi scegliere a quale livello effettuare il rimappaggio di una funzione.

Un ulteriore file di compatibilità è stato realizzato per supportare alcune sintassi del compilatore TCC, che differiscono da quelle del GCC, in particolar modo si sono dovute esplicitare le convenzioni di chiamata alle funzioni della libreria *Winsock2*, che utilizzavano il vecchio standard *stdcall* invece del più recente *cdecl*.

6.3 Distribuzione binaria e script di test

È possibile generare tramite *make* un file compresso pronto per la distribuzione, contenente:

- il client compilato con GCC e fornito insieme alla dll di Cygwin;
- il client compilato con TCC;
- il modulo *.sys* insieme ai due file *.inf* necessari all'installazione;
- uno script *.bat* che testa le principali funzionalità del driver, impostando delay, limitazioni di banda, perdita di pacchetti e verificando con dei trasferimenti di prova che queste regole siano gestite correttamente; mostra inoltre all'utente la sintassi di alcuni comandi di *ipfw* e verifica che che questi vengano riconosciuti ed inseriti correttamente nel sistema;

in tal modo è immediatamente disponibile per l'utente finale un pacchetto binario pronto all'utilizzo che non necessita di download aggiuntivi.

6.4 File *.INF* di installazione

L'installazione di un *NDIS intermediate filter driver* richiede due file *.INF*, uno relativo al *Service*, ed uno relativo ai vari *Miniport* virtuali. Vengono di seguito descritte le sezioni obbligatorie.

- `Version section`: contiene informazioni relative alla versione di Windows a cui il driver è dedicato (in realtà sono accomunate tutte le versioni basate sul kernel di NT, da Windows XP in poi), la versione del driver e la data di rilascio, indicazioni sul *Manufacturer* e sul *Provider* (generalmente coincidenti), e la tipologia di driver. Nel file relativo al *Service* viene specificata la classe `NetService`, in quello relativo al *Miniport* la classe `Net`. Ciascuna classe ha il suo `ClassGUID` univocamente specificato, ricavabile dalla documentazione Microsoft.

- `Manufacturer section`: viene specificata una stringa che identifica il creatore del driver, seguita da una lista di tuple "sistema operativo / architettura hardware", ciascuna delle quali può avere specifiche direttive di installazione.

- `Models section`: per ciascuna tupla precedentemente dichiarata viene specificato il nome dell'oggetto da installare e la sezione contenente le relative informazioni.

- `DDinstall section`: per ciascun oggetto vengono specificate le *characteristics* del driver. Per il *Service* si specifica che è un driver di tipo *filter*, gestito dal sistema *NDIS*, per il *Miniport* invece si specifica che è un device di tipo virtuale, non visibile all'utente, ed il cui caricamento e scaricamento vengono gestiti automaticamente dal relativo *Service*.

- `SourceDisks section`: vengono dichiarati i file da copiare e le directory di sistema in cui copiarli.

- `AddReg section`: vengono dichiarate le chiavi da inserire nel registro di sistema. Questa sezione è presente solo nel file *Service*, e contiene il nome del *Miniport* virtuale che il servizio gestirà. Si noti che i *binding* vengono settati esplicitamente a `noupper,nolower`, per indicare che sarà il sistema *NDIS* a gestirli dinamicamente a runtime in base alla tipologia di scheda di rete installate.

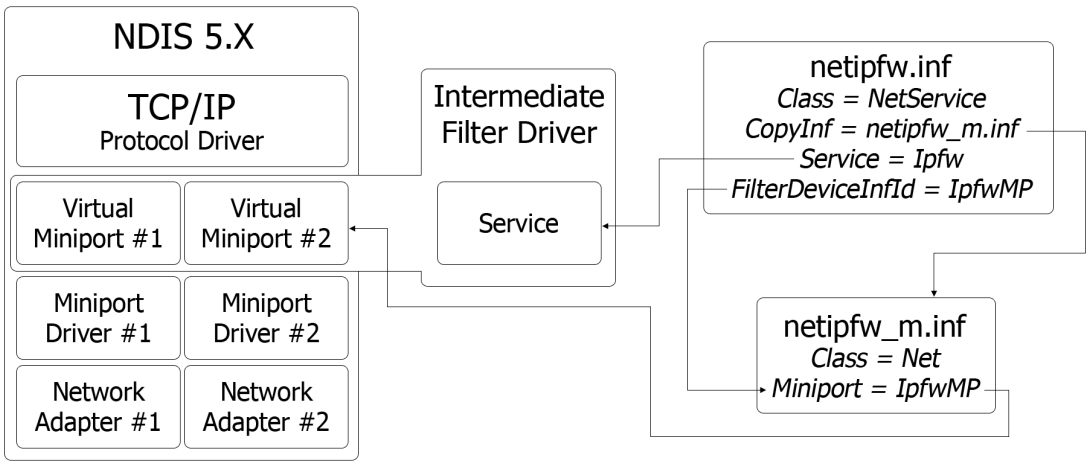

Figura 6.1: I file *.INF* ed i componenti del modulo

- `Strings section`: tipicamente alla fine del file *.INF* vengono espanse le stringhe utilizzate nelle sezioni precedenti come identificatori brevi.

La figura 6.1 (cfr. figura 3.3) mostra la relazione tra i file *.INF* e le componenti del modulo.

6.5 Librerie collegate

In fase di collegamento, il modulo si appoggia alle seguenti librerie:

- `ntoskrnl.lib`, è la libreria standard utilizzata da qualsiasi driver Windows;
- `BufferOverflowK.lib`, contiene le versioni *safe* di alcune primitive, esplicitamente progettate contro gli exploit del kernel ed i buffer overflow;
- `hal.lib`, è la libreria dell'*Hardware Abstraction Layer* contenente le funzioni la cui implementazione dipende dall'architettura della macchina su cui gira il sistema operativo, come le funzioni dei *Performance Counter*;
- `ndis.lib`, è la libreria del sistema *NDIS*;
- `wmilib.lib`, contiene le funzioni del *Windows Management Instrumentation*, un'estensione del *Windows Driver Module* utilizzato per la comunicazione gestita e controllata di eventi tra diverse parti del sistema operativo.

6.6 Supporto per piattaforme a 64 bit

Dato il progressivo abbandono da parte della Microsoft del supporto a Windows XP, e la continua e crescente diffusione dei sistemi operativi a 64 bit, per garantire continuità allo sviluppo del progetto, si è deciso di produrre una versione del driver per Windows a 64 bit.

Il compilatore Microsoft utilizzato per il sorgente del modulo è pienamente compatibile con le convenzioni a 64 bit, ed è in grado di crosscompilare per qualsiasi versione del sistema operativo, non sono state necessarie ulteriori modifiche al codice, in quanto gli operandi sono stati sempre dichiarati specificandone la lunghezza. Per integrare il supporto alla compilazione a 64 bit nell'ambiente ibrido GNU/Microsoft utilizzato nel progetto, sono state introdotte delle variabili all'interno del Makefile GNU principale:

- `DRIVE` identifica l'unità logica su cui è installato il *DDK*, comprensivo di due punti;
- `DDKDIR` identifica il path assoluto, privo della lettera di unità ed in formato UNIX, della directory radice del *DDK*;
- `TARGETOS` identifica il sistema operativo target per cui effettuare la compilazione, valori possibili sono:
  - `wnet` per Windows Server 2003,
  - `wlh` per Windows Vista e Windows Server 2008,
  - `win7` per Windows 7.

Uno script, a partire da questi dati, ricostruisce ed esporta le variabili d'ambiente necessarie all'utility di compilazione Microsoft relative al path dei file di inclusione e di libreria specifici per il sistema operativo target, e setta `_BUILDARCH=AMD64` per segnalare la compilazione a 64 bit.

Difficoltà maggiori invece sono state riscontrate nell'adattamento di *ipfw*, in quanto i compilatori utilizzati per il programma utente non supportano la compilazione a 64 bit (nel caso di Cygwin) oppure non la gestiscono ancora correttamente (nel caso del TCC). È stato quindi effettuato un adattamento del codice per poter permettere ad un binario a 32 bit di comunicare correttamente con un modulo a 64 bit. Principalmente il problema non è nella lunghezza degli operandi numerici, che

può essere fissata per entrambi i compilatori dichiarando opportunamente il numero di bit delle variabili, quanto nella dimensione dei puntatori, che nei sistemi a 64 bit occupano il doppio dello spazio, e il conseguente allineamento in memoria. Si è già sottolineato più volte come il passaggio di puntatori tra kernel e spazio utente sia un'operazione inutile poichè sono diversi e mutuamente inaccessibili gli spazi di indirizzamento; per cui sono state semplificate dove possibile le strutture dati coinvolte nella comunicazione, dove invece non è stato possibile è stato inserito un padding di variabili inutilizzate per far combaciare il layout delle strutture dati generate dal programma utente a 32 bit con quello che il modulo a 64 bit si aspetta di ottenere. In questa maniera non viene compromessa nè la stabilità nè le prestazioni del modulo, che è compilato nativamente per i nuovi sistemi operativi, e si è minimizzato il lavoro in spazio utente dove non è stato necessario effettuare ulteriori adattamenti ad altri compilatori.

6.7 Verifica statica del codice

È stata effettuata una verifica del codice utilizzando tutti gli strumenti messi a disposizione dal *DDK*:

- *poolmon*, una utility che mostra in tempo reale l'allocazione e la deallocazione della memoria di un driver, quando questo utilizza l'apposita funzione per marcare con un *tag* le pagine allocate, e consente di identificare tempestivamente la presenza di *memory leak*;

- verifica *PREfast* tramite *OACR monitor*, un tool che, durante la compilazione, esamina il codice per identificare rapidamente errori macroscopici di programmazione (come mancato rilascio di lock), e verifica che tutte le parti vitali di un driver vengano dichiarate correttamente, e vengano segnalate tramite le apposite macro al compilatore, per assicurare un corretto caricamento del modulo;

- *Static Driver Verifier*, un tool di verifica statica che ispeziona tutto il codice del driver, mirato ad identificare possibili violazioni del *Windows Driver Model* a runtime; la verifica è effettuata simulando l'esecuzione del driver, ed ispezionando tutti i possibili cammini del codice, facendo il minor numero possibile di assunzioni sullo stato della macchina, in modo da scovare problematiche

potenzialmente difficili da identificare e riprodurre con le tecniche di debug tradizionale; il set di regole che vengono verificate è molto esteso, e comprende le richieste *IRP*, i livelli di interrupt *IRQL*, la gestione del Plug and Play, il Power Management, la sincronizzazione dei thread.

Capitolo 7

Istruzioni per l'utente

Mettete qui la vostra citazione

B. Stroustrup

7.1 Installazione del modulo

Come già detto nella sezione 6.3, la distribuzione del software comprende il binario `ipfw.sys` e i due file `netipfw.inf` e `netipfw_m.inf`. La procedura di installazione è quella tipica di tutti i *Net Service*.

- Accedere alle proprietà di una qualsiasi scheda di rete installata ed attiva sul sistema: questo può essere fatto tramite l'icona eventualmente presente nel *systray*, o mediante il collegamento *Connessioni di rete* presente nel *Pannello di Controllo*;
- premere il pulsante per installare un nuovo servizio;
- selezionare *disco driver* e muoversi nella directory contenente i binari di *dummynet*;
- selezionare uno qualsiasi dei due file *.INF*;

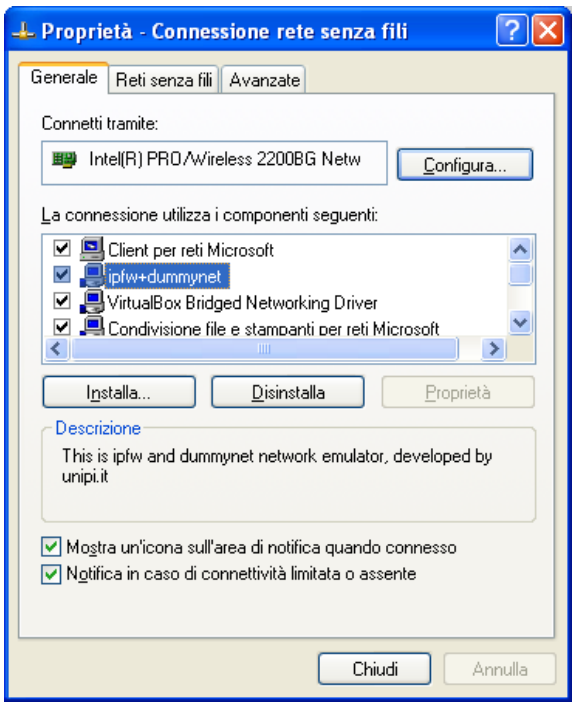

Figura 7.1: Installazione del modulo

- a seconda delle impostazioni di sicurezza del sistema operativo, potrebbe essere necessario confermare più volte l'installazione di un driver privo di firma digitale.

Al termine della copia automatica del file *ipfw.sys* nella cartella di sistema, e dell'inizializzazione del modulo, questo dovrebbe correttamente apparire nella lista dei servizi di rete installati, come mostrato in figura 7.1. A questo punto è possibile utilizzare una shell MS-DOS, o in alternativa una shell *bash* sotto Cygwin che offre maggiore comodità d'uso, per lanciare il programma utente *ipfw* e iniziare a utilizzare il modulo. Si noti che da Windows Vista in poi, un programma utente che voglia aprire un canale di comunicazione *IOCTL* con un device hardware, virtuale o meno, deve essere eseguito esplicitamente con i privilegi di amministrazione.

7.2 Configurazione del software

7.2.1 Configurazione del firewall

All'avvio del modulo, il firewall è configurato con un'unica regola di default, che permette il passaggio di tutto il traffico. La costruzione di regole è molto semplice

e utilizza la seguente sintassi:

- keyword `add`;
- numero della regola (opzionale): le regole sono numerate, ed il parsing avviene in maniera sequenziale, è quindi utile per assegnare una priorità alle regole; se questo parametro non viene specificato, le regole vengono automaticamente aggiunte in fondo alla lista e numerate ad intervalli di 100;
- probabilità `prob`: se i parametri della regola sono verificati, il match avviene solo con una determinata probabilità;
- parametro *action*: può essere `allow` o `deny`, il cui significato è intuitivo; altri valori saranno esaminati in seguito;
- corpo vero e proprio della regola: una lista di parametri che, durante l'analisi del pacchetto in transito, servono ad effettuarne il match; anche questa lista viene esaminata sequenzialmente, è quindi buona norma specificare i parametri in ordine decrescente di specificità. I parametri utilizzabili, il cui uso è assolutamente intuitivo, comprendono:
  - `in` o `out`, che identificano un pacchetto rispettivamente in ingresso o in uscita dal sistema, questo parametro è particolarmente importante, sia per semplificare notevolmente la struttura di una regola, sia per la configurazione di funzionalità avanzate di *shaping* (cfr. sezione 7.2.2);
  - `src-ip` e `dst-ip`, che effettuano il macth sull'ip di provenienza e di destinazione del pacchetto;
  - `src-port` e `dst-port`, che effettuano il match sulle porte;
  - `proto`, che effettua il match sul protocollo del pacchetto;
  - `tcpflags`, che effettua il match sui flag TCP del pacchetto;
  - keyword `established`, che effettua il match solo su connessioni tcp già stabilite, ed il suo duale `setup`

Ogni parametro può essere preceduto dalla keyword `not`, per effettuare il match complementare.

È possibile rimuovere una regola con il comando `del` seguito dal numero della regola che si intende rimuovere, oppure eliminarle tutte con il comando `flush`.

I comandi `show` e `list` mostrano il *ruleset* corrente, rispettivamente con e senza statistiche.

Il listato 7.1 mostra alcuni comandi di esempio, con la relativa descrizione.

Listato 7.1: Esempi di configurazione del firewall

```
-- dropping all packets of a specific protocol --
ipfw add deny proto icmp

-- dropping all packets from IP x to IP y --
ipfw add deny src-ip 1.2.3.4 dst-ip 5.6.7.8

-- dropping all web browsing outgoing connections --
ipfw add deny out dst-port 80

-- dropping randomly 50% of the outgoing udp traffic --
ipfw add prob 0.5 deny proto udp out
```

7.2.2 Configurazione avanzata

Per utilizzare le funzionalità avanzate di *dummynet* al fine di simulare un particolare stato della rete, è necessario spiegare brevemente le entità che il software utilizza per la modellizzazione di tale stato:

- le *pipe* emulano un link con una determinata banda, un determinato delay, ed altre caratteristiche della linea quali la dimensione della coda di uscita e la percentuale di perdita di pacchetti;
- le *queue* sono astrazioni che schematizzano la politica di schedulazione del traffico, ad ogni coda è associato un *peso* ed il riferimento ad una *pipe*.

In pratica, le *pipe* vengono usate per determinare la banda del link, e le *queue* per determinare in che modo i vari flussi di dati si ripartiranno tale banda.

Scenari tipici di utilizzo sono l'emulazione di una particolare connessione, o la schedulazione del traffico uscente.

Emulazione di un link ADSL ad altissima latenza

Il listato 7.2 mostra i comandi per emulare un link ADSL di scarse prestazioni ed elevatissime latenze.

Listato 7.2: Emulazione di un link ADSL

```
ipfw pipe 1 config bw 640Kbit/s delay 100ms
ipfw pipe 2 config bw 128Kbit/s delay 100ms
ipfw add pipe 1 in
ipfw add pipe 2 out
```

La configurazione avviene in due fasi:

- si creano delle *pipe* specificando le caratteristiche fisiche del link che vogliamo emulare (banda e latenza),
- si inseriscono le regole del firewall, utilizzando questa *action* per indirizzare il traffico all'interno delle *pipe* appena create.

Si noti l'utilizzo dei parametri `in` e `out`, che rendono particolarmente compatta la sintassi della regola, e che sono particolarmente indicati per identificare il traffico in quei sistemi in cui l'indirizzo della macchina sia determinato dinamicamente tramite DHCP, e che quindi possa addirittura cambiare durante l'esecuzione del modulo.

Schedulazione del traffico e ripartizione della banda

Altro scenario tipico, la cui configurazione è mostrata nel listato 7.3, è quello di una macchina su cui vengano offerti più servizi, ad esempio web ed ftp, e si voglia ripartire equamente la banda in uscita.

Listato 7.3: Schedulazione del traffico uscente

```
ipfw pipe 1 config bw 9.5Mbit/s
ipfw queue 1 config weight 1 pipe 1
ipfw queue 2 config weight 1 pipe 1
ipfw add queue 1 src-port 80 out
ipfw add queue 2 src-port 21 out
```

Supponendo di avere a disposizione un link a 10 megabit per la macchina, si effettua come prima cosa un *hard limiting* al 95% della banda disponibile, questo garantisce allo schedulatore una certa tolleranza e previene la formazione di code non gestite sul gateway in presenza di sovraelongazione temporanea, quindi si configurano due *queue* con lo stesso *weight* che convergono all'interno della medesima *pipe*; infine, come nel caso precedente, si settano le regole del firewall per la classificazione dei pacchetti.

In questo modo si ottengono i seguenti risultati:

- ciascun servizio, quando l'altro non ha client da servire, può occupare tutta la banda disponibile;

- in caso di carico di entrambi i servizi, essi dividono equamente la banda disponibile (diversi rapporti sono naturalmente possibili, assegnando *weight* diversi);
- viene garantita la *fairness* tra i vari client all'interno della medesima *queue*.

Per un elenco esaustivo delle opzioni disponibili e della sintassi completa, si faccia riferimento alla *man page* di *ipfw*[2].

7.3 Limitazioni rispetto al software originale

Alcune funzionalità del software originale non sono state implementate nella versione Windows.

- Intercettazione del traffico locale: in Windows l'interfaccia di loopback è implementata internamente al protocollo IP, inoltre anche il traffico generato verso gli indirizzi locali delle interfacce reali non attraversa lo stack di rete, ma viene processato direttamente dal protocollo, pertanto la sua intercettazione è impossibile, ed è una limitazione non aggirabile in alcun modo.
- Classificazione del traffico basato sul nome dell'interfaccia: in Windows le interfacce di rete non hanno nomi simbolici come nei sistemi UNIX, ma vengono identificate attraverso nomi complessi del filesystem virtuale dei device, del tipo `{9E407963-4C68-4336-9008-3236DF509606}`, che corrisponde alla chiave di registro che indentifica univocamente l'hardware; da un punto di vista pratico la loro scomodità avrebbe reso questa funzionalità scarsamente usata.
- Keyword `me`: questa keyword permette il match di qualsiasi indirizzo locale della macchina, tuttavia non esiste un metodo efficiente, per un *intermediate driver* di risalire dalla struttura `ADAPTER` che caratterizza una scheda di rete alla lista dei suoi indirizzi IP, pertanto questa funzionalità non è stata implementata per motivi di efficienza.
- Classificazione del traffico basata su *uid/gid*: Windows è un sistema non orientato alla multiutenza, la diversificazione di utenti e gruppi si limita alla proprietà dei file, e alla personalizzazione delle impostazioni dei programmi; il kernel non tiene traccia, nelle strutture dati che rappresentano il pacchetto, dell'utenza che l'ha generato, non è quindi possibile effettuarne il match.

- Supporto a *Internet Protocol Version 6*: anche se Windows supporta ufficialmente *ipv6*, questa caratteristica è scarsamente utilizzata ed attualmente *dummynet* supporta solo *ipv4*.

- Funzionalità di *LOG*, vista la complessità delle *logging facility* di Windows, la funzionalità è attualmente disabilitata, in attesa di una soluzione dedicata e integrata nel modulo.

7.4 Windows come router

In Windows è possibile abilitare il routing dei pacchetti, utilizzando il programma `RegEdit` per modificare il registro di sistema, e seguendo le istruzioni riportate nel listato 7.4:

Listato 7.4: Abilitazione del routing

```
Nell'editor del Registro di sistema individuare la seguente chiave:
KEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

Impostare il seguente valore di registro:
Nome valore: IPEnableRouter
Tipo valore: REG_DWORD
Dati valore: 1
```

Diventa quindi possibile, per quelle reti basate su Windows, che hanno già una macchina server dedicata (server *WINS*, *Active Domain*, server Web o Sql tramite *IIS*), o che utilizzano le versione server del sistema operativo per gestire la connettività privata o aziendale, usare *dummynet* per monitorare gli accessi, filtrare il traffico indesiderato, effettuare shaping sul traffico uscente per limitare i costi, e gestire tutte quelle operazioni tipiche di un router con funzionalità avanzate senza spendere in ulteriore hardware dedicato o in licenze di software commerciali costosi, affidandosi ad un prodotto completamente gratuito, open source, ed in continuo sviluppo.

Bibliografia

[1] FreeBSD Handbook, *Configuration and Tuning*
February 27, 2010
`http://www.freebsd.org/doc/handbook/configtuning-sysctl.html`

[2] FreeBSD System Manager's Manual, *Ipfw man page*
June 24, 2009
`http://www.freebsd.org/cgi/man.cgi?query=ipfw`

[3] M.Carbone and L.Rizzo, *An emulation tool for PlanetLab*
March 16, 2010
`http://info.iet.unipi.it/~luigi/papers/20100316-cc-preprint.pdf`

[4] M.Carbone and L.Rizzo, *Dummynet revisited*
SIGCOMM CCR, Vol. 40, No. 2, April 2010
`http://www.freebsd.org/cgi/man.cgi?query=ipfw`

[5] Microsoft Corporation, *MSDN: Defining I/O Control Codes*
November 19, 2009
`http://msdn.microsoft.com/en-us/library/ms795909.aspx`

[6] Microsoft Corporation, *MSDN: DeviceIoControl Function*
December 11, 2009
`http://msdn.microsoft.com/en-us/library/aa363216(VS.85).aspx`

[7] Microsoft Corporation, *MSDN: Driver Support Routines*
November 19, 2009
`http://msdn.microsoft.com/en-us/library/ms795146.aspx`

[8] Microsoft Corporation, *MSDN: Introduction to Spin Locks*
November 19, 2009
`http://msdn.microsoft.com/en-us/library/aa490179.aspx`

[9] Microsoft Corporation, *MSDN: IRP*
November 19, 2009
http://msdn.microsoft.com/en-us/library/dd852053.aspx

[10] Microsoft Corporation, *MSDN: IRP Major Function Codes*
November 19, 2009
http://msdn.microsoft.com/en-us/library/ms806157.aspx

[11] Microsoft Corporation, *MSDN: NDIS_PACKET*
November 19, 2009
http://msdn.microsoft.com/en-us/library/ms797623.aspx

[12] Microsoft Corporation, *MSDN: Overview of the Windows I/O Model*
November 19, 2009
http://msdn.microsoft.com/en-us/library/ms796160.aspx

[13] Microsoft Corporation, *MSDN: Timer Objects and DPCs*
November 19, 2009
http://msdn.microsoft.com/en-us/library/aa490171.aspx

[14] Microsoft Corporation, *MSDN: Using MDLs*
November 19, 2009
http://msdn.microsoft.com/en-us/library/aa489506.aspx