Enrico Manzini - s102713

Programmazione in Ambienti Distribuiti

selfDHCP

un programma di autoconfigurazione di rete

Docente:
Prof. Antonio Lioy

A.A. 2002 - 2003

Sommario

Introduzione

selfDHCP nasce da una necessità incontrata più volte lavorando su reti informatiche: poter configurare una macchina in maniera trasparente all'utente. In effetti, esistono già strumenti, come il protocollo DHCP, molto più evoluti e potenti del programma presentato in questa relazione. Non sempre però questi strumenti sono disponibili, o quando anche lo fossero si può non essere in grado di utilizzarli, o si può non volerlo fare.

Pur riprendendo nel nome il protocollo DHCP, selfDHCP non ha nulla a che vedere con il Dynamic Host Configuration Protocol, se non le finalità. A differenza del vero DHCP, che richiede la presenza di un server, selfDHCP è pensato per operare *stand-alone*, nella maniera più discreta e meno invasiva possibile.

Gli scopi principali del programma sono due. Il primo è il riconoscimento automatico di reti conosciute tramite l'utilizzo di profili. Questo può essere utile per connettere in maniera rapida un host ad una rete, anche quando questa non abbia il supporto per DHCP. Il secondo scopo è di ricostruire i parametri di una rete sconosciuta, attraverso l'analisi dei pacchetti in transito, e configurare l'host in modo che possa utilizzare in maniera attiva la rete, senza però causare problemi agli altri host (duplicazione di IP, trasmissione di traffico fuori netmask, etc.).

Per permettere una maggiore flessibilità, selfDHCP ha due modi operativi principali. Può comportarsi come una normale applicazione, e in questo caso termina non appena abbia individuato una configurazione di rete funzionante; oppure può comportarsi come *daemon*, e continuare a monitorare la rete per individuare eventuali cambiamenti (soprattutto per evitare il pericolo della duplicazione di IP).

Poiché selfDHCP si basa sulla cattura dei pacchetti in transito, il suo ambito di funzionamento è limitato a reti di tipo *non-switched*, ovvero dotate di apparati di rete non intelligenti (HUB ad esempio). Gli switch infatti inoltrano selettivamente i pacchetti solo sulle porte giuste, utilizzando una tabella di corrispondenze interna, e non permettono quindi la cattura del traffico destinato alle altre macchine.

selfDHCP è strettamente legato ai protocolli ethernet e IP (IPv4 per la precisione). Con pochi adattamenti dovrebbe essere in grado di funzionare su reti token-ring, è non è escluso che il principio sia applicabile ad altri protocolli; tuttavia la versione presente limita il supporto alla suddetta coppia di protocolli.

Per quanto riguarda infine il sistema operativo, la versione attuale di selfDHCP è stata sviluppata per funzionare su kernel linux. Avendo avuto cura di utilizzare il più possibile librerie conformi allo standard Posix o comunque portabili, il programma dovrebbe essere in grado di funzionare con modifiche minime in ambienti simili, come BSD, Solaris e altri sistemi basati su Unix. Più difficile sarebbe probabilmente il porting

verso sistemi basati su Microsoft Windows, anche per le grosse differenze soprattutto riguardo alla gestione dei thread.

Note preliminari

Questa relazione vuole essere un supporto per la comprensione del funzionamento del programma selfDHCP. Tuttavia, data la mole di codice che compone il programma, e nell'ottica di fornire un quadro d'insieme, non è stato ritenuto opportuno riportare troppe parti di codice sorgente, limitandosi per lo più alle strutture dati. Per quanto riguarda le funzioni, non sono riportati i prototipi, ma solo i nomi e qualche accenno al loro funzionamento.

D'altra parte, il codice di selfDHCP è ampiamente commentato (seppure in inglese), e per ogni funzione sono indicati chiaramente gli argomenti richiesti e i valori di ritorno. Lo stesso può dirsi di molti altri elementi non trattati specificamente in questa relazione, quali costanti simboliche o strutture dati di minore importanza.

Si rimanda pertanto alla lettura dei file sorgenti, in particolare agli *header* file di ogni modulo, qualora fossero necessari chiarimenti sul funzionamento di parti del programma.

selfDHCP è in una fase ancora preliminare di sviluppo. La versione cui fa riferimento questa relazione va considerata un'*alpha release*, pertanto alcune parti di codice potrebbero risultare inutilizzate, o alcune *feature* previste non ancora implementate, o implementate parzialmente. Alcune delle idee all'origine del progetto si sono rivelate essere non attuabili, o troppo difficili da implementare, o ancora non sono state implementate per mancanza di tempo, e saranno invece presenti come miglioramenti in versioni successive; in quest'ambito ricadono soprattutto la parte euristica e in particolare quegli accorgimenti che dovrebbero garantire una maggiore *stealthness*.

Il progetto selfDHCP è ospitato dal sito Sourceforge all'indirizzo http://selfdhcp.sourceforge.net

Principio di funzionamento

selfDHCP basa il suo funzionamento sull'analisi dei pacchetti in transito sulla rete, e su alcune caratteristiche delle reti di tipo ethernet/IP. Attraverso l'analisi e la catalogazione degli indirizzi MAC e IP è infatti possibile attraverso un'opportuna euristica risalire alla configurazione della rete, o almeno ad una compatibile con quella effettivamente presente.

Una configurazione di rete tipica comprende, per una rete IP, un indirizzo di rete per la propria macchina e una netmask. Con queste due sole informazioni è possibile scambiare dati con gli altri host della rete, a patto di conoscere i loro indirizzi IP. Se però si vuole comunicare con l'esterno è necessario conoscere un'informazione in più, ovvero l'indirizzo IP di una macchina che faccia da gateway, ed è proprio sulle meccaniche del gateway che selfDHCP basa buona parte del suo funzionamento.

A livello ethernet infatti la rete è isolata, e non vengono passate informazioni dall'esterno. A livello IP invece è possibile la comunicazione verso l'esterno, ma c'è distinzione tra rete interna ed esterna, ed è basata sulla netmask. Semplificando le cose, quando un host vuole comunicare con un'altra macchina, confronta il proprio IP con quello del destinatario, e verifica che stia nella netmask. Se è così, gli indirizzi MAC di sorgente e destinatario corrispondono con quelli effettivi, e la comunicazione si svolge normalmente. Se invece l'indirizzo è fuori dalla netmask, interviene il meccanismo del gateway: il mittente spedisce il pacchetto sulla rete interna, con i suoi indirizzi IP e MAC come sorgenti, ma per destinazione mette l'IP della destinazione effettiva, e *l'indirizzo MAC del gateway*. Il gateway, quando riceve un pacchetto così fatto, sa dove deve inoltrarlo. Viceversa, quando il gateway riceve una risposta destinata alla rete interna, la inoltra verso la destinazione ponendo come sorgente l'IP del mittente effettivo, ma il suo MAC.

L'idea su cui si basa selfDHCP è di sfruttare questo meccanismo di “traduzione” automatica degli indirizzi, catalogando le coppie MAC/IP dei pacchetti che transitano in rete. Per ogni pacchetto è possibile, nella maggior parte dei casi, identificare due coppie (sorgente e destinazione; sono esclusi i pacchetti di tipo ARP-request[1]). Raccolto un numero sufficiente di pacchetti, e quindi di coppie, sarà possibile identificare quel MAC

[1] per questo tipo di pacchetto è nota solo la coppia sorgente, visto che è lo scopo del pacchetto quello di scoprire l'indirizzo IP del destinatario.

cui sono legati più indirizzi IP, che dovrebbe essere quello del gateway.

In realtà il compito di selfDHCP non si esaurisce con l'identificazione del gateway; è necessaria un'opportuna euristica in grado, una volta individuato il gateway, di risalire alla netmask e trovare un indirizzo IP disponibile. A differenza della parte di identificazione del gateway, che può contare su dati abbastanza precisi, queste due fasi sono più soggette ad errori. La netmask in realtà può non essere esattamente corrispondente con quella effettiva, purché sia *troppo* non più grande[2] di quella effettiva, soprattutto se più che con gli host della rete interna si vuole comunicare con l'esterno.

La ricerca di un IP è più critica, in quanto la collisione con un IP già esistente in rete può dare luogo a malfunzionamenti anche piuttosto evidenti, mentre l'attribuzione di un IP fuori netmask impedisce di fatto la comunicazione, sia con l'interno che con l'esterno della rete.

Come è evidente dai paragrafi precedenti, e come già accennato nell'introduzione, il meccanismo appena spiegato può funzionare solo se è possibile catturare tutto il traffico della rete. Questo accade su reti *non-switched*, come ad esempio quelle che utilizzano un HUB come apparato di interconnessione, ma diventa difficoltoso su reti di tipo *switched*[3]. In realtà sarebbe possibile fare sì che selfDHCP possa funzionare anche su reti dotate di switch, ma al prezzo di ricorrere a tecniche piuttosto invasive (per esempio l'avvelenamento della tabella ARP dello switch), con conseguente degrado delle prestazioni e perdita della "*stealthness*".

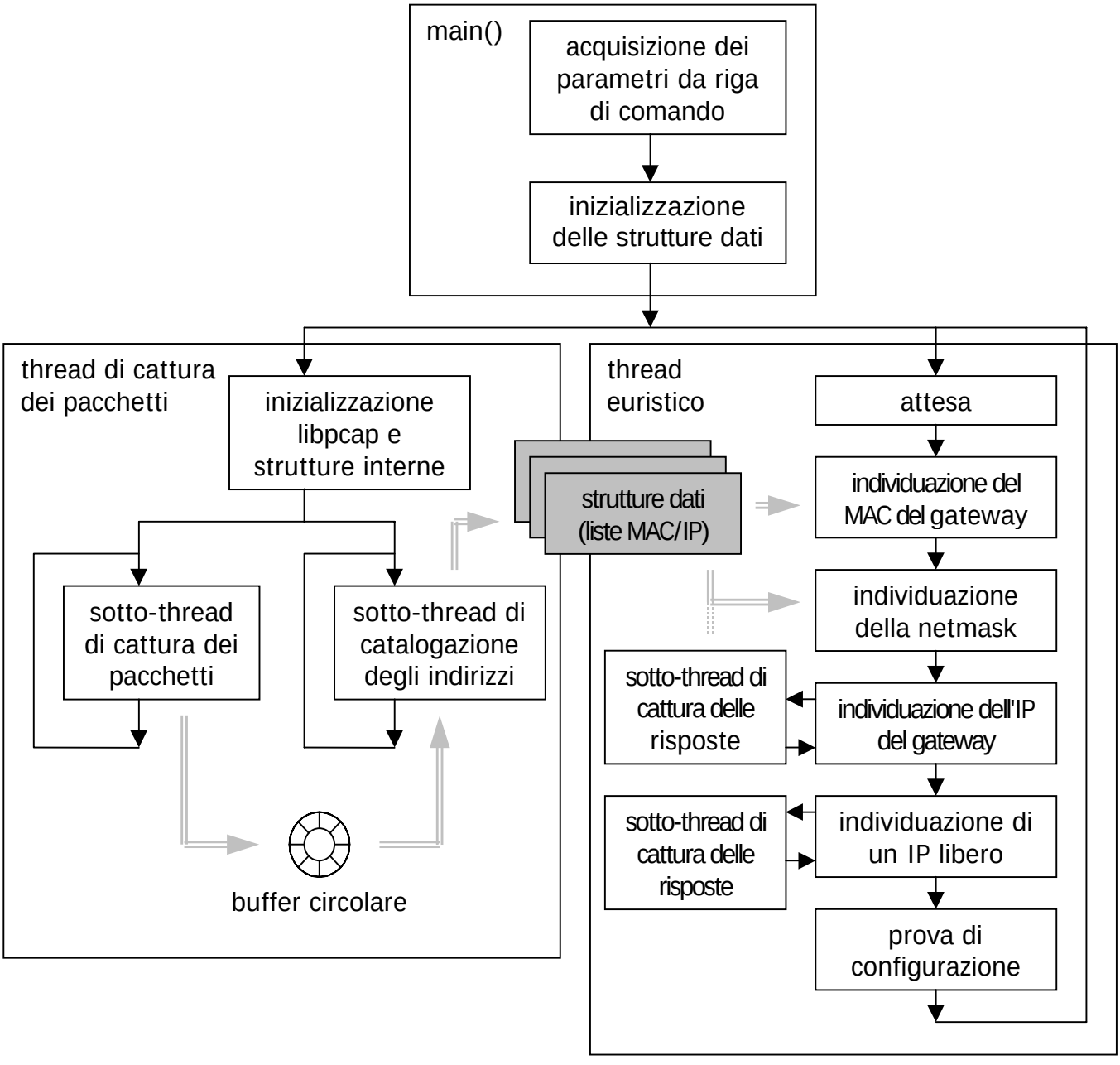
[2] ci sono casi un cui una netmask più grande di quella effettiva non pregiudica il funzionamento della rete, per esempio quando su una singola rete fisica coesistono più reti logiche separate, ad esempio, tramite una netmask piuttosto restrittiva.

[3] il pregio dello switch è proprio quello di limitare le collisioni che si hanno utilizzando invece un HUB, inoltrando il traffico solo sulle porte opportune. Uno switch la cui tabella ARP sia stata avvelenata non fa in effetti altro che comportarsi come un HUB.

Struttura del programma

In selfDHCP sono necessari diversi tipi di operazioni, con tempistiche piuttosto diverse: la sezione di cattura dei pacchetti, per la natura *bursty* del traffico di rete, deve essere in grado di assorbire in poco tempo discrete quantità di dati; d'altra parte la catalogazione delle coppie MAC/IP è un'operazione lenta, a causa della complessità delle strutture dati e dall'implementazione *thread-safe*. L'euristica si compone di una serie di operazioni abbastanza rapide, da svolgere in maniera saltuaria, ma senza interrompere la cattura dei pacchetti.

La necessità di conciliare queste necessità piuttosto differenti ha portato ad un'implementazione basata su due thread principali dedicati alla cattura dei pacchetti e all'euristica, divisi a loro volta in sotto-thread quando necessario. La struttura di massima del programma è schematizzata in figura:



Come si può notare dalla figura, il programma principale provvede all'acquisizione delle opzioni da linea di comando e all'inizializzazione delle strutture dati condivise. Dopo aver fatto questo, il programma si divide in due thread principali, uno dedicato alla cattura dei pacchetti e catalogazione degli indirizzi, l'altro all'euristica (implementativamente, il thread euristico prosegue il thread principale).

Il thread di cattura a sua volta è diviso in due sotto-thread fissi, uno che si occupa della cattura vera e propria dei pacchetti, l'altro della catalogazione delle coppie di indirizzi estratte dai pacchetti. I due thread comunicano attraverso un semplice buffer circolare dimensionato empiricamente in modo che non possa essere riempito completamente dal sotto-thread di cattura. Questa configurazione è stata suggerita dalla pratica, in quanto è stato notato come l'inserimento di una coppia all'interno delle liste di indirizzi sia operazione piuttosto lunga, e durante raffiche particolarmente intense di pacchetti un'implementazione a thread singolo rischiava di perdere pacchetti o, peggio, bloccarsi. Il buffer circolare ha risolto in parte questi problemi, disaccoppiando le diverse velocità dei due processi.

Per quanto riguarda il thread euristico, dopo un'attesa fissa (ma sarebbe meglio una condizione sul numero di coppie diverse registrate, o un misto delle due condizioni; ciò verrà implementato in una prossima versione) procede a cercare in sequenza i parametri della configurazione. L'ordine è importante, poiché l'unico parametro che può essere trovato in modo indipendente è l'indirizzo MAC del gateway, mentre tutti gli altri sono in qualche modo interdipendenti.

Due dei passi dell'algoritmo prevedono di uscire dalla modalità *stealth* e generare delle richieste ARP.I due passi sono l'individuazione dell'IP del gateway[4] e la ricerca di un IP libero da assegnare alla nostra macchina. In entrambi i casi, le richieste sono fatte "a nome di un altro", ovvero utilizzando come indirizzi MAC e IP quelli di un altro host per ridurre la possibilità di generare disturbo (ricordiamo che siamo su una *non-switched LAN*, quindi non è un problema che la richiesta ARP sembri provenire da un altro host, tanto noi vedremo comunque la risposta).

Sia per l'individuazione dell'IP del gateway che di un IP libero, è necessario poter intercettare le risposte alle richieste ARP iniettate in rete (nel primo caso per verificare la corrispondenza con l'indirizzo MAC, nel secondo per scartare l'IP considerato, poiché se si riceve risposta vuol dire che è già in uso da parte di un altro host).La soluzione inizialmente adottata prevedeva di affidare al thread di cattura dei pacchetti l'individuazione delle ARP reply da parte di specifici indirizzi, ma è stata scartata perché introduceva un eccessivo *overhead* operativo sul già sovraccarico thread di cattura,

[4] l'ideale per l'individuazione dell'IP del gateway sarebbe stata una richiesta RARP, che avrebbe fornito immediatamente il dato cercato senza bisogno di tentativi; purtroppo il protocollo RARP è per lo più in disuso, e non vi si può quindi fare affidamento per l'individuazione di questo parametro fondamentale

nonché un incremento di complessità del programma dovuto alla necessità di segnalare in qualche modo tra i due processi (di cattura ed euristico) su quali indirizzi filtrare le richieste e se si erano ricevute risposte.

Per ovviare questo inconveniente (nonché una certa propensione delle libpcap a perdere pacchetti in acquisizione o addirittura a bloccarsi), è stata adottata una soluzione alternativa, consistente in dei piccoli sotto-thread, della durata di pochi secondi, che si occupano esclusivamente di catturare le ARP reply necessarie, e ne forniscono un conteggio. Le libpcap permettono inoltre di fare questo in maniera piuttosto efficiente, poiché è possibile impostare i filtri *run-time* filtrando solamente le reply dell'host che ci interessa. Una volta che le ARP request sono state iniettate in rete, e dato un piccolo intervallo di tempo per permettere di ricevere la risposta, il sotto-thread di cattura è terminato esplicitamente dal thread euristico principale.

Il programma termina quando è stata trovata una configurazione funzionante (e in questo caso il comportamento varia a seconda delle impostazioni ricevute da linea di comando; si va dalla sola stampa a video della configurazione trovata, alla configurazione effettiva della scheda di rete e della tabella di routing), oppure allo scadere di un opportuno timeout (il thread di euristica ha un suo timeout interno, detto `TIME_GRANULARITY`, che scandisce ogni quanto viene fatto un tentativo di individuazione dei parametri di rete; il thread di euristica dunque farà un numero di cicli pari a timeout/`TIME_GRANULARITY`).

Se invece il programma è in modalità demone continua a girare anche dopo aver trovato una configurazione valida, per assicurarsi che non avvengano variazioni che invalidino la configurazione trovata: l'accensione di un host precedentemente spento, per esempio, potrebbe causare una duplicazione di indirizzi IP; oppure la nostra macchina potrebbe essere fisicamente sconnessa da una rete e connessa ad un'altra (si pensi ad un portatile, che venga messo in condizioni di sospensione, spostato e riattivato dopo averlo connesso ad un'altra rete).

Moduli del programma

Data la non indifferente mole di codice necessaria, selfDHCP è stato scritto in maniera modulare, racchiudendo in singoli file sorgente in codice C funzionalità (strutture dati, funzioni, costanti simboliche, etc.) divise per campo di attinenza. Per ogni file sorgente è presente un *header* file, che permette di utilizzare le funzionalità del relativo sorgente in altre parti del programma. I moduli che compongono selfDHCP sono:

- selfdhcp_errlib — gestione degli errori e dei messaggi. È una versione leggermente modificata del file errlib.c che si trova in *Unix Network Programming*
- selfdhcp_struct — definizione delle strutture dati per il mantenimento delle liste di indirizzi MAC e IP, e delle funzioni per il loro utilizzo (ricerca, inserimento, etc.). Contiene anche funzioni di utilizzo generale, come le funzioni `ipcmp()` e `maccmp()` per il confronto di indirizzi di rete.
- selfdhcp_pthread — implementa la gestione *thread-safe* delle strutture dati, utilizzando le libpthread.
- selfdhcp_profiles — definizione delle strutture dati e delle funzioni per la gestione dei profili e delle configurazioni di rete.
- selfdhcp_capture — gestione della parte di cattura dei pacchetti, utilizzando le libpcap.
- selfdhcp_heuristics — implementazione dell'euristica per la determinazione dei parametri di rete.

I diversi moduli sono utilizzati dal programma vero e proprio, contenuto nel file main.c.

Questo provvede al controllo dei parametri passati la linea di comando, con l'ausilio della libreria libpopt. Grazie a questa è possibile gestire in poche righe la definizione dei parametri, la loro scansione e la gestione degli errori. La libreria si occupa anche di generare in maniera automatica i messaggi di help e di usage, il cui output è riportato qua sotto:

```
$ selfdhcp --help
Usage: selfdhcp [OPTION...]
  -i, --iface=device         Interface to be used
  -d, --daemon               Daemon mode
  -c, --consoleoutput        Don't do any configuration, just give a
                             configuration in plain text
```

```
  -t, --timeout=timeout      Capture timeout (minutes) without receiving
                             packets (default 1 min)
  -D, --debug                Displays very verbose debug messages
  -p, --profile=filename     Use filename as profile source (default
                             $HOME/.selfdhcp_profiles)
  -v, --verbose              Verbose mode
  -V, --version              Displays version

Help options:
  -?, --help                 Show this help message
  --usage                    Display brief usage message


$ selfdhcp --usage
Usage: selfdhcp [-i|--iface device] [-d|--daemon] [-c|--consoleoutput]
        [-t|--timeout timeout] [-D|--debug] [-p|--profile filename]
        [-v|--verbose] [-V|--version] [-?|--help] [--usage]
```

Il programma principale si occupa poi di definire una serie di parametri, in base anche agli argomenti ricevuti dall'esterno: l'interfaccia di rete da utilizzare, se il programma deve essere eseguito come *daemon* o no, la presenza o meno di messaggi verbosi e di debug, l'utilizzo o meno di un file di profili (e in questo caso si occupa anche dell'apertura e del caricamento in memoria del file).

In questa sede vengono anche definite le uniche due variabili globali dell'intero programma: una, `prog_name`, contiene il nome del programma stesso (preso da `argv[0]`, e privato delle informazioni sul path), l'altra, `program_flags`, è un intero che contiene diversi flag, definiti da opportune maschere tramite direttive di `#define` contenute nel file selfdhcp_errlib.h.

Dopo la definizione dei parametri, vengono inizializzate le strutture dati (liste di MAC e IP e vettore di configurazioni). Infine, se il programma deve girare come *daemon* vengono eseguite le opportune operazioni preliminari (disassociazione dal terminale, spostamento della directory di lavoro, chiusura dei file descriptor, redirezione degli stream standard, apertura di syslog). A questo punto, il programma si divide un due thread separati, che eseguono rispettivamente le operazioni di cattura dei pacchetti e di euristica. Per la gestione dei thread sono state scelte le libpthread, che sono state usate anche per la gestione della mutua esclusione nelle strutture dati.

Il programma è dotato di un singolo[5] punto di ritorno normale (`return 0` o `exit(0)`), dopo il ritorno con successo del thread di euristica. Per quasi tutti gli altri punti di ritorno ci si è affidati alle funzioni contenute nel file selfdhcp_errlib.c, che prevedono l'uscita forzata dal programma (`exit(1)`).

[5] se si escludono quelli per la visualizzazione dell'help e della versione del programma

selfdhcp_errlib

In questo file sono contenute le funzioni per la gestione di errori e messaggi. La maggior parte delle funzioni è identica a quelle definite nel libro *Unix Network Programming* di R. Stevens; unica modifica, il controllo sul modo di funzionamento del programma (*daemon* o no) è fatto non più direttamente su un intero, ma sulla variabile `program_flags` messa in AND binario con la costante simbolica `IS_DAEMON_PROC`.

Sono poi state aggiunte tre funzioni: la `dbg_msg()` è identica alla preesistente `err_msg()`, salvo che visualizza il proprio messaggio solo se è stata selezionata la modalità di debug; analogamente la `inf_msg()` è utilizzata per visualizzare le informazioni legate alla modalità *verbose*. La `display_options()` è utilizzata, sempre in modalità di debug, per visualizzare quali dei possibili flag sono attivi.

selfdhcp_struct

Il file selfdhcp_struct.c e il suo *header* sono un po' la base del programma, in quanto attorno ad essi sono state definite quasi tutte le altre parti. Per il funzionamento di selfDHCP è stata progettata una struttura dati *ad-hoc* piuttosto complessa, che permette di archiviare e richiamare i dati secondo diversi criteri.

I requisiti che hanno portato a questa specifica implementazione sono stati

- sapere quante volte è stato visto un certo indirizzo;
- sapere con quanti e quali altri indirizzi (chiamati *relatives*, o “parenti”) è stato associato;
- poter leggere e scrivere sui dati da due (o più) thread separati, senza che questi possano interferire;
- poter accedere ai dati secondo diversi criteri (indirizzo più visto, o con più “parenti”, o in ordine numerico);
- non è necessaria la cancellazione di elementi, ma solo l'inserimento;

La struttura risultante è composta di due tipi di elementi. Il primo è la struttura `lista_t`:

```
typedef struct lista {
     void *data;
     struct sublista *relatives;
     int datatype;
     unsigned int relnum;
     unsigned long howmany;
     pthread_mutex_t semafori[5];
     int numreaders, numwriters;
```

```
        struct lista *next;
        struct lista *nextnum;
        struct lista *nextrel;
        } lista_t;
```

Si tratta di una lista, che esiste un duplice copia (una per indirizzi MAC e una per indirizzi IP).Poiché la maggior parte dei dati è comune per i due tipi, è stata utilizzata una sorta di meta-struttura, in cui gli indirizzi veri e propri sono referenziati dal puntatore `void *data`; per riconoscere il tipo di dati puntati, il campo `datatype` contiene l'indicazione, scelta tra le opportune direttive di `#define`:

```
#define IP_TYPE       1
#define MAC_TYPE      0
```

Il campo `*relatives` punta ad una lista di “parenti”, la quale è però composta da strutture di un altro tipo, che verrà esaminato a breve.

Gli interi `howmany` e `relnum` indicano rispettivamente quante volte è stato visto il singolo indirizzo, e quanti “parenti” ha. Seguono alcune variabili (`semafori[]`, `numreaders` e `numwriters`) per l'implementazione *thread-safe* della lista, di cui si parlerà più diffusamente nella sezione dedicata alla libreria selfdhcp_pthread. Infine vengono tre puntatori, `*next`, `*nextnum` e `*nextrel`, che servono per collegare gli elementi della lista: `*next` per l'ordinamento numerico, `*nextnum` per numero di occorrenze, e `*nextrel` per numero di “parenti”.

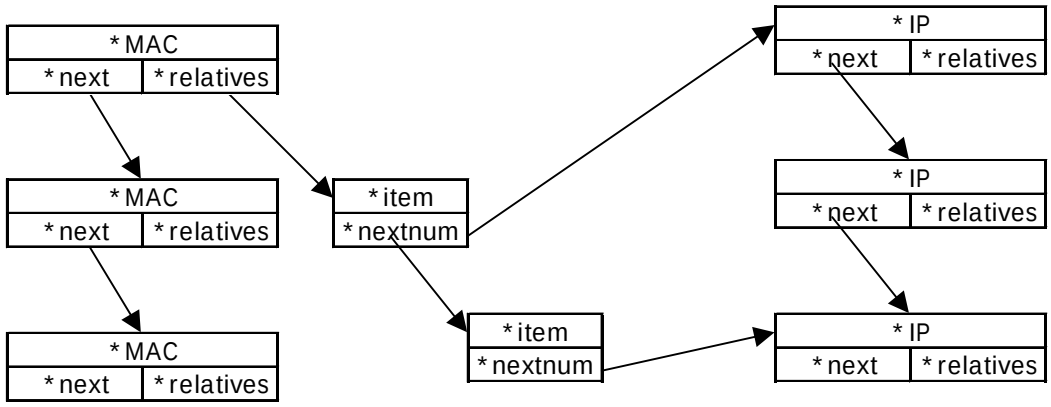
Il secondo tipo di struttura è fatto per contenere la lista dei “parenti” di un certo indirizzo. Per evitare inutili (e difficili da gestire) duplicazioni di dati, il riferimento al “parente” rimanda alla sua *entry* nella lista opposta. La struttura `sublist_t` è:

```
typedef struct sublista {
     lista_t *item;
     unsigned long howmany;
     pthread_mutex_t semafori[5];
     int numreaders, numwriters;
     struct sublista *nextnum;
     } sublist_t;
```

Oltre al suddetto puntatore `*item` all'*entry* del parente, c'è un intero `howmany` che indica quante volte è comparsa la coppia, il solito set di variabili per la gestione *thread-safe* e un unico puntatore `*nextnum` per collegare la lista (l'ordinamento è per numero di occorrenze).Per questo tipo di struttura non è necessario indicare il tipo di dati

puntati, in quanto è possibile risalirvi facilmente tramite l'indicazione contenuta nel campo `*item`.

Combinando le due strutture, si ottengono due liste parallele di tipo `lista_t`, una per gli indirizzi MAC e una per gli IP. Le liste non sono linkate singolarmente, ma secondo tre diversi criteri, per cui possono essere lette in ordine diverso. Da ogni elemento di ognuna delle due liste, parte una sotto-lista di tipo `sublist_t`, che punta a elementi dell'altra lista. Uno schema molto semplificato della struttura è riportato in figura (si tiene conto di un solo ordinamento, ed è mostrata la lista di parenti per un singolo indirizzo MAC, mentre ogni indirizzo, MAC o IP, ha la sua lista di parenti):



La struttura dati complessiva è indubbiamente complicata, ma serve il suo scopo in maniera soddisfacente. Le due liste partono da due record *dummy* vuoti, chiamati spesso `head` all'interno del programma, i quali a loro volta sono contenuti in un array. Per accedere ad entrambe le liste sarà quindi sufficiente disporre di un doppio puntatore `lista_t **liste`, come infatti viene fatto dalla maggior parte delle funzioni del programma, mentre per accedere ai due singoli elementi si possono utilizzare le stesse costanti simboliche utilizzate per il campo `datatype` della struttura `lista_t`.

La libreria non definisce solo le strutture dati, ma anche un'ampia serie di funzioni per la loro manipolazione. Sono comprese funzioni di ricerca, di inserimento, di ordinamento, nonché una serie di funzioni di utilità comune. Tra tutte, le più importanti sono:

- `str2mac()` e `mac2str()`: permettono la traduzione di un indirizzo MAC da stringa a formato numerico e viceversa (in pratica sono l'equivalente delle funzioni `inet_aton()` e `inet_ntoa()` per gli indirizzi IP).
- `maccmp()` e `ipcmp()`: la prima è un semplice *wrapper* per la `memcmp()`, ma con dimensione fissa (quella dell'indirizzo MAC). La seconda confronta numericamente i due indirizzi.

- `createlists()`: viene utilizzata all'inizio del programma per inizializzare le due liste di MAC e IP.
- `coupleinsert()`: rappresenta l'interfaccia per l'inserimento di una coppia MAC-IP nelle liste. Sono utilizzate internamente molte altre funzioni, qua non riportate; si rimanda per questo alla lettura del codice sorgente, abbondantemente commentato.
- `findlevel()` e `findnext()`: permettono di cercare un indirizzo per livelli (“qual'é il secondo IP più visto? Ce ne sono altri visti altrettante volte?”). `findlevel()` trova il primo elemento di un certo livello, e può operare in due modi: per numero di occorrenze o per numero di “parenti”. `findnext()` trova, uno alla volta, tutti gli altri elementi su quello stesso livello.

selfdhcp_pthread

Questo file contiene le funzioni per l'implementazione *thread-safe* delle strutture dati. La scelta per il tipo di implementazione è caduta sul classico *readers and writers* con precedenza ai *writers*, anche in considerazione di una futura espandibilità del progetto (per ora i thread sono solo due, uno che scrive e l'altro che legge, ma non è da escludersi che future versioni necessitino di più processi in lettura).

Sono state utilizzate le libpthread, in particolare gli oggetti di mutua esclusione `mutex_pthread_t`. Ogni elemento di una struttura possiede un array di cinque di questi oggetti, più due interi per tenere conto del numero di *reader* e *writer*.

Le funzioni dichiarate in selfdhcp_pthread.c sono:

- `read_reserv()`, `read_release()`, `write_reserv()`, `write_release()`: vengono chiamate per bloccare o sbloccare un oggetto in lettura o scrittura.
- `semaphores_init()`: utilizzata alla creazione di un elemento per inizializzare i semafori.
- `daemon_init()`: come definita nel cit. *Unix Network Programming* di R. Stevens.

Sono presenti inoltre le versioni *wrapped* delle funzioni pthread utilizzate all'interno del programma.

selfdhcp_profiles

Nel file selfdhcp_profiles.c e nel suo header sono definite funzioni e strutture di supporto per due ambiti, in realtà molto simili: le configurazioni di rete e i profili. I profili sono in effetti implementati come collezioni di configurazioni di rete precedentemente sperimentate, e sono quindi strutturalmente identici alle configurazioni di rete.

La struttura di base è definita come:

```
typedef struct netconfig {
     unsigned char gatewayMAC[ETH_ALEN];
     struct in_addr gatewayIP;
     struct in_addr dns1, dns2;
     struct in_addr netmask, netaddr;
     struct in_addr myIP;
     int what_didnt_work;
     int what_is_set;
     struct netconfig *next;
     } netconfig_t;
```

All'interno possiamo individuare i vari campi che definiscono una tipica configurazione di rete ethernet/IPv4: indirizzo IP, netmask, gli indirizzi IP e MAC di un gateway, e gli indirizzi IP di due server DNS. Il campo `netaddr` è utilizzato internamente dal programma, e non fa parte della configurazione di rete vera e propria.

Gli altri campi della struttura sono due set di flag, `what_is_set`, che contiene indicazioni su quali sono le informazioni valide nella configurazione, e `what_didnt_work`, che indica invece quali sono errate (per far sì che in iterazioni successive il programma non debba rifare tutto dal principio).

Il puntatore `*next` infine è utilizzato per creare una lista di configurazioni.

La struttura di tipo `netconfig_t` è utilizzata in due modi, a seconda che debba contenere una configurazione di lavoro su cui sta attivamente lavorando il programma, o un profilo caricato in memoria da file.

Nel primo caso, è definito un array statico di due elementi, in genere denominato `net_conf` e indirizzato tramite due costanti simboliche: il primo di questi contiene l'attuale configurazione di lavoro, mentre il secondo contiene una eventuale precedente configurazione rivelatasi non funzionante. I due elementi dell'array sono indicizzati tramite due apposite costanti simboliche:

```
#define NETCONF_CURR  0
#define NETCONF_BACK  1
```

Il confronto tra le due, insieme all'utilizzo dei flag contenuti in `what_didnt_work`, permette nelle successive iterazioni di raffinare la configurazione fino a ottenerne una perfettamente funzionante. Le costanti simboliche utilizzate sono le seguenti:

```
#define MY_MAC        0x01
#define GW_MAC        0x02
```

```
#define NETMASK       0x04
#define GW_IP         0x08
#define MY_IP         0x10
#define DONT_KNOW     0x80
```

Ovviamente tali flag hanno diverso significato a seconda del set di flag in cui si trovano.; la costante `DONT_KNOW` inoltre ha senso solamente nella configurazione di backup.

Per quanto riguarda i profili di rete, la struttura è invece utilizzata come elemento di una lista non ordinata, tramite il puntatore `*next`.

Per la gestione di questo tipo di struttura sono definite una serie di funzioni, le cui principali sono:

- `netconf_dropcurrent()`: sposta il contenuto dalla configurazione di rete attuale `NETCONF_CURR` a quella di memoria `NETCONF_BACK`, poi azzera la configurazione attuale. Questa funzione è utilizzata ogni volta che una configurazione non supera il test di funzionamento.
- `read_profiles()`, `read_profilentry()`, `write_profile()`, `write_profilentry()`: sono utilizzate rispettivamente per l'acquisizione da file di una lista di profili, o per la scrittura di tale lista su file.
- `find_profile()`: cerca in una lista se esiste o meno un profilo, identificato tramite l'indirizzo MAC del gateway, che è supposto essere univoco e quindi valido come chiave di riconoscimento.

selfdhcp_capture

La parte di selfDHCP che si occupa della cattura dei pacchetti è relativamente semplice, poiché affida il suo funzionamento sulle librerie libpcap. Queste consentono un'elevata semplicità d'uso, soprattutto per quanto riguarda la possibilità di impostare dei filtri selettivi su protocolli e indirizzi dei pacchetti da catturare, e dovrebbero al contempo garantire una maggiore portabilità.

Come visto in precedenza, il thread di cattura si occupa inizialmente dell'inizializzazione delle strutture interne (buffer circolare, strutture per il passaggio dei parametri, etc), poi si divide in due sotto-thread. Tutte le funzioni, le strutture e le definizioni necessarie sono contenute in questo modulo.

Poiché sono state utilizzate le libpthread per la gestione dei processi, è stato necessario, per comodità, dichiarare un nuovo tipo di struttura atta a contenere i parametri da passare alle varie funzioni che costituiscono i thread, e che per implementazione richiedono il passaggio di un singolo argomento di tipo `**void` (si

veda in merito `man pthread_create`).

Il buffer circolare è implementato in maniera molto semplice, con un vettore di strutture di tipo `couple_t` (che banalmente contengono una coppia MAC/IP). Due variabili globali per il modulo mantengono la sincronizzazione tra produttore e consumatore, senza bisogno di introdurre variabili di mutua esclusione.

Il thread di cattura è impostato in modo da ammettere solo il traffico di tipo ARP, RARP o IP (che comprende poi a sua volta i protocolli di livello superiore, come ICMP, TCP e UDP). È escluso il traffico IP multicast (che oltre a non essere utile per la configurazione costituirebbe un non indifferente disturbo nella ricerca della netmask e dell'indirizzo di rete), e il traffico ethernet broadcast (diretto all'indirizzo MAC FF:FF:FF:FF:FF:FF), in quanto genererebbe delle coppie con IP autentici, ma MAC broadcast, che sarebbero problematici da gestire per l'individuazione dell'indirizzo MAC del gateway.

Il sotto-thread di cattura si occupa anche di individuare eventuale traffico DNS (riconosciuto dalla porta di destinazione o sorgente del livello TCP o UDP). Tale informazione non è strettamente necessaria, ma può far parte di una configurazione ottimizzata per la rete specifica.

Il sotto-thread di catalogazione banalmente estrae dal buffer circolare le coppie, e provvede a inserirle nelle opportune liste tramite le funzioni fornite dal modulo selfdhcp_struct.c

Le principali funzioni contenute in selfdhcp_capture.c sono:

- `capture_start()`: costituisce la prima parte di inizializzazione delle strutture e delle libpcap.
- `insert_couple()`, `extract_couple()`, `init_buffer()`: le prime due provvedono rispettivamente all'inserimento e all'estrazione di una coppia dal buffer circolare; la terza inizializza il buffer e le variabili globali usate come indici di posizione.
- `packet_process()`, `catalog_packet()`: costituiscono i due sotto-thread di cui è composto il thread di cattura principale.

selfdhcp_heuristics

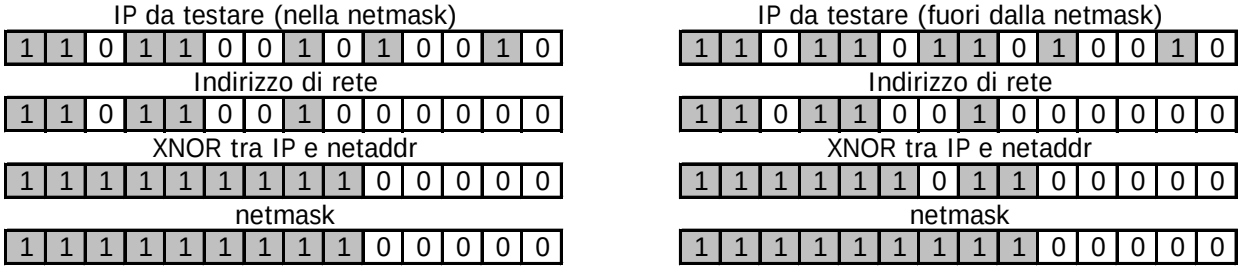
Questo modulo costituisce indubbiamente la parte più fondamentale di selfDHCP, in quanto in esso sono contenute tutte le funzioni che permettono di risalire dalle coppie di indirizzi precedentemente catalogati alla configurazione di rete.

Come spiegato in precedenza, il thread di euristica si attiva ad intervalli regolari, la cui durata è specificata dalla costante `TIME_GRANULARITY`, e per un numero di volte che dipende dal timeout impostato dall'utente (il default è di 5 minuti), oppure finché il programma non viene terminato se si è in modalità *daemon*.

Allo scadere del timeout, selfDHCP controlla in un ordine ben determinato quali parametri della configurazione non sono stati ancora determinati, sfruttando il flag `what_is_set` della configurazione attuale. Se alcuni parametri non sono ancora stati stabiliti, provvede a chiamare le opportune funzioni per la ricerca.

Facciamo prima una breve lista delle principali funzioni di appoggio dichiarate in questo modulo, per poi descrivere il funzionamento dei vari passi dell'euristica vera e propria:

- `heuristics_start()`: è la funzione principale del thread di euristica. In essa vengono eseguite alcune inizializzazioni (ad esempio delle strutture di supporto per le libnet, di cui si dirà in seguito), quindi inizia il loop di euristica, richiamando a tempo debito le diverse funzioni e facendo il controllo dei parametri già determinati.
- `is_not_relative()`: dato un indirizzo IP e l'indirizzo MAC del gateway, stabilisce se questo indirizzo sia o meno un parente del gateway. Per far questo semplicemente scansiona la lista di parenti del MAC del gateway, alla ricerca dell'IP indicato.
- `is_in_netmask()`: stabilisce se un dato IP appartiene o meno alla rete di cui si sia trovato l'indirizzo e la netmask. Per fare questo, l'indirizzo da testare è messo in bitwise XNOR[6] con l'indirizzo di rete; se il risultato è uguale alla netmask, l'indirizzo da testare è all'interno della rete, altrimenti va considerato esterno. Un esempio è riportato in figura:



- `netmask_adjust()`: serve ad "assestare" la netmask dopo che è stata trovata dall'opportuna funzione euristica. Ricordiamo che la netmask deve essere costituita da una serie di 1 seguita da una serie di 0, ma i valori non possono essere mai inframmezzati. La funzione `netmask_adjust()` non fa altro che contare i bit messi a 1 finché non trova uno 0, e quindi esegue un opportuno numero di shift per azzerare tutta la rimanente parte dell'indirizzo.
- `arp_req()`: è la funzione che si occupa di costruire le ARP request, iniettarle in

[6] ricordiamo che la funzione XNOR restituisce 1 se i due bit sono entrambi a zero o entrambi a uno, 0 altrimenti

rete, preparare il filtro per la ricezione delle ARP reply, far partire il sotto-thread di cattura che intercetta la risposte e terminarlo quando scade il timeout stabilito per la ricezione.

- `arp_capture()`: costituisce il corpo del sotto-thread di cattura delle ARP reply.

Il primo e più semplice passo dell'euristica consiste nell'individuazione dell'indirizzo MAC del gateway; di ciò si occupa la funzione `find_gatewayMAC()`. Questo dovrebbe essere l'unico indirizzo MAC della rete che è stato catalogato con più di un parente IP, ed è quindi sufficiente una chiamata alla funzione `find_level()` della libreria selfdhcp_struct, specificando di eseguire la ricerca sulla lista di indirizzi MAC e in base al numero di parenti. Bisogna poi controllare che l'indirizzo ritornato abbia effettivamente più di un parente (potrebbe essersi verificato il caso in cui siano stati catturati solo pacchetti da e verso la rete interna, in cui il gateway non ha avuto parte, oppure potrebbe essere stato contattato un singolo host esterno).

Una volta in possesso del MAC del gateway, il passo successivo è trovare la netmask e l'indirizzo della rete. La funzione che svolge questa mansione è la `find_netmask()`.Vengono cercati tra gli indirizzi IP interni quello numericamente minore e quello numericamente maggiore, scartando tramite la `is_not_relative()` gli IP parenti del MAC del gateway, che sono sicuramente esterni. I due indirizzi vengono messi in bitwise AND, e il risultato viene raffinato dalla `netmask_adjust()` per ottenere una netmask valida. Questo passaggio è quello più suscettibile di errori, perché dipende in maniera considerevole dal numero di diversi IP interni che sono stati catalogati. L'osservazione empirica dei risultati ha dimostrato come sia difficile ottenere una netmask uguale a quella effettiva della rete, anche se in condizioni normali si ottiene di solito una netmask più restrittiva. Dopo aver trovato la netmask, l'indirizzo di rete viene trovato semplicemente mettendo in bitwise AND un qualunque indirizzo interno con la netmask appena trovata.

A questo punto è possibile cercare l'indirizzo IP del gateway, funzione assolta dalla `find_gatewayIP()`. Purtroppo è di solito improbabile avere l'indirizzo effettivo già catalogato tra i parenti del gateway, per cui non si può semplicemente cercare quale di questi sia all'interno della netmask. La soluzione adottata è di mandare tramite la `arp_req()` ARP request a tutti gli indirizzi nella netmask, a partire dall'indirizzo di rete e incrementando ogni volta di uno. In realtà le ARP request inviate sono 5 (numero definito nella costante simbolica ARP_REQ_2SEND), per minimizzare il rischio che una richiesta (o peggio una risposta) vada persa. Le ARP request vengono costruite utilizzando come indirizzi sorgente quelli di un host interno scelto a caso, in modo da evitare di genare traffico anomalo.

In modo analogo funziona la `find_myIP()`, che è preposta alla ricerca dell'ultimo

parametro di rete necessario, ovvero un IP libero. Partendo dall'indirizzo di rete, ed aumentando ogni volta di uno, si controlla se l'IP sotto analisi non è già stato catalogato. In caso di esito negativo della ricerca, si procede ad inoltrare ARP request tramite la `arp_req()`, e se non otteniamo risposta, abbiamo conferma che l'IP è libero, almeno al momento.