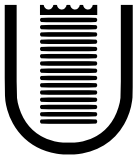


Università degli Studi di Roma "Tor Vergata"

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria informatica

Tesi di Laurea magistrale

Gestione della mobilità verticale su base applicazione: progetto e realizzazione per la piattaforma Android

Candidato:
Alessio Bianchi
Matricola 0130525

Relatore:
Prof. Francesco Lo Presti

Correlatori:
Prof. Stefano D. Salsano
Ing. Marco Bonola

Anno Accademico 2009-2010

Sommario – Abstract

La mobilità verticale, ossia tra reti di accesso eterogenee, sta assumendo sempre maggiore importanza. Una soluzione che implementa questo paradigma è UPMT (Universal Per-application Mobility management using Tunnels), che consente all'utente di impostare politiche di handover su base applicazione. L'obiettivo del lavoro di tesi è progettare e implementare i componenti necessari per consentire a UPMT di gestire la mobilità su base applicazione. Sarà necessario, quindi, identificare le applicazioni e associare ad esse le connessioni di rete, e fornire alla GUI di controllo gli strumenti per determinare l'instradamento delle connessioni sui tunnel in base a politiche personalizzabili. Si vuole, inoltre, effettuare il porting dei componenti sviluppati, insieme agli altri componenti di basso livello esistenti, sulla piattaforma mobile Android, in preparazione del futuro porting della GUI di controllo come applicazione nativa Android.

Vertical mobility, meaning among heretogeneous access networks, is lately gaining importance. A solution implementing this paradigm is UPMT (Universal Per-application Mobility management using Tunnels), which allows the user to define handover policies on a per-application basis. The goal of this work is to design and implement the components needed to allow UPMT to take per-application mobility decisions. It will be necessary to identify the applications and match them with their traffic flows, as well as to provide the control GUI with the tools to decide whether and how to forward the connections over tunnels based on customizable policies. Additionally, we want to port the implemented components, along with the other existing low-level ones, to the Android mobile platform, in preparation of the future porting of the control GUI as an Android native application.

Ringraziamenti

Alla mia famiglia.

Indice

1 La mobilità verticale

1.1 Definizioni

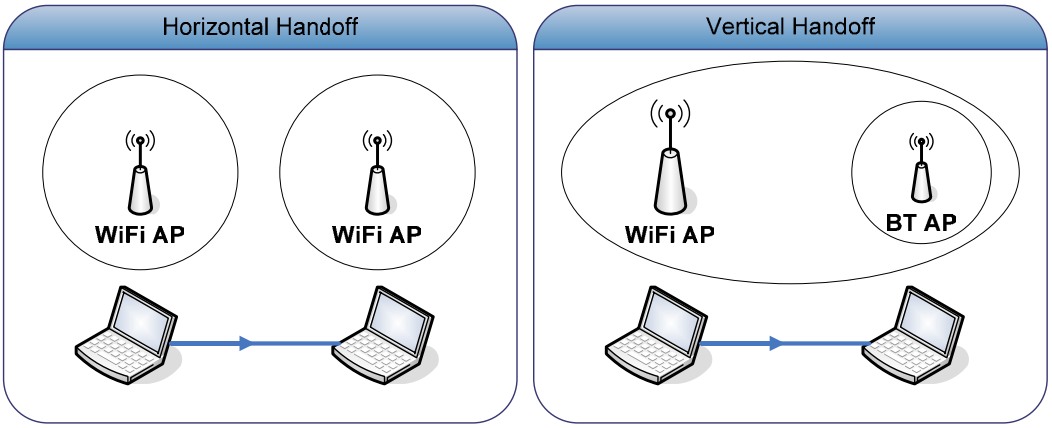
Con l'espressione *mobilità verticale* ci si riferisce alla possibilità, per un nodo di rete, di passare dinamicamente da una tecnologia per reti di accesso a un'altra senza che l'utente percepisca interruzioni di servizio. Tale passaggio è definito *handover verticale* (vedi figura 1.1), il cui scopo è, appunto, consentire la mobilità del nodo. Ad esempio, un notebook dotato di connettività wifi e 3G e opportunamente configurato, consentirebbe di utilizzare principalmente la rete wifi e, in caso di sua indisponibilità, la rete 3G. In tal modo si combinano i vantaggi dell'alta velocità offerta dalle reti wifi e dell'ubiquità della rete 3G. Questo tipo di handover si contrappone all'handover orizzontale, in cui un nodo si sposta da un punto di accesso a un altro, ma utilizzando un'unica tecnologia di accesso. Un esempio di handover di questo tipo è quello effettuato dai cellulari quando si spostano tra diverse BTS.

1.2 Visione ed esempi applicativi

Molte situazioni che si verificano nella quotidianità potrebbero trarre notevole beneficio da un uso diffuso della mobilità verticale. Consideriamo ad esempio un utente in possesso di uno smartphone dotato di connettività wifi e 3G, attualmente connesso tramite rete wireless. Il telefono sta compiendo un trasferimento dati in background, ad esempio il download di applicazioni, e contemporaneamente l'utente è impegnato in una videoconferenza. Se l'utente uscisse fisicamente dal raggio di copertura della rete wireless, causandone la disconnessione, egli desidererebbe poter continuare la video conferenza senza interruzioni veicolandone il traffico tramite la rete 3G. Nel contempo, però, il download in background dovrebbe interrompersi, sia per motivi di tariffazione del traffico sia per motivi di prestazioni.

Data la recente grande diffusione di dispositivi mobili evoluti, il supporto alla mobilità nelle reti IP sta assumendo sempre maggiore importanza nel mondo della ricerca e da parte dei provider. Al giorno d'oggi le reti cellulari offrono connessioni dati basate su IP ad alto tasso di trasferimento, la copertura wifi è capillare nelle abitazioni, nelle aziende e in punti strategici delle città (aeroporti, stazioni ecc.) tramite hotspot. Altre tecnologie senza fili come il WiMax in ambienti esterni o il Bluetooth nelle PANs (Personal Area Networks)

Figura 1.1 Handover orizzontale e verticale.



completano lo scenario. I recenti e versatili dispositivi di terza generazione sono ora in grado di connettersi contemporaneamente a varie tecnologie di accesso con e senza fili.

1.3 Soluzioni per la mobilità verticale

Negli ultimi anni, sono state proposte varie soluzioni per la mobilità che supportano gli handover verticali a differenti livelli della pila protocollare, dal livello data-link fino al livello applicativo: l'articolo [1] ne presenta una panoramica. Una soluzione di mobilità deve supportare alcuni requisiti, riportati nella sezione 1.4. La soluzione UPMT aggiunge un ulteriore requisito, ossia la possibilità di trasportare contemporaneamente applicazioni diverse su tecnologie di rete diverse e di prendere decisioni sull'handover distinte, su base applicazione. Moonjeong, Meejeong e Hyunjeong, nell'articolo [2], affermano che questo requisito è stato trascurato o non pienamente realizzato nella maggior parte delle soluzioni disponibili. Nella soluzione proposta in tale articolo è richiesto che sia il terminale mobile, da questo momento chiamato "Mobile Host" o "MH", che l'altro capo della comunicazione, da questo momento chiamato "Correspondent Host" o "CH", debbano entrambi cooperare ed essere equipaggiati con la suddetta tecnologia. La soluzione UPMT invece permette di soddisfare questo requisito anche se solo il Mobile Host è dotato di essa.

1.4 Requisiti per le soluzioni di gestione della mobilità

Una soluzione di mobilità dovrebbe idealmente soddisfare i seguenti requisiti, delineati in [3]:

1. Non dovrebbe richiedere alcun supporto particolare né da parte delle reti di accesso, né da parte degli operatori di rete. In ogni modo, questi ultimi dovrebbero poter integrare in modo semplice il servizio di mobilità nella loro infrastruttura di rete.

2. Non dovrebbe richiedere alcuna modifica al Correspondent Host né alle applicazioni in esecuzione su di esso. Tutti i terminali esistenti dovrebbero essere in grado di interagire

con i Mobile Hosts in movimento. D'altro canto, se anche il Correspondent Host o un'applicazione su di esso supporta la soluzione di gestione della mobilità, dovrebbe sfruttarla per fornire una soluzione più efficiente.

3. Non dovrebbe richiedere alcuna modifica alle applicazioni in esecuzione sul Mobile Host. Tuttavia, dovrebbe essere semplice sviluppare nuove applicazioni o estendere quelle esistenti per sfruttare appieno le capacità offerte dalla soluzione di gestione della mobilità.

4. Dovrebbe permettere di attuare handover su base applicazione basandosi su un insieme di criteri, ad esempio il costo e vari parametri di qualità del servizio (QoS/QoE) come throughput, perdite, ritardo/jitter, sicurezza ecc.

5. Dovrebbe supportare tutti i tipi di applicazione, sia quelle basate su UDP che quelle basate su TCP.

6. Dovrebbe essere compatibile con il NAT traversal. Gli utenti dovrebbero essere in grado di muoversi da una rete di accesso all'altra anche quando una o entrambe usano indirizzi IP privati e sono dietro un NAT. I terminali che non sono connessi dietro un NAT o che sono nella stessa rete di accesso dovrebbero essere in grado di comunicare direttamente, quando possibile.

7. Dovrebbe offrire buona scalabilità, distribuendo le risorse e il carico di elaborazione su più nodi di rete.

8. Dovrebbe essere possibile nascondere la posizione attuale e i movimenti dell'utente al Correspondent Host, in modo da preservare la privacy dell'utente.

9. Nello spostamento tra reti di accesso, la segnalazione di supporto alla mobilità dovrebbe essere inviata sulla rete di destinazione, dato che la precedente potrebbe essere diventata improvvisamente non disponibile; in tal caso è necessario realizzare l'intera procedura di handover sulla nuova rete. Questo approccio prende il nome di "forward handover" o "break before make". Al contrario, se la vecchia rete è ancora disponibile, la disponibilità di entrambe può essere sfruttata in modo da assistere e velocizzare l'intera procedura ("make before break").

10. L'handover verticale deve essere il più veloce possibile: l'utente, infatti, non dovrebbe percepire nessuna interruzione di servizio. Se non è possibile nascondere completamente l'effetto dell'handover, il disturbo al servizio dovrebbe essere minimizzato.

1.5 UPMT: Universal Per-application Mobility management using Tunnels

Questo lavoro di tesi si basa su una soluzione per la gestione della mobilità chiamata UPMT (Universal Per-application Mobility management using Tunnels) che soddisfa tutti i requisiti sopracitati e, in particolare, il requisito 4 sulla possibilità di eseguire handover di ciascuna applicazione e perfino di ogni singola connessione.

UPMT introduce un meccanismo di trasporto di pacchetti IP basato su un insieme di tunnel instaurati tra il Mobile Host e un cosiddetto "Nodo di Ancoraggio" (da questo momento chiamato "Anchor Node" o "AN") ovvero un'entità specifica di supporto alla comunicazione. Questo fornisce un servizio di "NAT virtuale" al Mobile Host indipendentemente dalle reti di accesso utilizzate e dai NAT fisici attraversati, così da permettere l'interazione con tutti i terminali e con le applicazioni legacy.

La soluzione considerata si colloca al livello applicativo della pila protocollare. Pregi e difetti delle soluzioni di mobilità operanti a livello applicativo sono trattati in [1]. Soluzioni a livello rete che hanno ricevuto molta attenzione sono Mobile IPv4 e Mobile IPv6, ma entrambe richiedono esplicito supporto da parte della rete e dei terminali. Proprio questo aspetto costituisce, ad oggi, un grosso vantaggio delle soluzioni di livello applicativo. L'introduzione di IPv6, con i suoi meccanismi di mobilità, porterà sicuramente dei benefici, ma solo sul medio o lungo periodo. Si noti che la maggior parte dei meccanismi di UPMT che verranno in seguito descritti hanno il loro corrispondente in Mobile IPv6.

2 Una soluzione per la mobilità verticale: UPMT

2.1 Scenari di riferimento

UPMT supporta molteplici scenari, differenti per topologia di rete. Questi coinvolgono varie reti di accesso (gestite da operatori distinti e sulle quali non è necessaria alcuna modifica), vari NAT, e flussi applicativi TCP o UDP di applicazioni sia legacy che "UPMT-aware" (cioè realizzate con esplicito supporto all'interazione con UPMT).

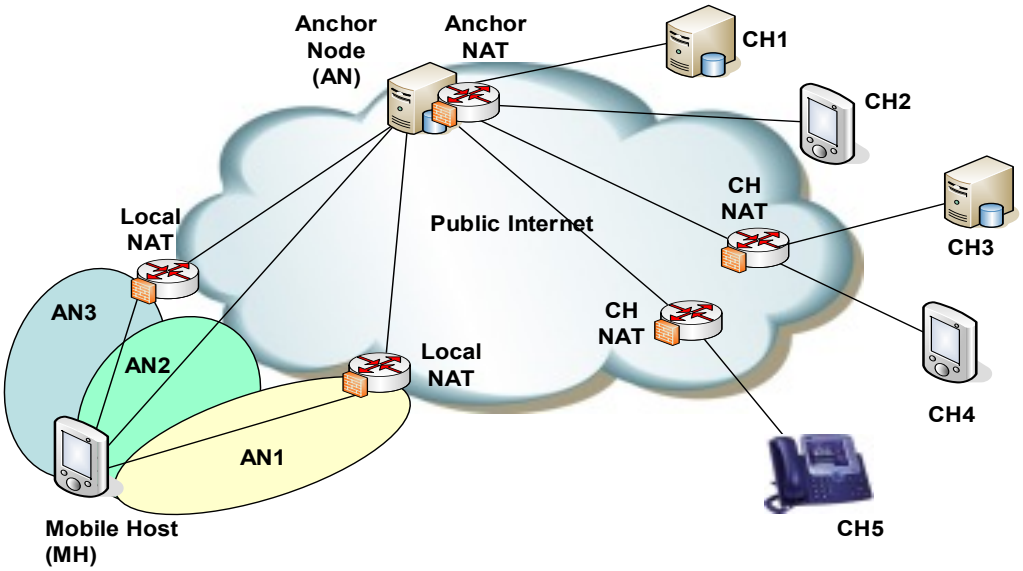
2.1.1 Anchor Node singolo

Questo scenario, definito anche *UPMT Centralized AN Mode* o *UPMT-CAM*, è illustrato in figura 2.1. Un Mobile Host può essere connesso tramite differenti reti di accesso, sia via cavo che wireless (ad es. wifi, Bluetooth, GPRS/EDGE, 3G/HDSPA, WiMax, Ethernet). La maggior parte di queste reti fornisce un indirizzo IP privato al Mobile Host ed è connessa ad Internet tramite NAT box (Local NAT).

Il Mobile Host si connetterà ad un Correspondent Host che può essere su un indirizzo IP pubblico (ad es. CH1 e CH2) o dietro a un NAT box (CH-NAT, ad es. CH 3-4-5). Il Correspondent Host può essere sia un "server", ad esempio un web server, o un "client", ad esempio un client VoIP. Il servizio di mobilità è reso disponibile da un Anchor Node che fornisce al Mobile Host un secondo servizio di NAT (Anchor-NAT), che rappresenta un elemento fondamentale nella nostra architettura di mobilità. L'idea è che il Mobile Host accederà sempre ad Internet attraverso il NAT dell'Anchor Node. La procedura di gestione della mobilità UPMT permetterà al Mobile Host di scegliere il canale di comunicazione verso l'Anchor Node, se necessario anche in modo separato per ciascun flusso/applicazione da supportare.

Si noti che il meccanismo UPMT può essere visto dal Mobile Host (e dalle applicazioni in esecuzione su di esso) esattamente come un servizio NAT. In linea di principio, tutte le applicazioni che possono essere eseguite dietro NAT box possono farlo anche usando UPMT (è infatti possibile controllare il tipo e il comportamento del NAT nell'Anchor Node). In particolare UPMT può fornire tutti i "servizi di connettività" che sono offerti tramite NAT. Nel modello descritto ci focalizzeremo sul "Port Address Translation" (PAT) o sul "Network

Figura 2.1 Scenario UPMT con Anchor Node singolo.



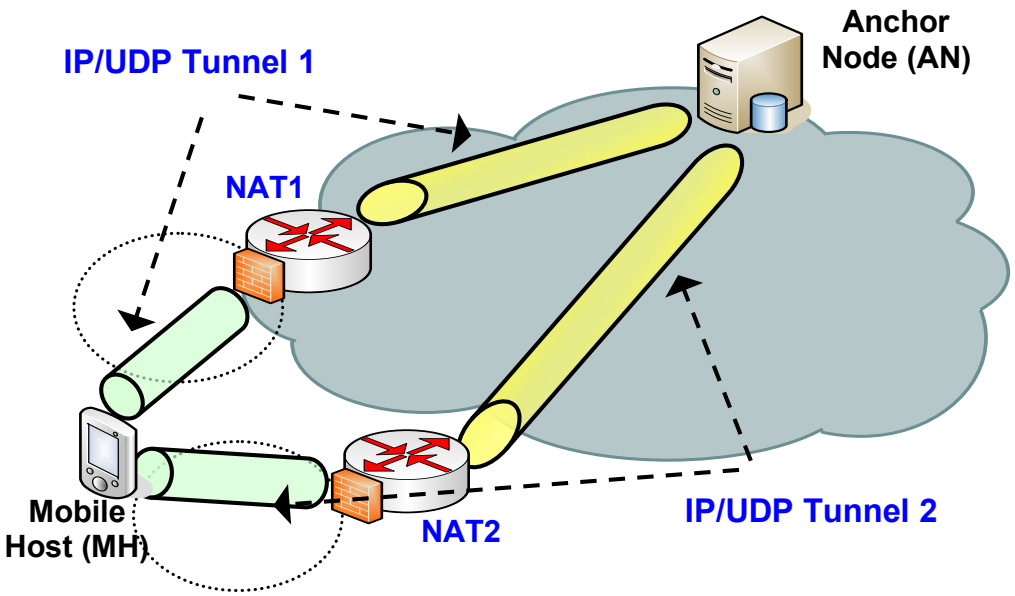
Address Port Translation" (NAPT), ma è possibile realizzare anche il cosiddetto "NAT statico" o "port forwarding".

I problemi per la mobilità possono derivare dai local NAT illustrati in figura 2.1, ovvero i NAT nel percorso tra la rete di accesso locale e l'Anchor Node, che non sono sotto il controllo della soluzione UPMT. Questi NAT locali (che possono essere anche associati a funzionalità di firewall) possono limitare la connettività che può essere offerta su una data rete d'accesso locale. Questo è esattamente ciò che accade oggi quando si accede ad Internet attraverso un ISP pubblico o in una LAN aziendale: agli utenti generalmente non è consentito eseguire tutte le loro applicazioni, alle quali può essere bloccato anche del tutto l'accesso ad Internet. Il meccanismo UPMT necessita di "testare" le differenti reti di accesso e i relativi NAT locali/firewall per poi utilizzare la connettività delle sole reti d'accesso disponibili.

L'architettura proposta è raffigurata in figura 2.2. Come già accennato prima, il Mobile Host è fornito di due (o più) interfacce di rete, connesse attraverso diverse reti di accesso. Ogni interfaccia ha un indirizzo IP che può essere pubblico o privato. Il Mobile Host instaurerà un tunnel con l'Anchor Node per ciascuna delle interfacce che può voler usare per l'invio e la ricezione di dati. Il flusso di dati verso qualsiasi Correspondent Host è incapsulato ed inviato all'Anchor Node all'interno di uno dei tunnel. Allo stesso modo, i dati provenienti da un Correspondent Host sono inviati dall'Anchor Node in uno dei tunnel. In UPMT l'incapsulamento effettuato dal Mobile Host e dall'Anchor Node è di tipo UDP/IP, ma possono essere usati anche diversi meccanismi di tunnelling. L'idea di avere differenti tunnel potenzialmente attivi nello stesso istante tra il Mobile Host e l'Anchor Node è simile all'approccio "Multiple Care Of Address" di IPv6.

Il Mobile Host invierà i pacchetti di dati nel tunnel appropriato in base ad una tabella di inoltro per-applicazione (da questo momento chiamata "Per-Application Forwarding Table" o

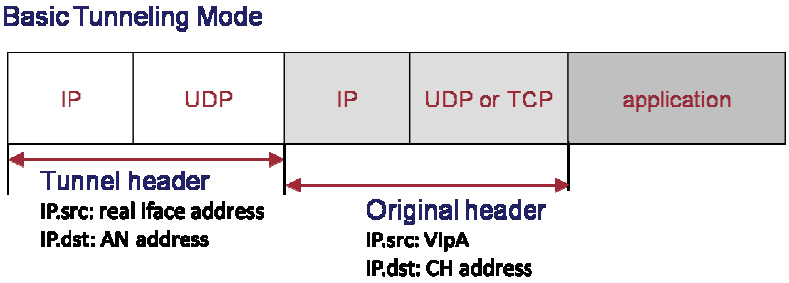
Figura 2.2 Interazione tra Mobile Host e Anchor Node.



"PAFT"). Ogni voce è composta dalla quintupla: protocollo, indirizzo IP sorgente, indirizzo IP destinazione, porta TCP/UDP sorgente e porta TCP/UDP destinazione. In pratica, la PAFT associa un socket al rispettivo tunnel. La creazione della PAFT è un aspetto critico: è facile per le applicazioni UPMT-aware che apriranno esplicitamente i loro socket con una specifica API, mentre è necessario introdurre un meccanismo di identificazione e classificazione dei socket per le applicazioni legacy in modo da associare i pacchetti al process-ID dell'applicazione. L'Anchor Node ha un'importanza cruciale nell'architettura proposta:

1. L'Anchor Node agisce come un *tunnel broker* per il Mobile Host. Durante la procedura di associazione, un indirizzo IP virtuale, chiamato *Virtual IP Address* o *VIPA*, è assegnato al Mobile Host. Questo VIPA identifica il Mobile Host per l'Anchor Node indipendentemente dal particolare tunnel attraverso il quale viene ricevuto il traffico. Ogni tunnel è univocamente identificato dall'Anchor Node attraverso il "tunnel ID" composto dalla coppia TID $=<$ indirizzo IP sorgente, porta UDP sorgente $>$. Vengono chiamati indirizzo *sorgente* e porta *sorgente* perché sono l'indirizzo e la porta dei pacchetti UDP entranti, ossia ricevuti dall'Anchor Node, che "aprono" il tunnel. L'Anchor Node invierà i pacchetti di risposta su questo tunnel se l'indirizzo IP di destinazione è uguale all'indirizzo IP sorgente interno (vedi figura 2.3) dei pacchetti ricevuti (il VIPA) e la porta UDP di destinazione è uguale alla porta UDP sorgente interna dei pacchetti ricevuti, in modo da seguire l'approccio del symmetric NAT traversal. Si noti che quando il tunnel attraversa un NAT l'indirizzo IP sorgente e la porta UDP sorgente esterni dei pacchetti ricevuti dall'Anchor Node saranno quelli nattati. Per fare in modo che le corrispondenze del NAT rimangano aperte, sarà usato

Figura 2.3 Schema del pacchetto IP incapsulato per l'invio in un tunnel.



un meccanismo di keep-alive gestito dal client, simile a quello definito nel RFC 3948.

2. L'Anchor Node agisce come punto di ancoraggio per il Mobile Host. Tutti i pacchetti all'interno dei tunnel dal Mobile Host al Correspondent Host sono decapsulati e inoltrati dall'Anchor Node. Le associazioni tra i flussi di dati applicativi e i tunnel dai quali sono ricevuti, sono mantenute dall'Anchor Node in una tabella apposita che è usata da quest'ultimo per instradare i pacchetti provenienti dal Correspondent Host e destinati al Mobile Host. La sua struttura è identica a quella della PAFT già illustrata nel Mobile Host, ovvero mappa i socket nei TID.

3. L'Anchor Node esegue le operazioni standard di NAT su tutti i pacchetti incapsulati dal Mobile Host. In particolare, dopo che un pacchetto è decapsulato, l'indirizzo virtuale del Mobile Host (ovvero l'indirizzo IP sorgente dei pacchetti inviati dal Mobile Host) è sostituito con un indirizzo pubblico dell'Anchor-NAT.

2.1.2 Scenario distribuito

Questo scenario, definito anche *UPMT Distributed AN Mode* o *UPMT-DAM*, è illustrato in figura 2.4.

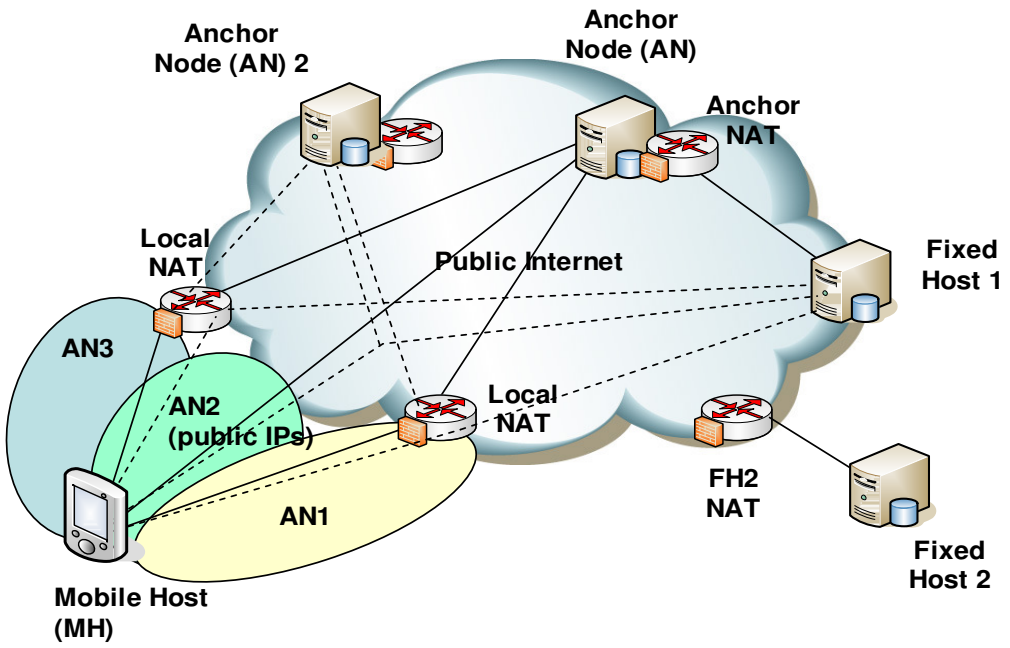
Rispetto allo scenario UPMT-CAM, si introduce un insieme di Anchor Node e si permette la comunicazione diretta tra Mobile Host e Fixed Host UPMT-aware, e tra gli stessi Mobile Host. Si identificano tre diversi scenari tramite cui è possibile rendere distribuita la funzionalità precedentemente centralizzata degli Anchor Node:

1. Anchor Node multipli (multiAN) in cui i Mobile Host possono usare differenti Anchor Node per la comunicazione con Correspondent Host legacy. Gli Anchor Node possono essere scelti per ottimizzare le prestazioni.

2. Scenario MH-to-FH end-to-end (mh2fh), in cui i Mobile Host UPMT-aware e i Fixed Host possono comunicare direttamente tra loro senza affidarsi necessariamente a un Anchor Node per l'inoltro dei pacchetti.

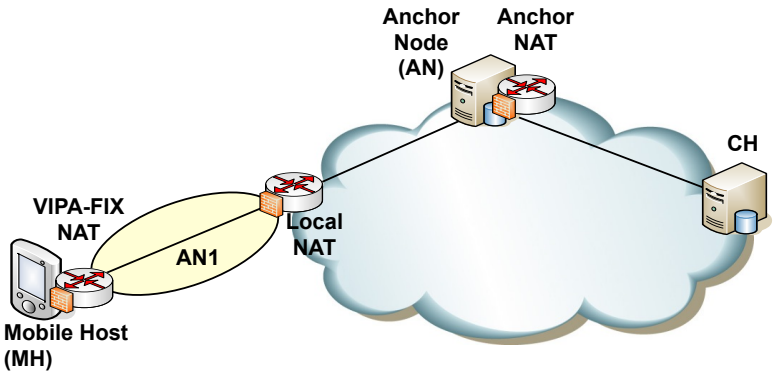
3. Scenario MH-to-MH end-to-end (mh2mh), in cui i Mobile Host UPMT-aware e i Fixed Host possono comunicare direttamente tra loro senza affidarsi necessariamente a un Anchor Node per l'inoltro dei pacchetti e usare differenti Anchor Node.

Figura 2.4 Scenario di riferimento UPMT multi-AN e con Fixed Host.



In questi scenari, il VIPA è unicamente associato ad un MH ma solo dal punto di vista dell'Anchor Node che glielo assegna ("Per-Association Unique VIPA"). In questo modo differenti Anchor Node possono assegnare lo stesso VIPA a differenti Mobile Host e un Mobile Host in generale riceverà differenti VIPA dai differenti Anchor Node a cui si assocerà. Se un Mobile Host è associato a n Anchor Node UPMT, esso riceverà n VIPA, ciascuno dei quali sarà unico solo nell'Anchor Node che gliel'ha fornito. Dato che avere n interfacce virtuali sul Mobile Host renderebbe il sistema non scalabile e difficile da integrare in modo trasparente con le applicazioni legacy, l'interfaccia virtuale sarà configurata con un indirizzo IP fissato e localmente unico, chiamato *Fixed Virtual IP Address* o *VIPAfix*. Si avrà quindi un'operazione di NAT locale, interna al Mobile Host, che sarà effettuata sull'indirizzo IP sorgente per tutti i pacchetti uscenti ed entranti (vedi figura 2.5). Per quelli uscenti quest'operazione di NAT cambierà l'indirizzo IP sorgente con il VIPA relativo all'Anchor Node con cui è stato instaurato il tunnel che si sta usando per quel flusso applicativo. Per realizzare questo NAT interno del VIPAfix, è utilizzata una tabella di tunnel che memorizza tutte le associazioni tra ogni tunnel attivo e il VIPA che dev'essere usato come indirizzo IP sorgente (per i pacchetti in uscita da quel tunnel). Per i pacchetti entranti invece l'indirizzo di destinazione verrà sempre sostituito con il VIPAfix, indipendentemente dal tunnel utilizzato per riceverli.

Figura 2.5 Schema dei NAT logici e reali attraversati.



2.2 Architettura delle entità UPMT

Riportiamo in questa sezione alcuni brevi cenni all'architettura interna dei tre tipi di terminali coinvolti negli scenari esaminati: Mobile Host, Anchor Node e Fixed Host.

2.2.1 Mobile Host

Come mostrato in figura 2.6, un Mobile Host dotato di tecnologia UPMT, oltre alle varie applicazioni legacy ed UPMT-aware, include un Mobility Management Client (MMC) che gestisce l'intero meccanismo di mobilità.

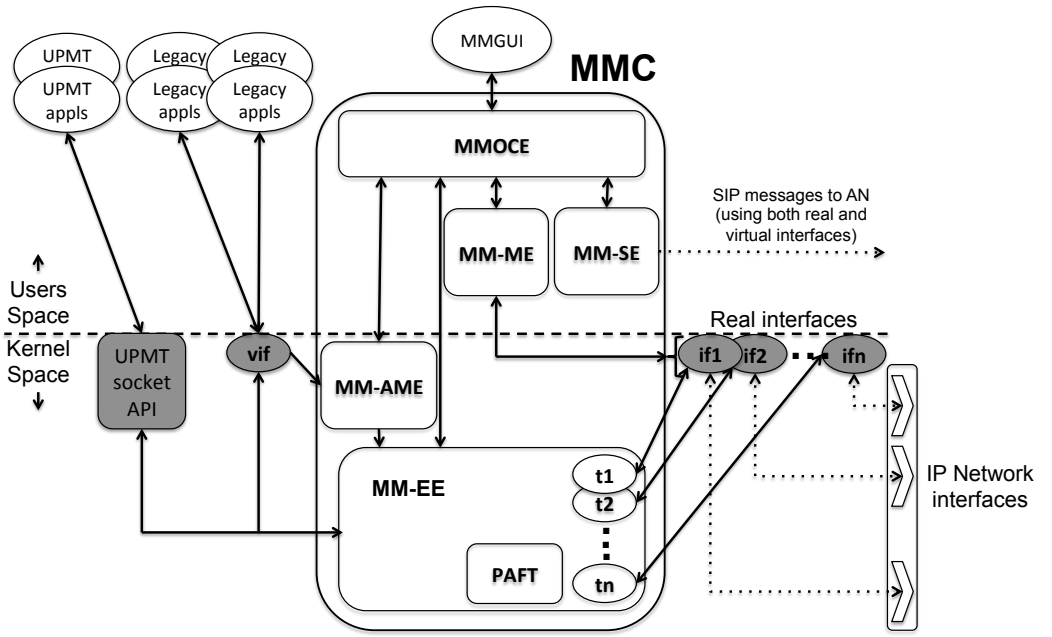
Le applicazioni possono usufruire dei servizi di mobilità offerti dal MMC in due modi: utilizzando i socket UPMT con esplicito supporto alla mobilità, nel caso di applicazioni UPMT-aware, oppure tramite un'interfaccia IP virtuale, nel caso di applicazioni legacy. Per capire più in dettaglio il funzionamento del MMC, lo possiamo ulteriormente decomporre in un insieme di entità:

Mobility Management Overall Control Entity (MMOCE) Si occupa di impostare i criteri di mobilità e di guidarne le decisioni. Il MMOCE offre un'interfaccia grafica all'utente per la configurazione di tali criteri, dopodiché la maggior parte delle decisioni di mobilità saranno prese automaticamente.

Mobility Management Measurement Entity (MM-ME) Ha il compito di effettuare determinate misure a vari livelli, sulle diverse interfacce fisiche e sulle reti di accesso relative. Si occupa, inoltre, di rilevare e segnalare al MMOCE eventi quali l'attivazione di un'interfaccia o la perdita di connettività su una delle interfacce monitorate.

Mobility Management Execution Entity (MM-EE) Fornisce invece il servizio di trasporto alle applicazioni e il servizio di gestione e controllo dei tunnel al MMCOE. Il MM-EE esegue le operazioni di creazione, modifica e chiusura dei tunnel UDP tra ogni interfaccia attiva del Mobile Host e l'Anchor Node. Gestirà inoltre tutti i pacchetti inviati dalle applicazioni in esecuzione sul Mobile Host, inoltrandoli nel tunnel appropriato in base alle regole presenti nella PAFT.

Figura 2.6 Architettura interna del Mobile Host.



Mobility Management Application Monitor Entity (MM-AME) Si occupa di monitorare l'apertura di nuovi socket da parte delle applicazioni legacy. Il suo compito è quello di associare i socket all'applicazione che li ha generati, informare il MMOCE per eventuali decisioni, e popolare la PAFT con le apposite corrispondenze tra ciascun socket e il tunnel da utilizzare in base alla specifica politica di ciascuna applicazione.

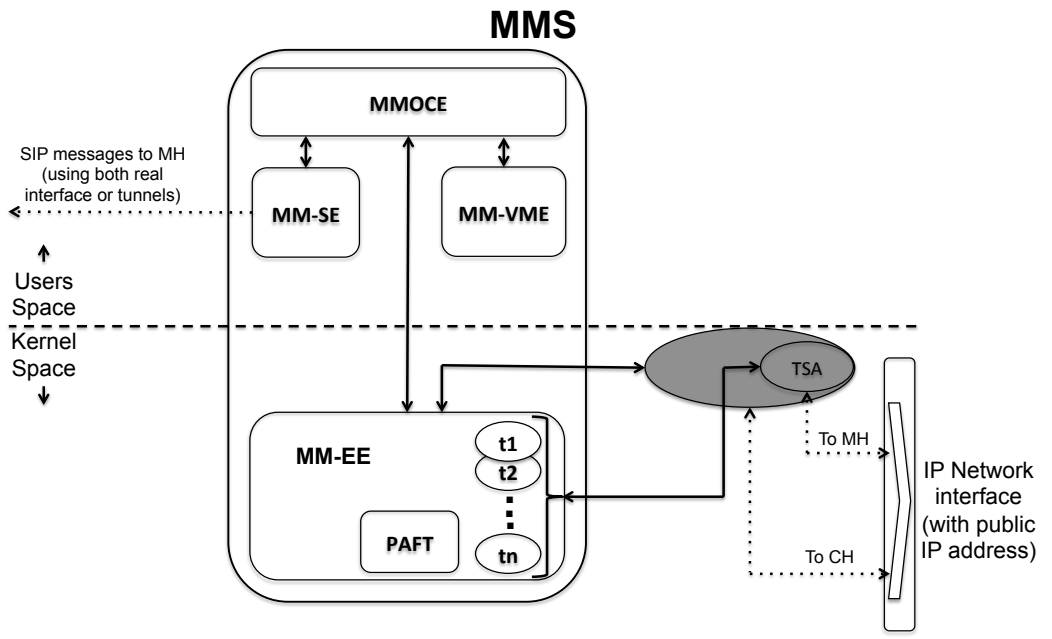
Mobility Management Signaling Entity (MM-SE) Ha il compito di gestire la segnalazione verso le altre entità remote realizzando le procedure UPMT necessarie per ottenere la connettività ed effettuare gli handover. Il MM-SE può inviare i suoi messaggi sia attraverso le interfacce di rete reali, sia attraverso i tunnel instaurati con Anchor Node o Fixed Host.

Il componente "vif" in figura 2.6 è un'interfaccia di rete virtuale, su cui le applicazioni legacy aprono i loro socket, così da poterne gestire i pacchetti attraverso il MMC. Per tali applicazioni, è un compito del MMOCE scegliere quali applicazioni dovrebbero essere gestite dal MMC e quale politica di handover sarà applicata, e ciò è configurabile dall'utente attraverso la GUI del MMOCE.

Per differenziare il comportamento di routing dei pacchetti che devono essere gestiti tramite UPMT da quelli che non devono esserlo, UPMT utilizza il policy routing, che verrà trattato più in dettaglio nella sezione 3.1.

Tutti i pacchetti da incapsulare vengono processati dall'interfaccia virtuale `upmt0`, grazie all'aggiunta di una nuova tabella di routing, chiamata `upmt0_table`, che contiene una rotta con l'indirizzo dell'interfaccia virtuale come default gateway. Per fare in modo che questa tabella sia consultata prima delle altre, si aggiunge una regola che specifica una priorità

Figura 2.7 Architettura interna dell'Anchor Node.



elevata per `upmt0_table`. In questo modo, si isolano le applicazioni da eventuali cambiamenti di indirizzo IP delle interfacce fisiche.

I pacchetti già incapsulati o che non si vuole gestire tramite UPMT devono essere trasmessi direttamente dalle interfacce fisiche. La tabella di routing `main` contiene più default gateway, uno per ciascuna interfaccia fisica, utilizzato per instradare i pacchetti in uscita. L'interfaccia fisica in uscita viene scelta direttamente dal modulo `upmt`. Questi pacchetti sono quindi nattati tramite masquerading così che utilizzino l'indirizzo delle interfacce fisiche come indirizzo sorgente invece dell'indirizzo di `upmt0`.

2.2.2 Anchor Node

L'architettura dell'Anchor Node, illustrata in figura 2.7, è decisamente più semplice di quella del Mobile Host. Anche qui abbiamo un componente di gestione della mobilità chiamato Mobility Management Server (MMS) che possiamo suddividere logicamente in una serie di entità.

Come per il Mobile Host, vi è un MMOCE che si occupa di coordinare l'intero funzionamento del MMS, ed un componente chiamato Mobility Management Signaling Entity (MM-SE) che si occupa della segnalazione. Si noti che mentre il Mobile Host ha la necessità di un'apposita procedura per essere reperibile dal punto di vista dei messaggi di segnalazione, l'Anchor Node dovrebbe poter essere raggiungibile conoscendo l'indirizzo IP o tramite DNS.

Un componente chiamato Mobility Management VIPA Manager Entity (MM-VME) è incaricato invece di gestire l'assegnazione degli indirizzi IP virtuali che l'Anchor Node deve assegnare ad ogni Mobile Host che intende associarsi.

Infine anche nel MMS, così come nel MMC, vi è il componente che si occupa della gestione dei tunnel chiamato Mobility Management Execution Entity (MM-EE).

Come si può vedere dalla figura 2.7, il server è dotato di almeno un'interfaccia pubblica (ovvero connessa ad Internet tramite un indirizzo pubblico e non dietro NAT) che viene usata sia come parte terminale dei tunnel instaurati con il Mobile Host, sia come interfaccia per inoltrare i pacchetti verso il Correspondent Host. In particolare il MM-EE riserva una o più porte UDP dell'interfaccia, per "accettare" l'instaurazione dei tunnel da parte dei Mobile Host. La coppia < indirizzo dell'Anchor Node, porta UDP in ascolto >, viene chiamata Tunnel Server Address (*TSA*) ed è l'indirizzo di destinazione di ogni pacchetto incapsulato generato da qualsiasi Mobile Host. I pacchetti ricevuti da questa porta attraversano il MM-EE, che si occupa di decapsularli, effettuare il NAT visto nella sottosezione 2.1.1, ed inoltrarli attraverso l'interfaccia pubblica dell'Anchor Node. Ogni flusso di pacchetti ricevuto in questo modo (in particolare la quintupla che identifica il socket del pacchetto interno), viene memorizzato nella PAFT, insieme all'identificativo del tunnel da cui è stato ricevuto. In questo modo, le risposte provenienti dal Correspondent Host, possono essere nuovamente incapsulate nel tunnel corretto ed arrivare al Mobile Host che ha originato le richieste.

2.2.3 Fixed Host

L'architettura interna del Fixed Host è molto simile a quella dell'Anchor Node in quanto, come abbiamo visto nella sottosezione 2.1.2, esso può essere considerato un particolare Anchor Node che però viene utilizzato solo per il traffico destinato alle applicazioni in esecuzione su di esso e non verso Correspondent Host esterni.

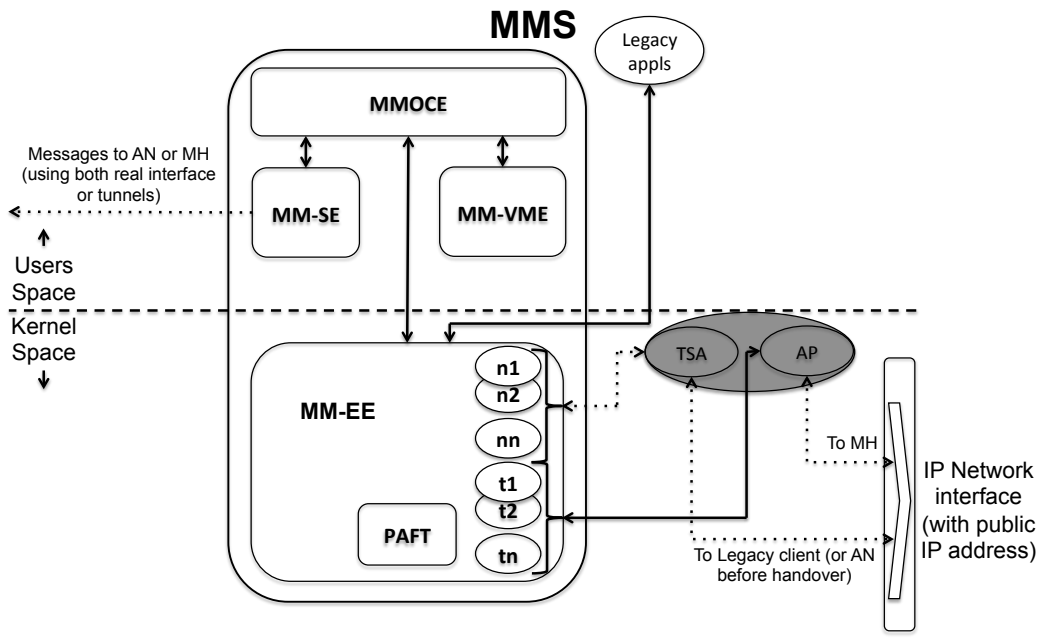
Come si può vedere in figura 2.8 anche qui, quindi, si ha un MMS molto simile a quello dell'Anchor Node e un'interfaccia pubblica con un TSA per gestire i tunnel instaurati con i Mobile Host associati. Una volta che il Mobile Host si è associato e i tunnel sono instaurati, l'unica differenza rispetto al meccanismo illustrato nella sezione precedente è che in questo caso i pacchetti ricevuti attraverso i tunnel (e che hanno come indirizzo sorgente un VIPA) non sono nattati ma sono recapitati direttamente all'applicazione in esecuzione sul Fixed Host.

Il Fixed Host dev'essere però in grado di gestire anche i pacchetti esterni ai tunnel ed inviati sia da altri terminali legacy che da Anchor Node temporaneamente utilizzati dai Mobile Host prima della procedura di associazione e handover. I pacchetti in ingresso vengono quindi gestiti dal MM-EE che distingue i pacchetti provenienti da terminali legacy (lasciati passare senza intervento all'applicazione in ascolto) dai pacchetti provenienti da Anchor Node UPMT (in base ad una lista di indirizzi IP da confrontare con l'indirizzo sorgente). Questi ultimi vengono preventivamente nattati con un indirizzo VIPA che verrà poi assegnato al Mobile Host origine del flusso (vedi sottosezione 2.1.2).

2.3 Modulo tunneling

Finora si è parlato dell'interfaccia virtuale `upmt0` che realizza le funzioni di tunneling. Questa è realizzata tramite un modulo del kernel, che registra un device di rete virtuale. Tutti i pacchetti inviati attraverso questo device vengono gestiti da un'apposita funzione del modulo. Questa funzione compie tre importanti passi nella gestione del pacchetto:

Figura 2.8 Architettura interna del Fixed Host.



- PAFT-lookup
- Incapsulamento
- Inoltro

Il primo consiste nella ricerca, all'interno della struttura dati che realizza la PAFT, del socket relativo al pacchetto considerato. La struttura dati che implementa la PAFT è, per motivi di efficienza e considerando che la dimensione di questa tabella può assumere notevoli dimensioni (soprattutto sull'Anchor Node), una hash table. Dopo aver individuato la tupla corretta, si ottiene il tunnel in cui inoltrare il pacchetto.

Nel secondo passo, tramite l'identificativo del tunnel, si risale ai parametri dello stesso attraverso una tabella chiamata TUNT (Tunnel Table). Anche questa, date le notevoli dimensioni è implementata tramite una hash table. Ottenuti i parametri corretti si passa al vero e proprio incapsulamento. La struttura dati utilizzata nel kernel per rappresentare un pacchetto è la `struct sk_buff`. Questa contiene, tra gli altri, un campo `headroom` che contiene gli header IP e UDP del pacchetto. Dopo diversi di controlli per verificare lo spazio disponibile ed eventualmente clonare l'intero pacchetto, viene aggiunto l'header esterno secondo i parametri del tunnel e viene inoltre calcolato il checksum della sola parte UDP. Questo perché il pacchetto viene incapsulato a metà del suo tragitto all'interno del kernel: è già stata effettuata, infatti, la gestione dello strato di trasporto, ma il pacchetto modificato deve ancora concludere il percorso nello strato IP.

Si occupa di questo infatti la terza fase che, a partire dal pacchetto incapsulato, invoca la funzione del kernel `ip_local_out()`: poiché è stato inserito un nuovo header IP il pacchetto

non viene fisicamente inviato in rete ma reinserito nello strato IP e inviato successivamente attraverso una delle reali interfacce presenti.

Per quanto riguarda la ricezione di pacchetti, una struttura importante è quella del TSA, che rappresenta l'end point per l'instaurazione di tunnel "passiva", ovvero avvenuta per iniziativa dell'altro terminale (che nell'architettura UPMT è il Mobile Host). Il modulo contiene una lista di coppie $<$ interfaccia, porta $>$ da utilizzare come TSA e sulle quali si mette in ascolto.

All'arrivo di un pacchetto si controllano il checksum IP e UDP degli header esterni del pacchetto e si verifica se il pacchetto è incapsulato analizzando anche il checksum degli header IP e transport interni. Se queste condizioni sono soddisfatte allora si utilizzano i parametri del tunnel estrapolati dal pacchetto per effettuare una ricerca all'interno della TUNT. Se questa ha esito negativo allora si è in presenza di una richiesta implicita per la creazione di un nuovo tunnel, che viene soddisfatta inserendo semplicemente il nuovo tunnel nella TUNT, altrimenti si tratta di un pacchetto appartenente ad un tunnel già instaurato. Il pacchetto viene quindi decapsulato e tramite i parametri degli header IP e transport, che prima facevano parte del pacchetto interno, viene effettuata la ricerca nella PAFT. A questo punto si possono verificare tre possibili situazioni:

- La ricerca ha esito positivo: il tunnel associato alla tupla del socket è lo stesso da cui il pacchetto proviene.
- La ricerca ha esito negativo: la nuova associazione tra il socket e il tunnel di provenienza viene registrata nella PAFT.
- La ricerca ha esito positivo ma il tunnel associato è diverso da quello di provenienza: si è in presenza di una richiesta implicita di handover.

Nel terzo caso, si controlla il campo `static` della voce della PAFT che indica il tipo di handover per tale socket. Se il valore è `false` allora l'handover deve essere effettuato e l'identificativo del tunnel di provenienza viene sostituito al precedente nella voce della PAFT. Altrimenti, se il valore è `true`, l'handover deve essere effettuato esplicitamente e, sebbene il pacchetto venga accettato, non viene modificata la voce corrispondente nella PAFT.

Oltre alla principale funzionalità di tunneling (sia in ricezione che in invio) è possibile porre in cascata a questa un NAT interno al terminale su cui il modulo è in esecuzione. Le operazioni di NAT vengono compiute nei due punti di analisi e modifica dei pacchetti, cioè nella ricezione del TSA e nella funzione di invio del device virtuale. Ad ogni entry della tabella in cui sono memorizzati i parametri dei tunnel corrisponde anche un'altra struttura denominata `inat` (Internal Nat), contenente due indirizzi IP che rappresentano il NAT che può essere effettuato in remoto o in locale. Prima dell'eventuale incapsulamento (che avviene, come già detto, se esiste un record della TUNT contenente i parametri del tunnel associato ad un socket nella PAFT) si verifica che i valori della struttura `inat` presente nel record della TUNT siano non nulli. Se questo accade, viene effettuato il NAT sostituendo l'indirizzo sorgente del pacchetto non ancora incapsulato e ricalcolando il checksum degli header network e transport. Viceversa, quando un pacchetto viene decapsulato, se i parametri del tunnel da cui proviene hanno un NAT definito, l'indirizzo destinazione viene sostituito con quello dell'interfaccia virtuale `upmt0` del sistema. Questa funzionalità di NAT (limitato in realtà solo sugli indirizzi IP perché non opera sui numeri di porta) è indispensabile per gestire il

meccanismo già illustrato del VIPAfixe dei VIPA multipli legati a ciascun Anchor Node con cui il Mobile Host è associato. Viene, inoltre, utilizzata anche nel Fixed Host per eseguire il NAT dei pacchetti provenienti dall'Anchor Node prima che il flusso venga spostato sui tunnel instaurati direttamente con il Mobile Host .

Il percorso dei pacchetti in trasmissione e in ricezione in un host dotato di UPMT è riportato, rispettivamente, nelle figure 2.9 e 2.10.

Per controllare l'intero modulo, ovvero per l'inserimento, rimozione e modifica di tunnel, voci della PAFT e TSA, si utilizza un programma user-space utilizzabile da riga di comando chiamato `upmtconf`. Questo comunica con il modulo tramite un socket Netlink (vedi sezione 3.6).

2.4 Obiettivo del lavoro di tesi

L'obiettivo del lavoro di tesi è sviluppare i componenti necessari a rendere *application-aware* l'implementazione di UPMT. Sarà necessario, quindi, identificare le applicazioni e associare ad esse le connessioni di rete, e fornire alla Control Entity gli strumenti per controllare, su base applicazione, l'instradamento delle connessioni sui tunnel in base a politiche definite dall'utente.

Si vuole, inoltre, effettuare il porting dei componenti sviluppati, insieme agli altri componenti di basso livello esistenti, sulla piattaforma mobile Android, in preparazione del futuro porting della Control Entity come applicazione nativa Android.

Figura 2.9 Percorso seguito da un pacchetto in uscita. In giallo sono evidenziati i contributi di questo lavoro di tesi.

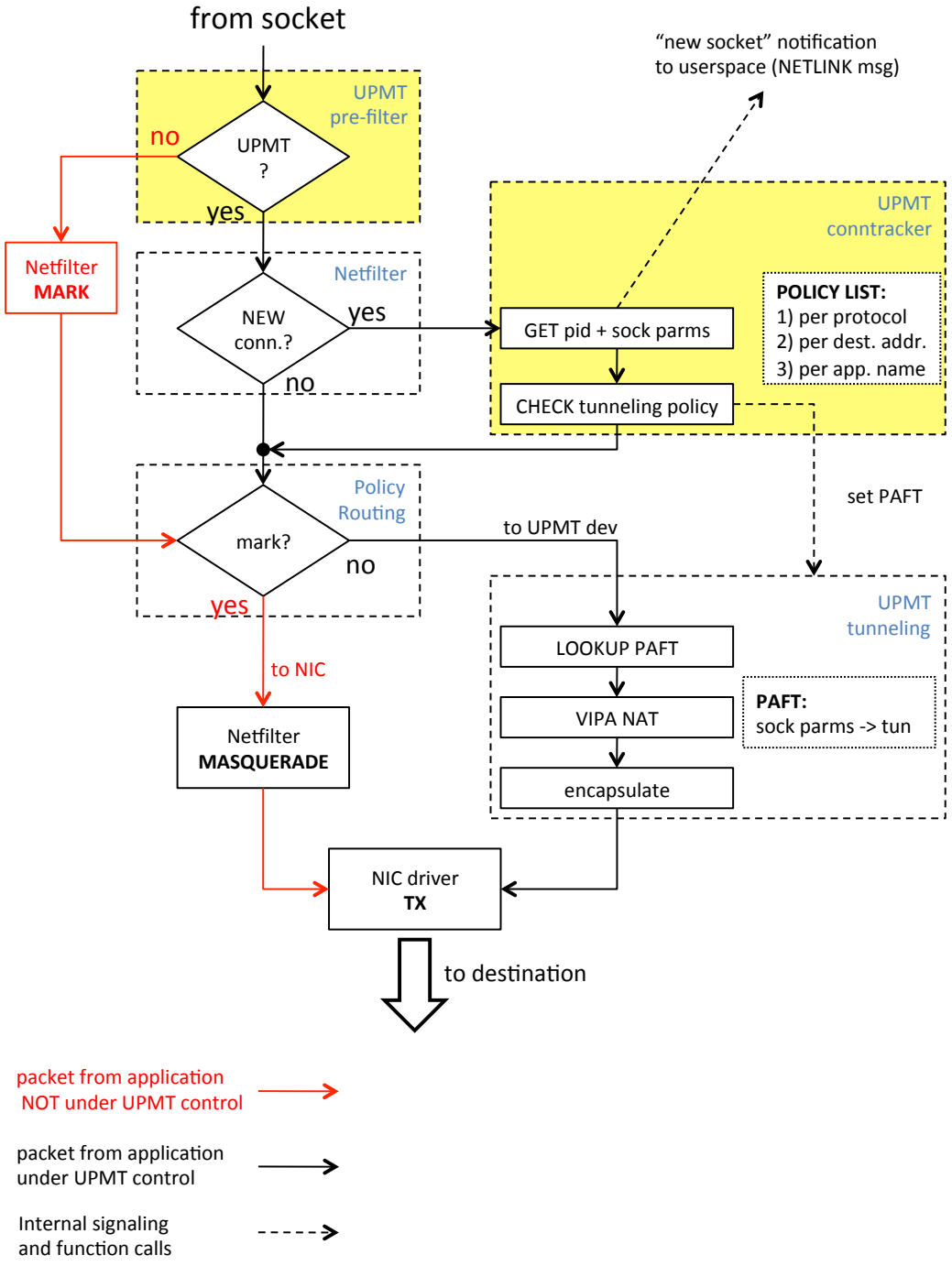
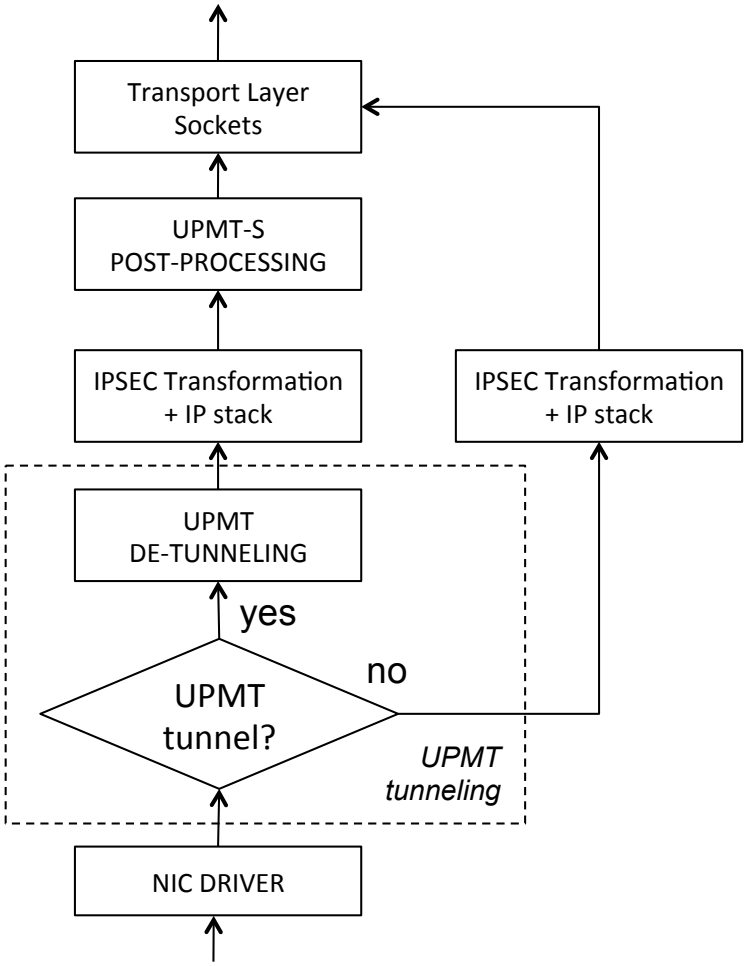


Figura 2.10 Percorso seguito da un pacchetto in ingresso.



3

Strumenti e tecnologie utilizzate

3.1 Tabelle di routing multiple e policy routing

Il kernel Linux supporta tabelle di routing multiple sin dalla versione 2.2. Tale supporto può essere abilitato in fase di configurazione del kernel tramite le opzioni:

```
1   Symbol: IP_ADVANCED_ROUTER [=y]
2   Prompt: IP: advanced router
3     Defined at net/ipv4/Kconfig:17
4     Depends on: NET [=y] && INET [=y]
5     Location:
6       -> Networking support (NET [=y])
7         -> Networking options
8           -> TCP/IP networking (INET [=y])
9
10  Symbol: IP_MULTIPLE_TABLES [=y]
11  Prompt: IP: policy routing
12    Defined at net/ipv4/Kconfig:101
13    Depends on: NET [=y] && INET [=y] && IP_ADVANCED_ROUTER [=y]
14    Location:
15      -> Networking support (NET [=y])
16        -> Networking options
17          -> TCP/IP networking (INET [=y])
18    Selects: FIB_RULES [=y]
```

Oltre alle due usuali tabelle di routing `local` e `main`, il kernel supporta fino a 252 tabelle di routing addizionali. Ogni tabella di routing è contraddistinta da un numero identificativo intero assegnabile dall'utente e compreso nell'intervallo $1 \div 252$ (gli altri valori sono riservati, vedi codice 3.1). Per aggiungere una tabella di routing, è sufficiente aggiungere una riga al file `/etc/iproute2/rt_tables`, scegliendo un valore numerico non utilizzato per identificare la tabella, e assegnandole un nome.

Il comando `route` permette di operare sulla sola tabella `main`: per operare sulle altre tabelle è indispensabile utilizzare il comando `ip` fornito dalla collezione di tool `iproute2`. `iproute2` ha lo scopo di rimpiazzare un intero set di software UNIX (a cui ci si riferisce spesso con il termine `net-tools`), precedentemente utilizzato per la configurazione delle

Codice 3.1 File `/etc/iproute2/rt_tables` contenente le tabelle predefinite.

```
1 #
2 # reserved values
3 #
4 255     local
5 254     main
6 253     default
7 0       unspec
8 #
9 # local
10 #
11 1      inr.ruhep
```

interfacce di rete, tabelle di routing e per la gestione delle tabelle ARP. Tali tool non sono più aggiornati dal 2001. I comandi rimpiazzati da `iproute2` sono:

- Indirizzo e configurazione del link: `ifconfig` → `ip addr`, `ip link`
- Tabelle di routing: `route` → `ip route`
- Vicini: `arp` → `ip neigh`
- Tunnel: `iptunnel` → `ip tunnel`
- Multicast: `ipmaddr` → `ip maddr`
- `netstat` → `ss`

I comandi più utilizzati nel seguito sono i seguenti:

- `ip link set IFACE {up,down}` Attiva (`up`) o disattiva (`down`) l'interfaccia di rete `IFACE`
- `ip addr` Mostra lo stato delle interfacce di rete e gli indirizzi MAC e IP ad esse assegnati
- `ip addr add 10.0.0.1/8 dev IFACE` Aggiunge l'indirizzo `10.0.0.1/8` all'interfaccia `IFACE`
- `ip route` Mostra il contenuto della tabella di routing `main`
- `ip route show table TABLE` Mostra il contenuto della tabella di routing `TABLE`
- `ip route add default via GWIP dev IFACE table TABLE` Aggiunge il default gateway `GWIP` per l'interfaccia `IFACE` nella tabella `TABLE`
- `ip route add 8.8.8.8 via GWIP dev IFACE table TABLE` Forza i pacchetti verso l'indirizzo IP `8.8.8.8` ad essere instradati tramite il gateway `GWIP` e ad uscire dall'interfaccia `IFACE`

L'uso di queste tabelle fornisce un'infrastruttura flessibile per l'implementazione del policy routing. Mentre il routing tradizionale si basa sull'indirizzo di destinazione di un pacchetto, il policy routing permette di tenere conto, nella decisione di routing, anche dell'indirizzo sorgente, della dimensione del pacchetto, del protocollo del payload o di particolari opzioni nell'header del pacchetto.

Il policy routing viene utilizzato in UPMT per differenziare il comportamento di routing per i pacchetti che devono essere gestiti tramite UPMT da quelli che non devono esserlo (vedi sezione 2.3).

3.2 Programmazione kernel e sviluppo di moduli

Il kernel Linux, grazie alla sua natura di software libero, può essere modificato per adattarlo a praticamente ogni contesto applicativo. È possibile sia intervenire direttamente sul codice sorgente, compilarlo e ottenere un kernel binario personalizzato, sia indirettamente, scrivendo uno o più *moduli* che realizzino le funzionalità desiderate, e in seguito *inserirli* in un kernel esistente.

Il primo approccio è sicuramente il più potente, in quanto è possibile modificare ogni aspetto del kernel, tuttavia risulta meno flessibile, in quanto sarà necessario sostituire interamente il kernel eventualmente presente su una macchina con il proprio kernel modificato. Questo metodo risulta conveniente nel caso di dispositivi embedded, il cui kernel è tipicamente configurato con le sole opzioni strettamente necessarie; in questo caso, inoltre, la modularizzazione può introdurre un overhead inaccettabile.

Il secondo approccio è, al contrario, molto flessibile in quanto è possibile caricare (e, se non si è esplicitamente disattivata questa possibilità, scaricare) un modulo da un kernel in esecuzione, senza causare interruzioni di servizio. Tuttavia, i moduli del kernel possono utilizzare solo un sottoinsieme delle funzioni di cui è composto il kernel, in particolare solo le funzioni che siano *esportate*, ossia quelle che appaiano come parametri di apposite macro chiamate `EXPORT_SYMBOL()` o `EXPORT_SYMBOL_GPL()` all'interno del kernel. Questo metodo è più adatto a sistemi general-purpose, in cui è necessario offrire il supporto a una gran varietà di funzioni (ad es. driver) o in casi in cui non sia desiderabile interrompere il funzionamento del sistema per aggiungere una funzionalità al kernel. Tutte le moderne distribuzioni Linux per PC desktop utilizzano questo approccio per supportare la grande varietà di hardware disponibile sul mercato.

Dal punto di vista implementativo, un modulo del kernel Linux è un frammento di codice eseguibile memorizzato su disco sotto forma di file oggetto in formato ELF, similmente a quanto accade con un normale programma user-mode come, ad esempio, `ls`. La directory in cui normalmente sono memorizzati i moduli per un dato kernel è `/lib/modules/VERSION/`, dove `VERSION` è la versione del kernel (ad es. `2.6.37-ARCH`). Un modulo viene inserito nel kernel tramite il comando `insmod`: questa operazione consiste nel caricare in memoria il contenuto del file del modulo leggendolo da disco, nell'allocare le regioni di memoria necessarie, nell'eseguire il linking del modulo con il kernel e nel saltare all'entrypoint predefinito del modulo.

Durante il linking di un modulo, ogni riferimento a simboli globali del kernel (variabili e funzioni) presenti nel codice oggetto del modulo deve essere sostituito con gli indirizzi

appropriati. Questa operazione, molto simile a quella effettuata dal linker durante la compilazione di un programma user-mode, è effettuata dalla system call `init_module()`. Tali indirizzi sono riportati in alcune tabelle di simboli del kernel, contenute in apposite sezioni del segmento codice del kernel, e sono quelli dei simboli esportati tramite le macro `EXPORT_SYMBOL()` o `EXPORT_SYMBOL_GPL()`. Ogni modulo, a sua volta, può esportare i propri simboli, mettendoli a disposizione di altri moduli.

3.2.1 Dipendenze tra moduli

Quando un modulo A usa un simbolo esportato dal modulo B, si crea una relazione di dipendenza tra i due moduli, e si dice che il modulo A "dipende" dal modulo B. Più in dettaglio, il modulo B esporterà un simbolo tramite le macro `EXPORT_SYMBOL()` o `EXPORT_SYMBOL_GPL()`, mentre nel modulo A lo stesso simbolo sarà dichiarato come `extern`. A differenza del caso di programmi user-space, il kernel build system richiede *a tempo di compilazione* "una conoscenza completa di tutti i simboli" di moduli in relazione di dipendenza [4]. Il modo più semplice per soddisfare questo requisito consiste nel compilare congiuntamente tali moduli, tramite un unico makefile oppure, nel caso in cui i sorgenti siano posti in directory separate, tramite un makefile posto alla radice dell'albero delle directory che invochi ricorsivamente i makefile delle singole sottodirectory. In generale, quando un modulo viene compilato, viene generato il file `Module.symvers` che contiene tutti i simboli esportati dal modulo non definiti nel kernel. Questo fatto consente di compilare separatamente moduli in relazione di dipendenza: riprendendo l'esempio del modulo A che dipende dal modulo B, è sufficiente copiare il file `Module.symvers` prodotto dalla compilazione del modulo B nella directory dei sorgenti del modulo A. Alla compilazione del modulo A, il kernel build system unirà i simboli del modulo B riportati nel file con quelli ricavati dal modulo A. Nei casi in cui sia poco pratico copiare i file `Module.symvers`, è possibile utilizzare la variabile di ambiente `KBUILD_EXTRA_SYMBOLS` per passare al kernel build system una lista separata da spazi di percorsi di tali file. Quest'ultimo approccio è utilizzato per il modulo `xt_UPMT` che ha come dipendenza il modulo `compat_xtables`.

Le relazioni di dipendenza devono essere tenute in conto anche in fase di inserimento e di scaricamento di moduli. Se il modulo A dipende dal modulo B, il modulo B deve essere inserito prima del modulo A. Analogamente, perché il modulo B possa essere scaricato, sarà necessario che tutti i moduli che dipendono da esso vengano prima scaricati. Sebbene l'inserimento di questi moduli, nella giusta sequenza, possa essere effettuato tramite il solito comando `insmod`, ciò risulta poco agevole, specialmente quando esistono parecchie dipendenze tra moduli. Per ovviare a questo inconveniente, è possibile utilizzare il comando `modprobe` per caricare un modulo e tutte le sue dipendenze e, analogamente, `modprobe` con l'opzione `-r` per scaricare un modulo e tutte le dipendenze da esso richieste. Per risolvere le relazioni di dipendenza tra moduli, `modprobe` si avvale del file `/lib/modules/VERSION/modules.dep`, le cui righe, della forma `A: B` indicano che il modulo A dipende dal modulo B. Tale file è generato automaticamente dal comando `depmod`, invocato dal kernel build system all'installazione di nuovi moduli[1].

[1] Più specificatamente dal target `install` del makefile.

3.3 Kernel API per liste collegate

Il kernel Linux mette a disposizione una ricca API di macro per la gestione di liste collegate, contenuta nel file header `linux/list.h`. Ciascun elemento è una struct di qualunque tipo, è sufficiente che includa un campo di tipo `struct list_head`, il cui nome e posizione all'interno della struct può essere arbitrario. La `struct list_head` è composta da una coppia di puntatori che collegano un elemento con l'elemento precedente e successivo.

Una lista deve essere dichiarata del tipo della propria `struct` contenente il campo di tipo `struct list_head` e inizializzata come segue:

```
1 struct mystruct {
2   struct list_head list;
3   int foo;
4 };
5
6 struct mystruct mylist;
7 INIT_LIST_HEAD(&(mylist.list));
```

L'aggiunta dell'elemento `tmp` in coda a una lista si effettua:

```
1 list_add_tail(&(tmp->list), &(mylist.list));
```

L'API per le liste mette a disposizione una serie di utili macro per iterare sugli elementi di una lista utilizzando un puntatore `tmp` all'elemento corrente:

```
1 struct mystruct *tmp;
2 list_for_each_entry(tmp, &mylist.list, list) {
3   /* ... */
4 }
```

Se, si desidera si desidera eliminare un elemento durante l'iterazione di una lista, è necessario utilizzare la variante `list_for_each_entry`, che utilizza un ulteriore puntatore. Una volta in corrispondenza dell'elemento da eliminare, è sufficiente utilizzare `list_del`:

```
1 struct mystruct *pos, *q;
2 list_for_each_safe(pos, q, &(mylist.list)) {
3   tmp = list_entry(pos, struct mystruct, list); /* elemento corrente */
4   if (/* ... */)
5     list_del(pos);
6 }
```

Si rimanda a [5] per ulteriori approfondimenti.

3.4 Architettura di Netfilter

Netfilter è un componente del kernel Linux che consente l'intercettazione e la manipolazione dei pacchetti che attraversano un host. Permette di realizzare alcune funzionalità di rete avanzate come la realizzazione di firewall basati sul filtraggio stateful dei pacchetti o configurazioni anche complesse di NAT, o ancora la modifica delle opzioni dei pacchetti in transito.

Il programma user-space utilizzato per la configurare Netfilter è *iptables*, che permette di definire regole per di filtraggio, NAT e manipolazione di pacchetti. Spesso, anche se impropriamente, con il termine iptables ci si riferisce all'intera infrastruttura, incluso Netfilter.

Netfilter è basato su regole raggruppate in catene (*chain*), a loro volta raggruppate in tabelle (*table*). Una tabella definisce un tipo diverso di operazione che è possibile effettuare sui pacchetti; una catena definisce come vengono trattati i pacchetti nelle diverse fasi della loro elaborazione da parte dello stack di rete.

Una regola è costituita da due parti: un *match*, cioè un predicato che un pacchetto deve soddisfare affinché la regola venga applicata, e un *target*, che indica l'azione da eseguire quando il pacchetto soddisfa il match. Ogni catena dispone di una politica di default, che definisce il trattamento dei pacchetti che non soddisfano nessuna regola. Normalmente, non appena un pacchetto soddisfa una regola, l'elaborazione di tale pacchetto da parte di Netfilter termina e le regole successive della catena per tale pacchetto vengono ignorate.

In ogni tabella esistono alcune catene predefinite, ma l'utente può crearne di nuove; uno dei possibili obiettivi è infatti il collegamento a un'altra catena.

Di default, esistono tre tabelle:

`filter` esegue il filtraggio dei pacchetti. Ogni pacchetto attraversa questa tabella. Essa contiene le seguenti catene:

- `INPUT` pacchetti destinati al sistema
- `OUTPUT` pacchetti creati dal sistema
- `FORWARD` pacchetti che hanno come destinazione finale un altro sistema e che non sono stati generati dal sistema stesso

`nat` contiene le regole per la modifica di indirizzi e porte dei pacchetti. Il primo pacchetto di una connessione passa attraverso questa tabella, e il risultato del passaggio del primo pacchetto determina come tutti gli altri pacchetti della stessa connessione verranno modificati. Contiene le seguenti catene:

- `PREROUTING` pacchetti in entrata, prima che la tabella di routing locale venga consultata. Usata per il NAT sulla destinazione o DNAT.
- `POSTROUTING` pacchetti in uscita, dopo che la tabella di routing locale venga consultata. Usata per il NAT sulla sorgente o SNAT.
- `OUTPUT` permette un DNAT limitato sui pacchetti generati localmente.

`mangle` contiene le regole per la manipolazione delle opzioni dei pacchetti, ad esempio l'impostazione di *mark* o di opzioni relative alla QoS. Tutti i pacchetti passano attraverso questa tabella. Essa contiene tutte le catene predefinite:

- `PREROUTING` esamina tutti i pacchetti che in qualche modo entrano nel sistema. Questo processo avviene prima che il routing decida se il pacchetto debba essere inoltrato (catena `FORWARD`) o se sia destinato al sistema. Viene utilizzata per manipolare l'header del pacchetto (catena `INPUT`).
- `INPUT` tutti i pacchetti destinati al sistema passano per questa catena.
- `FORWARD` tutti i pacchetti che vengono instradati dal sistema ma di cui il sistema non è né sorgente iniziale né destinazione finale passano per questa catena.
- `OUTPUT` tutti i pacchetti generati dal sistema passano per questa catena.

`POSTROUTING` tutti i pacchetti che lasciano il sistema, sia quelli in `OUTPUT` sia quelli in `FORWARD`, passano poi per questa catena.

`raw` introdotta per evitare il connection tracking per quei pacchetti che, per una qualche ragione, non si vogliono filtrare in maniera stateful. Le catene previste sono solo due:

`OUTPUT` pacchetti generati da processi locali

`PREROUTING` pacchetti provenienti da qualsiasi interfaccia di rete

Le catene predefinite `PREROUTING`, `INPUT`, `FORWARD`, `OUTPUT` e `POSTROUTING` sono implementate, a livello kernel, tramite altrettanti *hook*, definiti (per il protocollo IPv4) nel file header `linux/netfilter_ipv4.h`. Questi sono:

`NF_IP_PRE_ROUTING` Invocato dopo controlli di validità del socket buffer, prima della decisione di routing

`NF_IP_LOCAL_IN` Invocato dopo la decisione di routing se il pacchetto è destinato all'host

`NF_IP_FORWARD` Invocato se il pacchetto è destinato a un'altra interfaccia di rete

`NF_IP_LOCAL_OUT` Invocato per i pacchetti in uscita generati da processi locali

`NF_IP_POST_ROUTING` Invocato dai pacchetti in uscita dopo la decisione di routing

Un hook, dunque, è un'interfaccia per registrare, in un punto preciso dello stack IP, una funzione di callback per intercettare o manipolare pacchetti.

In figura 3.1 è riportato il percorso di un generico pacchetto all'interno di Netfilter.

Il match di una regola può essere effettuato su diverse caratteristiche dei pacchetti, le più comuni sono il protocollo, l'indirizzo sorgente o destinazione e le porte sorgente o destinazione. Tuttavia, esistono diverse estensioni di Netfilter che mettono a disposizione costrutti più avanzati per la costruzione di match.

I target di una regola possono essere una catena definita dall'utente, uno dei target predefiniti oppure un target aggiuntivo definito da un'estensione. I target predefiniti sono:

`ACCEPT` A seconda della catena che sta processando il pacchetto, può assumere diversi significati: per la catena `INPUT`, il pacchetto può essere ricevuto dal sistema, per la catena `FORWARD` il pacchetto può essere inoltrato, ecc.

`DROP` Il pacchetto viene scartato senza fornire alcuna indicazione all'applicazione o al sistema che lo ha inviato.

`QUEUE` Il pacchetto viene inserito in una coda, in modo che possa essere processato da una applicazione.

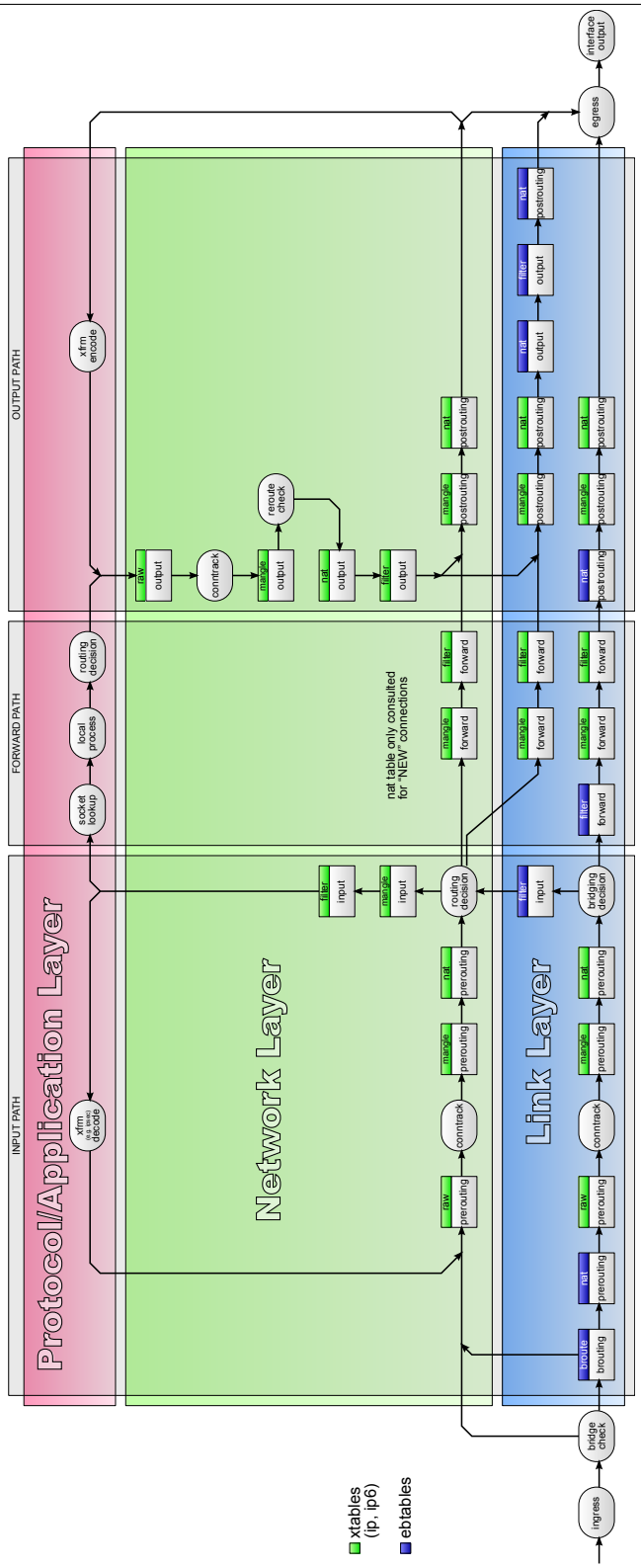
`RETURN` Ha lo stesso effetto di raggiungere la fine della catena.

Alcuni dei più noti e usati target disponibili sono:

`REJECT` Simile a `DROP` ma viene inviato un pacchetto di errore ICMP al mittente del pacchetto.

`LOG` La ricezione del pacchetto viene loggata sul log del kernel, accessibile tramite il comando `dmesg`.

Figura 3.1 Percorso di un pacchetto all'interno di Netfilter [27].



`DNAT` Comporta la riscrittura dell'indirizzo di destinazione del pacchetto (Destination NAT).

`SNAT` Comporta la riscrittura dell'indirizzo sorgente del pacchetto (Source NAT).

`MASQUERADE` Tipologia di SNAT per indirizzi IP dinamici, come quelli forniti da molti ISP ai loro utenti.

Netfilter è progettato per poter essere facilmente esteso attraverso moduli che aggiungono funzionalità:

- predicati per identificare i pacchetti (match)
- operazioni da applicare ai pacchetti (target)
- tracciare le connessioni di un protocollo, necessario per gestire un protocollo che utilizza più connessioni e in cui una di esse viene usata per controllarne altre (ad esempio, FTP)
- supportare il NAT di un protocollo
- analizzare il traffico alla ricerca di particolari protocolli di livello applicazione (`l7_filters`), per identificare protocolli che non è possibile riconoscere solo sulla base delle porte utilizzate

Ciascuna estensione è composta da un modulo del kernel Linux[2] e da un plugin user-space per iptables che fornisce all'utente un modo per controllare il modulo[3].

3.4.1 Connection tracking

Una delle funzioni più importanti fornite da Netfilter è il *connection tracking*. Questo meccanismo consente al kernel di tenere traccia di tutte le connessioni di rete logiche o *sessioni*, e quindi di mettere in relazione i pacchetti che appartengono a una stessa connessione. Questa possibilità è fondamentale per il NAT, che usa queste informazioni per tradurre allo stesso modo gli indirizzi dei pacchetti di una stessa connessione. Si noti che lo stato della connessione è indipendente dallo stato dei protocolli di rete che gestiscono tale connessione a livello di trasporto, come TCP. La motivazione dietro questa scelta è sia legata a motivi di efficienza, in quanto durante l'inoltro di un pacchetto non è necessario processarlo a livello trasporto, sia per assegnare uno stato, seppur solo logico, a protocolli stateless come UDP. In questo caso, gli stati della connessione vengono assegnati in modo euristico basandosi su dei valori di timeout per rilevare periodi di inattività e, in tal caso, "chiudere" la connessione.

Gli stati possibili per una connessione sono:

`NEW` il pacchetto inizia una nuova connessione

`ESTABLISHED` il pacchetto fa parte di una connessione già stabilita

`RELATED` il pacchetto ha qualche relazione con un'altra connessione già stabilita

[2] È anche possibile compilare tali funzionalità aggiuntive staticamente nel kernel.
[3] Anche in questo caso è possibile linkare staticamente tale plugin in iptables.

`INVALID` il pacchetto non fa parte di alcuna connessione

Tipicamente, il primo pacchetto visto dal firewall viene classificato come `NEW`, la risposta viene classificata come `ESTABLISHED` e un messaggio di errore, ad esempio un errore ICMP, come `RELATED`. Un errore ICMP che non appartiene a nessuna connessione può essere classificato come `INVALID`.

Una connessione Netfilter è identificata univocamente dalla tupla:

$$< \text{protocollo L3},\ \text{indirizzo sorgente},\ \text{indirizzo destinazione},\ \text{protocollo L4},\ \text{key L4} >$$

La key L4 dipende dal protocollo di trasporto: per TCP e UDP è composta da numeri di porta, in altri casi è spesso nulla.

3.5 Sviluppo di target con Xtables

In figura 3.2 è riportato un diagramma a blocchi dell'architettura di Netfilter. Come si può vedere, il layer immediatamente sopra a Netfilter è Xtables, il quale fornisce un'interfaccia comune a diversi programmi user-space quali iptables, ip6tables, arptables, ecc.

Per estendere le funzionalità di Netfilter, è possibile avvalersi di Xtables-addons, una ricca collezione di match e target di tipico utilizzo. Questo pacchetto fornisce inoltre uno strato di API, simile a quello nativo del kernel, per l'implementazione di match e target: il vantaggio di questa soluzione è che le estensioni possono essere eseguite su un kernel più vecchio. Allo stato attuale, Xtables-addons fornisce retrocompatibilità fino al kernel 2.6.17, il che corrisponde a una copertura temporale di circa 4 anni.

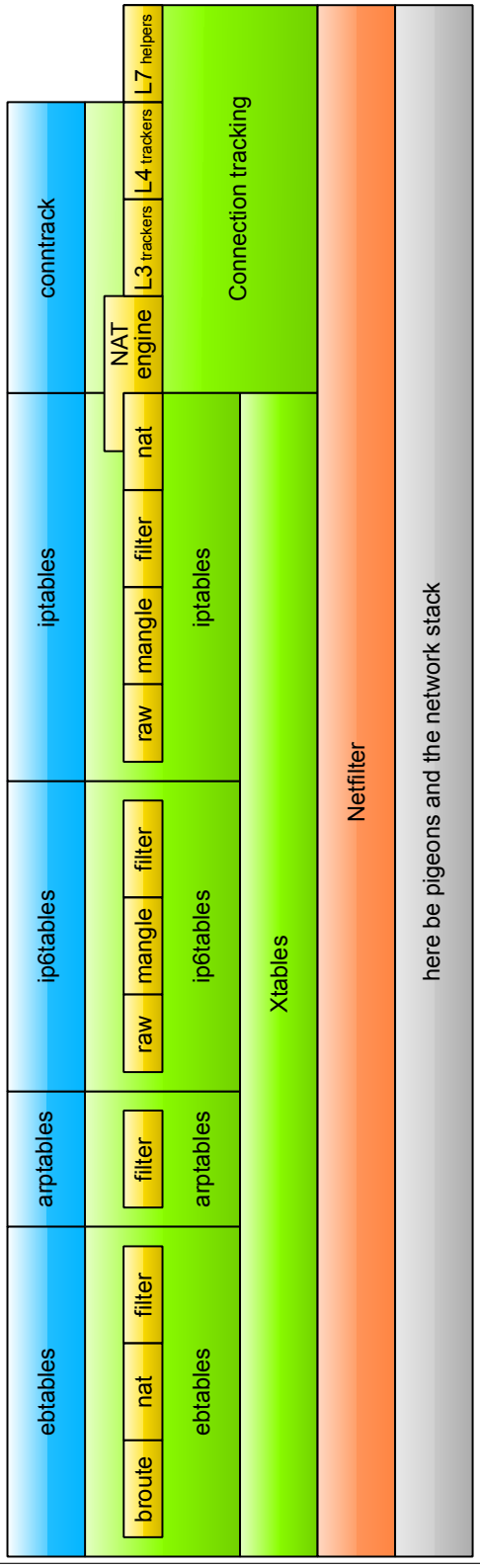
Estendere Netfilter è un compito abbastanza agevole, anche grazie alla buona documentazione disponibile. In particolare, l'articolo [6] fornisce molti chiarimenti utili sull'uso dell'API Xtables. Illustriamo di seguito i passi necessari a scrivere un nuovo target, riportando l'esempio del target `xt_ECHO` presente in Xtables-addons.

Analizziamo innanzitutto il modulo del kernel. Le funzioni di inizializzazione e di terminazione sono definite e registrate come di consueto per un modulo:

```
1  static int __init echo_tg_init(void)
2  {
3    return xt_register_target(&echo_tg_reg);
4  }
5
6  static void __exit echo_tg_exit(void)
7  {
8    return xt_unregister_target(&echo_tg_reg);
9  }
10
11 module_init(echo_tg_init);
12 module_exit(echo_tg_exit);
13 MODULE_AUTHOR("Jan Engelhardt <jengelh@medozas.de>");
14 MODULE_DESCRIPTION("Xtables: ECHO diagnosis target");
15 MODULE_LICENSE("GPL");
16 MODULE_ALIAS("ipt_ECHO");
```

La variabile `echo_tg_reg` è il descrittore del target, ed è di seguito inizializzato:

Figura 3.2 Componenti di Netfilter [28].



```
1 static struct xt_target echo_tg_reg __read_mostly = {
2   .name       = "ECHO",
3   .revision   = 0,
4   .family     = NFPROTO_IPV4,
5   .proto      = IPPROTO_UDP,
6   .table      = "filter",
7   .target     = echo_tg4,
8   .me         = THIS_MODULE,
9 };
```

Il campo `name` è il nome del target, ossia la stringa da utilizzare in iptables come argomento dell'opzione `-j`, in questo caso `-j ECHO`. Il campo `proto` restringe l'utilizzo del target al solo protocollo UDP. Il campo `table` restringe l'uso del target alla sola tabella `filter`. Il campo `target` è un puntatore alla funzione invocata quando un pacchetto soddisfa il match e che viene passato al target. La funzione `echo_tg4` ha il seguente prototipo:

```
1 static unsigned int echo_tg4(struct sk_buff **pskb, const struct xt_action_param *par);
```

Dereferenziando il puntatore `pskb` si ottiene il socket buffer contenente il pacchetto. La funzione deve ritornare uno dei seguenti valori:

`XT_CONTINUE` Continua con la prossima regola, usato tipicamente da target che loggano o modificano pacchetti tramite mangling

`NF_DROP` Interrompi l'attraversamento della catena corrente e scarta il pacchetto

`NF_ACCEPT` Interrompi l'attraversamento della catena corrente e accetta il pacchetto

`XT_RETURN` Ritorna alla catena precedente o alla policy di default della catena

La compilazione di questo modulo è analoga a quella di un qualsiasi altro modulo, tranne per il fatto che esso richiede da alcuni simboli esportati dal modulo di Xtables-addons `compat_xtables` (ad es. le funzioni `xt_register_target` e `xt_unregister_target`). Pertanto, sarà necessario compilare e installare tale modulo come illustrato nella sottosezione 3.2.1.

Il plugin user-space è composto dal seguente codice:

```
1  static void echo_tg_help(void)
2  {
3    printf("ECHO takes no options\n\n");
4  }
5
6  static int echo_tg_parse(int c, char **argv, int invert, unsigned int *flags, const void *
       entry, struct xt_entry_target **target)
7  {
8    return 0;
9  }
10
11 static void echo_tg_check(unsigned int flags)
12 {
13 }
14
15 static struct xtables_target echo_tg_reg = {
```

```
16   .version       = XTABLES_VERSION,
17   .name          = "ECHO",
18   .family        = NFPROTO_UNSPEC,
19   .help          = echo_tg_help,
20   .parse         = echo_tg_parse,
21   .final_check   = echo_tg_check,
22 };
23
24 static __attribute__((constructor)) void echo_tg_ldr(void)
25 {
26   xtables_register_target(&echo_tg_reg);
27 }
```

Tale plugin deve essere compilato producendo una libreria dinamica:

```
1 gcc -Wall -I${XTABLES_SRC}/include -I${XTABLES_SRC}/extensions -fPIC -o libxt_ECHO.o -c
      libxt_ECHO.c
2 gcc -Wall -shared -o libxt_ECHO.so libxt_ECHO.o
```

dove `${XTABLES_SRC}` è la directory dove è stato estratto il tarball di Xtables-addons. Il file `libxt_ECHO.so` ottenuto deve essere posto nella directory dove iptables cerca i plugin. Tale percorso dipende da come è stato compilato iptables, ma in generale può essere ottenuto tramite il comando:

```
1 pkg-config --variable=xtlibdir xtables
```

e su Ubuntu 10.10 è `/lib/xtables/`.

A questo punto, il target può essere utilizzato tramite iptables, ad esempio:

```
1 # iptables -A OUTPUT -p udp -j ECHO
```

3.6 Socket Netlink e Generic Netlink

Uno dei meccanismi più diffusi per la comunicazione tra kernel e processi user-mode è costituito dai *socket Netlink* [7, 9]. Questa primitiva di comunicazione, utilizzata da molti programmi, tra cui spiccano `iptables`, `iproute2`, `wpa_supplicant` (gestione autenticazione WPA in reti 802.11) ecc., sostituisce altri protocolli per la comunicazione con il kernel, quali i socket raw e le chiamate `ioctl`. Netlink fornisce un link di comunicazione full-duplex tramite l'uso dell'API standard dei socket per i processi user-space e una API speciale per il kernel. I socket Netlink utilizzano la famiglia di indirizzi `AF_NETLINK`, a differenza della famiglia `AF_INET` usata dai socket TCP/IP.

Netlink è un protocollo asincrono: la system call per l'invio di un messaggio si limita ad inserire tale messaggio nella coda di ricezione del destinatario e a invocare l'handler di ricezione del destinatario. Quest'ultimo potrà decidere se elaborare il messaggio immediatamente o lasciarlo in coda per elaborarlo in un secondo momento.

Netlink supporta la comunicazione multicast: un processo può inviare un messaggio a un gruppo di indirizzi netlink, e un qualsiasi numero di processi può mettersi in attesa del messaggio in tale gruppo di indirizzi. Inoltre, la comunicazione è bidirezionale: può essere iniziata, cioè, sia dal kernel che da un processo user-space. Ciò consente ai processi user-space di evitare il polling per ricevere messaggi originati dal kernel.

Tramite socket Netlink possono essere veicolati protocolli di diverso tipo, per ognuno dei quali esiste una definizione nel file header `include/linux/netlink.h`. Alcuni protocolli di questo tipo sono:

`NETLINK_ROUTE` Utilizzato da demoni di routing quali BGP, OSPF, RIP, nonché dal modulo di inoltro dei pacchetti del kernel

`NETLINK_FIREWALL` Utilizzato da Netfilter per permettere la gestione dei pacchetti da parte di tool user-space

`NETLINK_NFLOG` Utilizzato da iptables

Per creare un nuovo protocollo da utilizzare tramite socket Netlink, è sufficiente definire un protocol type in `include/linux/netlink.h`. Inoltre, il massimo numero possibile di protocolli è 32. Questo approccio, pertanto, benché semplice, risulta poco flessibile.

Per ovviare a questo inconveniente, è stato introdotto *Generic Netlink*. Questo definisce il protocol type `NETLINK_GENERIC` e offre una serie di canali di comunicazione di cui viene eseguito il multiplexing su una singola famiglia netlink. Ciascun canale è identificato univocamente da un valore numerico, assegnato dimanicamente dal controller generic netlink. I produttori di servizi (programmi user-space o il kernel), dunque, registrano i servizi offerti tramite questo controller. I consumatori di servizi effettuano una query al controller per verificare la disponibilità di tali servizi e ottenere il numero di canale corrispondente. Poiché generic netlink è solo un modo per definire canali di comunicazione e non è usato direttamente dalle entità in comunicazione, ciascun canale viene chiamato, un po' impropriamente, famiglia generic netlink.

Ogni famiglia generic netlink prevede un insieme di tipi di messaggio o `comandi`, che possono includere, opzionalmente, degli `attributi`. Ad ogni comando è associata una funzione callback nel kernel, invocata alla ricezione del comando e a cui vengono passati, tramite un descrittore, i valori degli attributi previsti per il comando.

4 Progettazione della soluzione

Nella sezione 2.4 si è descritto l'obiettivo di questo lavoro di tesi. In questo capitolo si speficano le scelte progettuali effettuate durante lo sviluppo della soluzione.

4.1 Requisiti della soluzione

I requisiti che la soluzione deve soddisfare sono i seguenti:

1. La Control Entity deve essere notificata, in modo asincrono (senza effettuare polling), dell'instaurazione di nuove connessioni, in modo tale da consentire all'utente di effettuare l'handover a livello di singola connessione o di applicazione
2. Ognuna di queste connessioni deve essere associata all'applicazione che l'ha instaurata, e il nome di tale applicazione deve essere presentato all'utente
3. L'utente deve poter definire, tramite file di configurazione (e in futuro tramite GUI), una lista di policy su base applicazione, riguardanti restrizioni o forzature relative alle interfacce di rete da usare
4. Al rilevamento di una nuova connessione, la PAFT deve essere modificata in modo da garantire il rispetto delle policy delle applicazioni
5. Il traffico da/per le reti locali a cui il Mobile Host è collegato, compreso il traffico broadcast e multicast locale, non deve essere gestito da UPMT.

4.2 Rilevazione delle connessioni

Per soddisfare il requisito 1, è necessario un meccanismo che consenta di intercettare una nuova connessione da parte di un'applicazione. Un naturale candidato a questo ruolo è il modulo che realizza il connection tracking di Netfilter, utilizzabile tramite il target di iptables denominato `conntrack`. Questo modulo, tramite il match `--ctstate`, permette di valutare lo stato di una connessione. Il pacchetto che inizia una connessione pone questa nello stato

`NEW`: nel caso che il protocollo L4 sia TCP, questo è tipicamente un SYN. Ad esempio, tramite il seguente comando:

```
1 # iptables -A OUTPUT -m conntrack --ctstate NEW -j LOG
```

è possibile registrare sul kernel log i pacchetti che iniziano nuove connessioni verso l'esterno, invocando il target `LOG` sui pacchetti che soddisfano il match.

La strategia da adottare consiste nella scrittura di un nuovo target da utilizzare congiuntamente al match offerto dal conntrack. In tal modo, la funzione target di questo modulo viene registrata come funzione di callback per Netfilter, che la invoca a ogni nuova connessione, passandole come argomento il socket buffer contenente il primo pacchetto.

4.3 Associazione tra connessione e applicazione

All'interno della funzione target precedentemente definita si ha a disposizione il socket buffer contenente il primo pacchetto. Per soddisfare il requisito 2, occorre un meccanismo per risalire all'applicazione che ha generato il pacchetto e, quindi, che detiene la connessione corrispondente. In passato, Netfilter forniva, tramite il modulo `owner`, il match `--pid-owner`, che permetteva di associare a un pacchetto il PID del processo che lo aveva generato. Tale match, però, è stato rimosso nella release 2.6.14 del kernel [10]:

```
1 [NETFILTER]: Remove tasklist_lock abuse in ipt{,6}owner
2 Rip out cmd/sid/pid matching since its unfixable broken and stands in the way of locking
      changes to tasklist_lock.
```

L'approccio proposto consiste nell'aggiungere un nuovo campo alla struttura dati `struct sk_buff` contenente il PID del processo che ha generato il pacchetto.

Tale valore è contenuto nel descrittore di processo, implementato dalla struttura `struct tast_struct`, nel campo `tgid`. Si noti che non esiste, come ci si aspetterebbe, un campo `pid`. Lo standard POSIX richiede che i thread di uno stesso gruppo condividano lo stesso identificatore, in modo che i segnali inviati a un processo possano essere ricevuti anche da tutti i suoi thread. Tale identificatore è il PID del thread group leader, ed è memorizzato nel campo `tgid` del descrittore processo. La funzione `getpid()` restituisce sempre il TGID: per il processo padre corrisponde al suo PID, per i thread figli corrisponde al PID del padre. Il descrittore di processo del processo attualmente in esecuzione è accessibile tramite la variabile per-CPU `current`.

Il codice 4.1 riporta la patch applicata al kernel con queste modifiche. Il primo segmento del codice (righe 1 ÷ 12) contiene la dichiarazione del campo `tgid` nel corpo della struttura `struct sk_buff`. Il secondo segmento (righe 14 ÷ 29) contiene l'assegnazione del TGID del processo attualmente in esecuzione al nuovo campo in `struct sk_buff`; tale assegnazione è effettuata nella funzione `__alloc_skb`, responsabile dell'allocazione di un socket buffer. L'`if` alla riga 21 è necessario per assicurarsi che il socket buffer corrisponda a un pacchetto generato da una system call (e, dunque, da un processo user-space), e non dal kernel. In quest'ultimo caso, infatti, la nozione di PID non avrebbe senso. Questa funzione può, su esplicita richiesta tramite il parametro `fclone`, allocare il socket buffer richiesto da una cache. In tal caso, è necessario copiare il TGID nel socket buffer da restituire dalla cache.

Codice 4.1 Patch applicata al kernel Linux per includere il campo `tgid` in `struct sk_buff`.

```
1  diff -Naur linux-source-2.6.34/include/linux/skbuff.h linux-source-2.6.34-test/include/linux/
       skbuff.h
2  --- linux-source-2.6.34/include/linux/skbuff.h  2010-10-16 21:16:01.131994668 +0200
3  +++ linux-source-2.6.34-test/include/linux/skbuff.h 2010-10-16 21:17:48.723999092 +0200
4  @@ -329,6 +329,8 @@
5     * first. This is owned by whoever has the skb queued ATM.
6     */
7    char      cb[48] __aligned(8);
8  +
9  + __u32     tgid; /* UPMT */
10
11   unsigned long   _skb_refdst;
12  #ifdef CONFIG_XFRM
13
14 diff -Naur linux-source-2.6.34/net/core/skbuff.c linux-source-2.6.34-test/net/core/skbuff.c
15 --- linux-source-2.6.34/net/core/skbuff.c 2010-10-16 21:15:52.732982098 +0200
16 +++ linux-source-2.6.34-test/net/core/skbuff.c  2010-10-16 21:16:42.495226510 +0200
17 @@ -213,8 +213,12 @@
18   memset(shinfo, 0, offsetof(struct skb_shared_info, dataref));
19   atomic_set(&shinfo->dataref, 1);
20
21 + if(in_interrupt() == 0) /* UPMT */
22 +   skb->tgid=current->tgid; /* UPMT */
23 +
24   if (fclone) {
25     struct sk_buff *child = skb + 1;
26 +   child->tgid=skb->tgid; /* UPMT */
27     atomic_t *fclone_ref = (atomic_t *) (child + 1);
28
29     kmemcheck_annotate_bitfield(child, flags1);
```

Una volta che la funzione target può estrarre il PID dal socket buffer, questo deve essere mappato in un nome di applicazione da presentare all'utente. In ambiente Linux su un tradizionale PC, le applicazioni sono per la maggior parte scritte in codice nativo, ossia sono costituite da file eseguibili in formato ELF. È quindi lecito utilizzare il nome del file eseguibile come nome dell'applicazione. Durante il caricamento di un eseguibile, il kernel crea un processo e assegna ad esso alcune regioni di memoria, consultabili tramite `/proc/PID/maps`:

```
1  00400000-0040b000 r-xp 00000000 08:01 1044558                    /bin/cat
2  0060b000-0060c000 rw-p 0000b000 08:01 1044558                    /bin/cat
3  00ee6000-00f07000 rw-p 00000000 00:00 0                          [heap]
4  7f26f151d000-7f26f1675000 r-xp 00000000 08:01 788720             /lib/libc-2.13.so
5  7f26f1675000-7f26f1874000 ---p 00158000 08:01 788720             /lib/libc-2.13.so
6  7f26f1874000-7f26f1878000 r--p 00157000 08:01 788720             /lib/libc-2.13.so
7  7f26f1878000-7f26f1879000 rw-p 0015b000 08:01 788720             /lib/libc-2.13.so
8  7f26f1879000-7f26f187e000 rw-p 00000000 00:00 0
9  7f26f187e000-7f26f189c000 r-xp 00000000 08:01 791974             /lib/ld-2.13.so
10 7f26f18c4000-7f26f1a77000 r--p 00000000 08:01 548623             /usr/lib/locale/locale-archive
11 7f26f1a77000-7f26f1a7a000 rw-p 00000000 00:00 0
12 7f26f1a9a000-7f26f1a9b000 rw-p 00000000 00:00 0
13 7f26f1a9b000-7f26f1a9c000 r--p 0001d000 08:01 791974             /lib/ld-2.13.so
14 7f26f1a9c000-7f26f1a9d000 rw-p 0001e000 08:01 791974             /lib/ld-2.13.so
15 7f26f1a9d000-7f26f1a9e000 rw-p 00000000 00:00 0
16 7ffff1979000-7ffff199a000 rw-p 00000000 00:00 0                  [stack]
17 7ffff19ff000-7ffff1a00000 r-xp 00000000 00:00 0                  [vdso]
18 ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

Per ogni processo, esiste sempre una regione di memoria di sola lettura e con permessi di esecuzione che costituisce l'immagine del file eseguibile del processo (righe 1 e 2). Nell'esempio sopra, il valore 1044558 in tali righe corrisponde al numero dell'inode del file eseguibile.

Si potrebbe quindi iterare sulla lista delle regioni di memoria di un processo, accessibile tramite il campo `mmap` di tipo `struct vm_area_struct*` del descrittore di memoria `struct mm_struct`, alla ricerca di una regione di memoria di sola lettura e con permessi di esecuzione e ottenere così il corrispondente oggetto `vm_file` di tipo `struct file*`, ossia il descrittore del file eseguibile su disco. Questa operazione è però già svolta dal proc file system, la cui voce `/proc/PID/exe` è un link simbolico al file eseguibile del processo con PID dato. L'oggetto `struct file*` corrispondente al file puntato da questo link simbolico è memorizzato nel campo `exe_file` del descrittore di memoria.

Il codice 4.2 riporta l'implementazione della funzione che restituisce il nome del file eseguibile di un processo in esecuzione dato il suo PID, implementata leggendo il campo `exe_file` del descrittore di memoria.

Alla riga 9, la funzione `find_get_pid()` restituisce il descrittore di tipo `struct pid` dato il PID come numero intero. Alla riga 10, la funzione `pid_task()` restituisce il descrittore del processo di tipo `struct task_struct`. Alla riga 11, si ottiene un puntatore al descrittore di memoria `struct mm_struct`, che descrive lo spazio di indirizzamento di un processo. Tra i campi di questo descrittore, alla riga 12, si accede al campo `exe_file` di tipo `struct file`, che definisce il descrittore di un file aperto. Di tale descrittore, alla riga 13 si accede all'oggetto `f_path` di tipo `struct path`, contenente il campo `dentry` di tipo `struct dentry`. Quest'ultimo, nel campo `d_name` contiene il nome del file eseguibile che ci occorre sotto forma

Codice 4.2 Funzione `pid_to_exe_name()` che restituisce il nome del file eseguibile di un processo in esecuzione dato il suo PID (ambiente Linux desktop).

```
1  const char* pid_to_exe_name(int pid) {
2    struct pid* spid;
3    struct task_struct* task;
4    struct mm_struct *mm;
5    struct qstr fname;
6    struct file * exe_file;
7    struct dentry * dentry;
8
9    if (!(spid = find_get_pid(pid))) return NULL;
10   if (!(task = pid_task(spid, PIDTYPE_PID))) return NULL;
11   if (!(mm = get_task_mm(task))) return NULL;
12   if (!(exe_file = mm->exe_file)) return NULL;
13   if (!(dentry = exe_file->f_path.dentry)) return NULL;
14   fname = dentry->d_name;
15   return fname.name;
16 }
```

di oggetto `struct qstr` (riga 14): il `char*` corrispondente è contenuto nel suo campo `name` che, finalmente, viene restituito dalla funzione.

In ambiente Android il mapping tra PID e nome applicazione è leggermente più complesso. Le applicazioni, infatti, non sono native, bensì vengono eseguite dalla Dalvik Virtual Machine, di cui esiste un'istanza per ogni applicazione. La funzione `pid_to_exe_name()` presentata nel codice 4.2, dunque, restituirebbe sempre il nome del file eseguibile della Dalvik VM. Una versione modificata di questa funzione, che sfrutta il fatto che, nel kernel utilizzato da Android, la voce `/proc/<pid>/cmdline` contiene il package name dell'applicazione, è riportata nel codice 6.2.

4.4 Supporto alle policy per applicazione

Configurando opportunamente la Control Entity, l'utente può definire una serie di policy per ciascuna applicazione. Scenari applicativi tipici sono impedire a un'applicazione di usare una o più interfacce di rete, per motivi di prestazioni o di tariffazione del traffico, oppure forzare un'applicazione ad utilizzare una certa interfaccia di rete.

Questi scenari possono concretizzarsi scegliendo opportunamente il tunnel su cui instradare una o più connessioni dell'applicazione. Come descritto in precedenza, l'instradamento dei pacchetti di una connessione su un tunnel viene determinato dalle regole presenti nella PAFT. La soluzione proposta, pertanto, consiste nel fornire al modulo `xt_UPMT` la possibilità di inserire regole nella PAFT, impostando il tunnel associato a un socket in modo da soddisfare la politica scelta.

Il modulo `xt_UPMT`, dunque, dovrà essere provvisto di una lista delle applicazioni per cui è in effetto una policy stabilita dall'utente, che associ il nome dell'applicazione al tunnel da utilizzare. La determinazione dello specifico tunnel che permetta di soddisfare la politica è demandata alla Control Entity. Tale lista, denominata `apps`, è implementata come una lista collegata di oggetti `struct app_node`, ciascuno definito dalla seguente struttura:

```
1 struct app_node {
2   struct list_head list;
3   char appname[MAX_APPNAME_LENGTH];
4   unsigned int tid;
5 };
```

dove la costante `MAX_APPNAME_LENGTH` è pari a 64. Questa lista è implementata utilizzando la API per le liste fornita dal kernel Linux, descritta nella sezione 3.3. L'utilizzo di questa lista da parte del modulo verrà illustrato nella sottosezione 5.3.3.

Similmente, per le politiche che negano a un'applicazione di utilizzare un'interfaccia, il modulo `xt_UPMT` mantiene una lista denominata `no_upmt` composta da oggetti di tipo `struct no_upmt_node`, ciascuno definito dalla seguente struttura:

```
1 struct no_upmt_node {
2   struct list_head list;
3   char appname[MAX_APPNAME_LENGTH];
4 };
```

L'utilizzo di questa lista da parte del modulo verrà illustrato nella sottosezione 5.3.4.

In luogo di queste liste, si sarebbero potute implementare delle hash table, la cui complessità temporale per operazioni di ricerca è costante, contrariamente alle liste per cui è lineare. Le hash table sono tipicamente implementate come un array di k liste chiamate *bucket*: una funzione di hashing mappa una chiave, che identifica univocamente un oggetto da inserire, in un valore compreso tra $0 \leq i < k$ corrispondente al bucket da utilizzare. In caso di collisione, l'oggetto viene aggiunto in coda al bucket i. La complessità temporale della ricerca, dunque, non dipende direttamente dal numero di elementi, ma solo dalla qualità della funzione di hashing scelta e dal numero di bucket. Una hash table, dunque, risulta conveniente per un gran numero di entry. Si può ragionevolmente prevedere, tuttavia, che il numero di applicazioni per cui l'utente vorrà definire politiche personalizzate o che non vorrà gestire tramite UPMT non supererà qualche decina, pertanto l'uso di hash table per numeri di elementi così piccoli potrebbe addirittura essere controproducente dal punto di vista delle prestazioni, a causa della complessità aggiunta dal calcolo della funzione di hashing e dall'iterazione sui bucket in caso di collisione.

4.5 Interazione con la Control Entity

Finora non si è illustrato come il modulo `xt_UPMT` possa comunicare con la Control Entity, che è un programma user-mode scritto in Java.

Per la comunicazione con processi user-space, il kernel può usare un socket Netlink, nella variante Generic Netlink: il programmatore deve definire una famiglia generica e dotarla di comandi e attributi per realizzare il protocollo desiderato. Il processo user-space che interagirà con il modulo tramite il socket Netlink deve allocare un socket nel seguente modo:

```
1 int sd = socket(AF_NETLINK, SOCK_RAW, NETLINK_GENERIC);
```

Ciò, però, è possibile solo utilizzando l'API standard dei socket. Java, in particolare, non supporta nativamente i socket raw. Esistono alcune librerie per aggiungere tale supporto quali [11], ma utilizzano Java Native Interface e, quindi, codice nativo che renderebbe difficoltoso

il futuro porting su Android. In ogni modo, anche utilizzando tali librerie, non viene offerto alcun supporto alla costruzione e il parsing di messaggi generic netlink.

Per questi motivi, si è scelto di interporre un componente aggiuntivo tra il modulo `xt_UPMT` e la Control Entity, denominato `upmt-appmon`. Questo è un programma user-space scritto in C dotato di un socket netlink per la comunicazione con il modulo tramite messaggi generic netlink, e di due socket UDP classici per la comunicazione con la Control Entity:

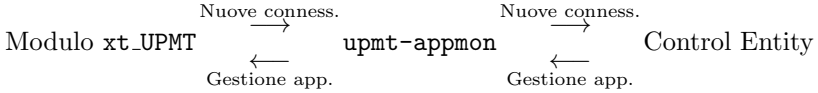
$$\text{Modulo } \texttt{xt\_UPMT} \overset{\text{Socket Netlink}}{\rightleftarrows} \texttt{upmt-appmon} \overset{\text{Socket UDP}}{\rightleftarrows} \text{Control Entity}$$

La scelta di utilizzare socket UDP garantisce la massima compatibilità. In generale, una soluzione più efficiente sarebbe stata l'uso degli *UNIX domain socket*, un meccanismo di IPC (InterProcess Communication) per processi in esecuzione sulla stessa macchina locale. Gli UNIX domain socket utilizzano un file di tipo socket, memorizzato su disco, come identificatore del socket stesso, contrariamente a socket TCP o UDP che usano indirizzi IP e numeri di porta. L'implementazione degli UNIX domain socket nel kernel è caratterizzata da minore overhead rispetto ai normali socket di rete: non è necessario invocare lo stack IP, non è necessario suddividere i dati in datagrammi, non ci sono pacchetti IP di cui calcolare il checksum, ecc. Pur possedendo questi numerosi vantaggi, non è disponibile un binding standard in Java per gli UNIX domain socket, ma è necessario ricorrere a librerie esterne. Queste, come nel caso delle librerie per i socket raw necessari per Netlink, sfruttano codice nativo, difficilmente portabile su Android.

Un altro candidato per la gestione della comunicazione tra `upmt-appmon` e la Control Entity è *DBUS*. Questo meccanismo di IPC, molto diffuso nelle moderne distribuzioni Linux per desktop, è stato preso in considerazione poiché la Control Entity utilizza già il binding Java DBUS per l'interazione con il Network Manager. Tuttavia, è stata infine scartata in ottica di portabilità su Android: esso, infatti, dispone di apposite API implementate nativamente all'interno della piattaforma per l'interazione con le interfacce di rete; inoltre, DBUS non è affatto utilizzato in Android.

In base a queste considerazioni, appare chiaro come `upmt-appmon` ricopra il ruolo di *message broker*, ossia si occupi di tradurre i messaggi dal formato generic netlink al formato previsto dalla Control Entity e viceversa. Quest'ultimo formato si basa su JSON (JavaScript Object Notation) [12], un linguaggio per lo scambio di dati leggero e human-readable. Tale scelta deriva dal fatto che esistono numerose librerie per i più diffusi linguaggi di programmazione, nel nostro caso siamo interessati a C e Java, che sollevano il programmatore dalla necessità di costruire manualmente i messaggi concatenando stringhe, effettuando escape delle sequenze di caratteri riservate ecc., e similmente permettono la validazione automatica della sintassi dei messaggi. Nella Control Entity, inoltre, si utilizza una libreria, denominata `JavaBean2JSON` [13], sviluppata internamente al Netgroup di Tor Vergata, per la serializzazione e deserializzazione automatica di oggetto Java (più precisamente, di un JavaBean) in un messaggio JSON. Il linguaggio C non supporta nativamente JSON, ma esistono numerose librerie esterne. La scelta di quale da adottare per `upmt-appmon` è ricaduta su cJSON [14], una libreria auto-contenuta composta da due soli file sorgente ed estremamente compatta (circa 28 KB).

Dal punto di vista dei messaggi scambiati, si ha:



Il formato dei messaggi JSON e dei messaggi Netlink è descritto, rispettivamente, nella sezione 5.2 e nella sottosezione 5.3.1.

5 Implementazione della soluzione

5.1 Visione architetturale

In figura 5.1 è riportata una visione d'insieme dei componenti che compongono la piattaforma UPMT e le loro interazioni. I blocchi in giallo costituiscono il contributo apportato da questo lavoro di tesi. La figura 2.9 illustra come i componenti realizzati si collocano nel processo di trasmissione di un pacchetto. Come si può vedere dalla 2.10, i componenti realizzati non intervengono alla ricezione di un pacchetto. Lo scopo di tali componenti, infatti, è quello di inoltrare opportunamente i flussi applicativi sui tunnel: una volta che vengono inserite nella PAFT le regole necessarie, l'unico componente che interviene nella ricezione di un pacchetto è il modulo `upmt`.

5.2 Comunicazione tra Control Entity e `upmt-appmon`

Come riportato in sezione 4.5, i messaggi che Control Entity e `upmt-appmon` si scambiano sono in JSON. Si individuano due categorie di messaggi: i messaggi inviati dalla Control Entity a `upmt-appmon`, per la configurazione di politiche per applicazione, e i messaggi inviati da `upmt-appmon` alla Control Entity, per la notifica di nuove connessioni.

Ogni messaggio della prima categoria è composto da un campo `msgtype` che ne denota il tipo, da un campo `command` che denota il comando da eseguire e da altri argomenti che dipendono dallo specifico tipo e comando del messaggio.

Come riportato nel file header `upmt_conntracker_appmon.h`, i valori possibili per il campo `msgtype` e il campo `command` sono definiti, rispettivamente, nell'`enum upmt_msgtype` e nell'`enum upmt_cmd`:

```
1 enum upmt_msgtype {
2   MSG_TYPE_APP,         /* = 0 */
3   MSG_TYPE_NO_UPMT,     /* = 1 */
4   MSG_TYPE_APP_FLOW,    /* = 2 */
5 };
6
7 enum upmt_cmd {
```

Figura 5.1 Architettura della piattaforma UPMT. In giallo sono evidenziati i contributi di questo lavoro di tesi.

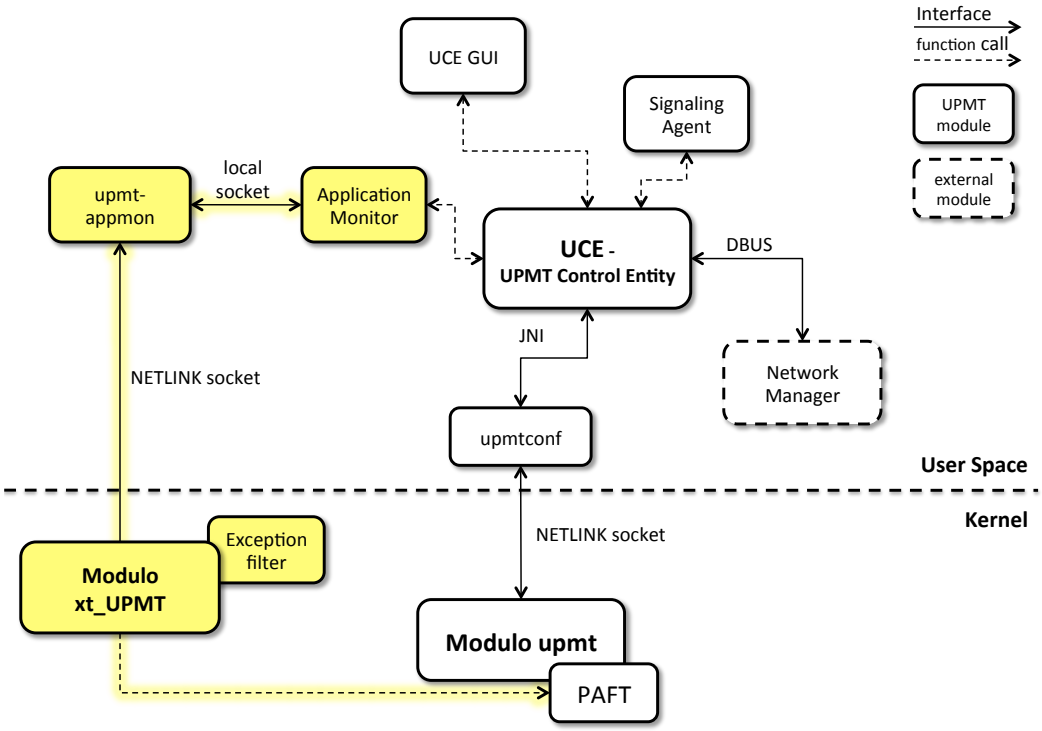


Tabella 5.1 Messaggi JSON dalla Control Entity a upmt-appmon: argomenti obbligatori.

		Argomenti		
msgtype	command	appname	tid	daddr
MSG_TYPE_APP	CMD_ADD	⋆	⋆	
MSG_TYPE_APP	CMD_RM	⋆	⋆	
MSG_TYPE_APP	CMD_FLUSH_LIST			
MSG_TYPE_APP	CMD_SET_DEFAULT_TID			⋆
MSG_TYPE_NO_UPMT	CMD_ADD	⋆		
MSG_TYPE_NO_UPMT	CMD_RM	⋆		
MSG_TYPE_NO_UPMT	CMD_FLUSH_LIST			
MSG_TYPE_APP_FLOW	CMD_ADD		⋆	⋆
MSG_TYPE_APP_FLOW	CMD_RM		⋆	⋆
MSG_TYPE_APP_FLOW	CMD_FLUSH_LIST			

```
8   CMD_ADD,              /* = 0 */
9   CMD_RM,               /* = 1 */
10  CMD_FLUSH_LIST,       /* = 2 */
11  CMD_SET_DEFAULT_TID   /* = 3 */
12 };
```

I messaggi di tipo MSG_TYPE_APP operano sulla lista apps (politiche per applicazione), i messaggi di tipo MSG_TYPE_NO_UPMT operano sulla lista no_upmt (eccezioni per applicazione), infine, i messaggi di tipo MSG_TYPE_APP_FLOW operano sulla lista app_flow (implementazione futura degli scenari UPMT Fixed Host ed End to End).

Il comando CMD_ADD aggiunge un oggetto alla lista selezionata dal tipo del messaggio, il comando CMD_RM rimuove un oggetto dalla lista, il comando CMD_FLUSH_LIST rimuove tutti gli oggetti dalla lista, infine il comando CMD_SET_DEFAULT_TID imposta il tunnel di default.

È inoltre disponibile una classe di messaggi di debug, che forzano il modulo xt_UPMT a stampare il contenuto della lista specificata sul kernel log:

```
1 // Stampa il contenuto della lista apps
2      {"debug":0}
3 // Stampa il contenuto della lista no_upmt
4      {"debug":1}
5 // Stampa il contenuto della lista app_flow
6      {"debug":2}
```

Nella tabella 5.1 sono riportati gli argomenti richiesti per ogni coppia ammessa di valori msgtype/command. Di seguito sono riportati alcuni messaggi di esempio:

```
1 // Aggiungi alla lista apps: wget con tunnel id 1
2      {"msgtype":0, "command":0, "appname":"wget", "tid":1}
3 // Rimuovi dalla lista apps: wget con tunnel id 1
4      {"msgtype":0, "command":1, "appname":"wget", "tid":1}
5 // Svuota lista apps
6      {"msgtype":0, "command":2}
7 // Imposta il tunnel id di default a 2
8      {"msgtype":0, "command":3, "tid":2}
```

```
9
10 // Aggiungi alla lista no_upmt: nc:
11      {"msgtype":1, "command":0, "appname":"nc"}
12 // Rimuovi dalla lista no_upmt: nc:
13      {"msgtype":1, "command":1, "appname":"nc"}
14 // Svuota lista no_upmt
15      {"msgtype":1, "command":2}
16
17 // Aggiungi alla lista app_flow: ip dst 8.8.8.8 con tunnel id 3
18      {"msgtype":2, "command":0, "daddr":"8.8.8.8", "tid":3}
19 // Rimuovi dalla lista app_flow: ip dst 8.8.8.8 con tunnel id 3
20      {"msgtype":2, "command":1, "daddr":"8.8.8.8", "tid":3}
21 // Svuota lista app_flow
22      {"msgtype":2, "command":2}
```

I messaggi di notifica di una nuova connessione inviati da `upmt-appmon` alla Control Entity, invece, sono composti da un oggetto `key` che rappresenta la quintupla che identifica il socket, da un campo `appname` contenente il nome dell'applicazione e da un campo `tid` contenente il numero di tunnel a cui il socket è attualmente assegnato. Un esempio di messaggio di questo tipo è il seguente[1]:

```
1 {
2    "key":{
3       "proto":"tcp",
4       "dstIp":"74.125.232.115",
5       "vipa":"5.6.7.8",
6       "dstPort":80,
7       "srcPort":37182
8    },
9    "appname":"chrome",
10   "tid":2
11 }
```

5.3 Il modulo `xt_UPMT`

Il modulo `xt_UPMT` è uno dei componenti chiave di UPMT, in quanto fornisce ad esso funzionalità *per-application*. Questo modulo, infatti, realizza l'associazione tra socket di rete, gestiti tramite il modulo `upmt`, e applicazioni. Ha inoltre il compito di gestire le politiche di uso della rete e di handover in modo *application-aware*.

5.3.1 Comunicazione con `upmt-appmon` tramite Generic Netlink

Come già detto, i messaggi JSON ricevuti da `upmt-appmon` vengono tradotti nei corrispondenti messaggi Netlink e inviati al modulo.

Per l'utilizzo di Generic Netlink, il modulo `xt_UPMT` definisce una famiglia generic netlink dal nome `UPMT_APPMON`:

```
1 struct genl_family upmt_appmgr_gnl_family = {
```

[1] Gli spazi e le interruzioni di linea sono stati aggiunti per migliorare la leggibilità del testo, ma non sono presenti nel messaggio originale.

```
2   .id = GENL_ID_GENERATE,
3   .hdrsize = 0,
4   .name = UPMT_GNL_FAMILY_NAME,          /* "UPMT_APPMON" */
5   .version = UPMT_GNL_FAMILY_VERSION,    /* 1 */
6   .maxattr = UPMT_APPMON_A_MAX,
7 };
```

che viene registrata nella funzione di inizializzazione del modulo:

```
1 // Generic Netlink family and commands registration
2 if (genl_register_family(&upmt_appmgr_gnl_family)) {
3   printk("\n\t upmt_genl_register - unable to register upmt_appmgr_gnl_family");
4   goto fail_genl_register_family;
5 }
```

Questa famiglia prevede i seguenti comandi, definiti in `upmt_conntracker_appmon.h`:

```
1  enum UPMT_GNL_APPMON_COMMANDS {
2    UPMT_APPMON_C_UNSPEC,
3    UPMT_APPMON_C_ASSOCIATE,   /* associazione tra modulo e upmt-appmon */
4
5    UPMT_APPMON_C_APP,         /* operazioni su lista apps */
6    UPMT_APPMON_C_NO_UPMT,     /* operazioni su lista no_upmt */
7    UPMT_APPMON_C_APP_FLOW,    /* operazioni su lista app_flow */
8
9    UPMT_APPMON_C_NEW_CONN,    /* nuova connessione */
10
11   UPMT_APPMON_C_DUMP_LIST,   /* dump lista su kernel log */
12
13   __UPMT_APPMON_C_MAX
14 };
15 #define UPMT_APPMON_C_MAX (__UPMT_APPMON_C_MAX - 1)
```

Gli attributi supportati dai comandi sono:

```
1  enum UPMT_GNL_APPMON_ATTRIBUTES {
2    UPMT_APPMON_A_UNSPEC,
3    UPMT_APPMON_A_ASSOCIATE,     /* magic number per associazione tra modulo e upmt-appmon */
4
5    UPMT_APPMON_A_APP,           /* struct upmt_app_msg */
6    UPMT_APPMON_A_NO_UPMT,       /* struct upmt_no_upmt_msg */
7    UPMT_APPMON_A_APP_FLOW,      /* struct upmt_app_flow_msg */
8
9    UPMT_APPMON_A_NEW_CONN,      /* struct upmt_new_conn_data */
10
11   UPMT_APPMON_A_DUMP_LIST_ID,  /* numero lista da stampare */
12
13   __UPMT_APPMON_A_MAX
14 };
15 #define UPMT_APPMON_A_MAX (__UPMT_APPMON_A_MAX - 1)
```

e i loro tipi di dato, definiti in `xt_UPMT.c`, sono:

```
1 struct nla_policy upmt_appmgr_genl_policy[UPMT_APPMON_A_MAX + 1] = {
2   [UPMT_APPMON_A_UNSPEC]      = { .type = NLA_BINARY },
3   [UPMT_APPMON_A_ASSOCIATE]   = { .type = NLA_U32 },
4   [UPMT_APPMON_A_APP]       = { .type = NLA_BINARY },
```

Tabella 5.2 Mapping tra messaggi JSON e comandi Netlink.

`msgtype` JSON	Comando Netlink	Payload
`MSG_TYPE_APP`	`UPMT_APPMON_C_APP`	`struct upmt_app_msg`
`MSG_TYPE_NO_UPMT`	`UPMT_APPMON_C_NO_UPMT`	`struct upmt_no_upmt_msg`
`MSG_TYPE_APP_FLOW`	`UPMT_APPMON_C_APP_FLOW`	`struct upmt_app_flow_msg`
(non presente)	`UPMT_APPMON_C_NEW_CONN`	`struct upmt_new_conn_data`

```
  [UPMT_APPMON_A_NO_UPMT]     = { .type = NLA_BINARY },
  [UPMT_APPMON_A_APP_FLOW]    = { .type = NLA_BINARY },
  [UPMT_APPMON_A_NEW_CONN]    = { .type = NLA_BINARY },
  [UPMT_APPMON_A_DUMP_LIST_ID]  = { .type = NLA_U32 },
};
```

Il tipo `NLA_U32` denota un intero senza segno a 32 bit, mentre il tipo `NLA_BINARY` indica una sequenza di byte grezzi. Gli attributi `UPMT_APPMON_A_APP`, `UPMT_APPMON_A_NO_UPMT`, `UPMT_APPMON_A_APP_FLOW` e `UPMT_APPMON_A_NEW_CONN` conterranno, nell'ordine, le seguenti struct:

```
struct upmt_app_msg {
  enum upmt_cmd command;
  char appname[MAX_APPNAME_LENGTH];
  int tid;
};

struct upmt_no_upmt_msg {
  enum upmt_cmd command;
  char appname[MAX_APPNAME_LENGTH];
};

struct upmt_app_flow_msg {
  enum upmt_cmd command;
  unsigned int daddr;
  int tid;
};

struct upmt_new_conn_data {
  struct upmt_key key;
  char appname[MAX_APPNAME_LENGTH];
  int tid;
};
```

In tabella 5.2 è riportato il mapping tra messaggi JSON e comandi Netlink.

Per ciascun comando, è necessario definire una funzione handler che verrà invocata dal controller Netlink ad ogni ricezione di quel comando. Per far ciò è necessario dichiarare un apposito descrittore:

```
static struct genl_ops upmt_gnl_ops_app_cmd = {
  .cmd = UPMT_APPMON_C_APP,
  .flags = 0,
  .policy = upmt_appmgr_genl_policy,
  .doit = app_cmd_handler,      /* funzione di callback */
  .dumpit = NULL,
```

```
7 };
```

e registrarlo all'avvio del modulo:

```
1 if (genl_register_ops(&upmt_appmgr_gnl_family, &upmt_gnl_ops_app_cmd)) {
2   printk("\n\t upmt_genl_register - unable to register upmt_gnl_ops_app_cmd");
3   goto fail_genl_register_ops_app_cmd;
4 }
```

Nel codice 5.1 è riportata la funzione handler `app_cmd_handler` che gestisce i comandi `UPMT_APPMON_C_APP`. Il messaggio netlink contenente tale comando viene passato sotto forma di puntatore tramite il parametro `struct genl_info *info`. La riga 6 mostra come estrarre dal messaggio l'oggetto `struct upmt_app_msg` contenuto nell'attributo `UPMT_APPMON_A_APP` utilizzando la funzione `extract_nl_attr()`. Tale funzione è così definita:

```
1 void *extract_nl_attr(const struct genl_info *info, const int atype) {
2   struct nlattr *na;
3   void *data = NULL;
4   na = info->attrs[atype];
5   if (na) data = nla_data(na);
6   return data;
7 }
```

La riga 4 accede all'array degli attributi contenuti nel messaggio indirizzandolo tramite il valore di attributo passato, la riga 5 invoca la funzione `nla_data()` definita nel file header di Linux `include/net/netlink.h` per ottenere un puntatore ai dati dell'attributo.

Per notificare `upmt-appmon` della rilevazione di nuove connessioni, è stata definita la funzione `send_conn_notif()`, il cui sorgente è riportato nel codice 5.2, che riceve come parametro un oggetto `struct upmt_new_conn_data` contenente i dati da inviare sul socket netlink. Tale funzione usa l'API di generic netlink per costruire e inviare un messaggio. Alla riga 5, la funzione `genlmsg_new()` alloca un nuovo socket buffer per contenere il messaggio: la costante `NLMSG_GOODSIZE` specifica una dimensione predefinita quando la dimensione del messaggio non sia ancora nota. Alla riga 11 si usa la funzione `genlmsg_put()` per definire il comando del messaggio netlink[2]. Il suo prototipo è:

```
1 static inline void *genlmsg_put(struct sk_buff *skb, u32 pid, u32 seq, struct genl_family *
      family, int flags, u8 cmd);
```

i cui parametri sono:

`skb` Il socket buffer che contiene il messaggio

`pid` Il PID Netlink del destinatario del messaggio, nel nostro caso `upmt-appmon`

`seq` Numero di sequenza

`family` Descrittore della famiglia Generic netlink

`flags` Flag del messaggio

`cmd` Comando generic netlink

[2]Più in generale, tale funzione aggiunge un header generic netlink a un messaggio.

Codice 5.1 Funzione handler per la gestione di comandi `UPMT_APPMON_C_APP`.

```
1  int app_cmd_handler(struct sk_buff *skb, struct genl_info *info) {
2    struct upmt_app_msg* cmd;
3    struct app_node *tmp;
4    struct list_head *pos, *q;
5
6    cmd = (struct upmt_app_msg*) extract_nl_attr(info, UPMT_APPMON_A_APP);
7
8    if (cmd->command == CMD_ADD) {
9      printk("app_cmd_handler: gestione comando CMD_ADD\n");
10     list_for_each_safe(pos, q, &(apps.list)) {
11       tmp = list_entry(pos, struct app_node, list);
12       if (strcmp(cmd->appname, tmp->appname) == 0) {
13         list_del(pos);
14         kfree(tmp);
15         break;
16       }
17     }
18
19     tmp = (struct app_node *) kmalloc(sizeof(struct app_node), GFP_KERNEL);
20     strncpy(tmp->appname, cmd->appname, MAX_APPNAME_LENGTH);
21     tmp->tid = cmd->tid;
22     list_add_tail(&(tmp->list), &(apps.list));
23     printk("added <%s,%d> to app list\n", tmp->appname, tmp->tid);
24   }
25   else if (cmd->command == CMD_RM) {
26     printk("app_cmd_handler: gestione comando CMD_RM\n");
27     list_for_each_safe(pos, q, &(apps.list)) {
28       tmp = list_entry(pos, struct app_node, list);
29       if (strcmp(cmd->appname, tmp->appname) == 0) {
30         list_del(pos);
31         kfree(tmp);
32       }
33     }
34   }
35   else if (cmd->command == CMD_FLUSH_LIST) {
36     printk("app_cmd_handler: gestione comando CMD_FLUSH_LIST\n");
37     list_for_each_safe(pos, q, &(apps.list)){
38       tmp = list_entry(pos, struct app_node, list);
39       list_del(pos);
40       kfree(tmp);
41     }
42   }
43   else if (cmd->command == CMD_SET_DEFAULT_TID) {
44     printk("app_cmd_handler: gestione comando CMD_SET_DEFAULT_TID\n");
45     default_tid = cmd->tid;
46   }
47   return 0;
48 }
```

Codice 5.2 Funzione `send_conn_notif()` per la notifica di nuove connessioni a `upmt-appmon`.

```
1  int send_conn_notif(struct upmt_new_conn_data *data) {
2    struct sk_buff *skb;
3    void *skb_head;
4
5    skb = genlmsg_new(NLMSG_GOODSIZE, GFP_ATOMIC);
6    if (skb == NULL){
7      printk("\n\t send_response_message - unable to allocate skb");
8      return -1;
9    }
10
11   skb_head = genlmsg_put(skb, appmon_pid, conn_notif_seqno++, &upmt_appmgr_gnl_family, 0,
         UPMT_APPMON_C_NEW_CONN);
12   if (skb_head == NULL) {
13     printk("\n\t send_response_message - unable to allocate skb_head");
14     return -ENOMEM;
15   }
16
17   if(nla_put(skb, UPMT_APPMON_A_NEW_CONN, sizeof(struct upmt_new_conn_data), data) != 0){
18     printk("\n\t send_response_message - unable to put UPMT_A_MSG_TYPE attribute");
19     return -1;
20   }
21
22   genlmsg_end(skb, skb_head);
23
24   if(genlmsg_unicast(&init_net, skb, appmon_pid ) != 0) {
25     printk("\n\t send_response_message - unable to send response");
26     return -1;
27   }
28   return 0;
29 }
```

e che restituisce un puntatore all'header modificato del messaggio. Alla riga 17, viene invocata la funzione `nla_put()` per aggiungere un attributo e il suo valore al messaggio. La funzione `genlmsg_end()`, chiamata alla riga 24, finalizza il messaggio generic netlink, che viene inviato dalla funzione `genlmsg_unicast()`, chiamata alla riga 24, al PID netlink di `upmt-appmon`.

Rimane da illustrare come il modulo `xt_UPMT` reperisca il PID netlink dell'`upmt-appmon`. A tale scopo, il modulo definisce il comando `UPMT_APPMON_C_ASSOCIATE` e registra la funzione handler associata `associate_cmd_handler()`. Tale funzione, riportata nel codice 5.3 non fa altro che estrarre il PID dell'`upmt-appmon` dal campo `snd_pid` del descrittore del messaggio netlink ricevuto `info` di tipo `struct genl_info`, e lo salva nella variabile globale `appmon_pid`.

5.3.2 Rilevamento delle connessioni

Come descritto nella sezione 4.2, il rilevamento di una nuova connessione è affidato al modulo conntrack di Netfilter, e gestito tramite la scrittura di un target personalizzato. Il codice di tale target costituisce il cuore del modulo `xt_UPMT`.

Codice 5.3 Funzione `associate_cmd_handler()` per il reperimento del PID netlink dell'`upmt-appmon`.

```
1 int associate_cmd_handler(struct sk_buff *skb, struct genl_info *info) {
2   int magic = *(int*) extract_nl_attr(info, UPMT_APPMON_A_ASSOCIATE);
3   appmon_pid = info->snd_pid;    //save pid of application monitor
4   printk("UPMT-AppMon with PID %d associated with magic number %d\n", appmon_pid, magic);
5   return 0;
6 }
```

Per registrare un nuovo target, è innanzitutto necessario definire il descrittore corrispondente:

```
1 static struct xt_target upmt_tg_reg[] __read_mostly = {{
2   .name   = "UPMT",
3   .revision   = 0,
4   .family   = NFPROTO_UNSPEC,
5   .target   = upmt_tg,
6   .me   = THIS_MODULE,
7 }};
```

Il campo `name` è il nome del target, ossia l'argomento da passare all'opzione `-j` di iptables. Il campo `target` è un puntatore alla funzione di callback da invocare quando il target deve essere applicato. Tale funzione ha il seguente prototipo:

```
1 unsigned int nf_target(struct sk_buff **pskb, const struct xt_action_param *tp);
```

dove il parametro `pskb` è un doppio puntatore al socket buffer che contiene il pacchetto.

Questo descrittore viene registrato dalla funzione di inizializzazione del modulo:

```
1 // Register the 'UPMT' xtables target
2 if (xt_register_targets(upmt_tg_reg, ARRAY_SIZE(upmt_tg_reg))) {
3   printk("\n\t xt_register_targets - unable to register UPMT target");
4   goto fail_xt_register_targets;
5 }
```

Nel codice 5.4 è riportata l'implementazione del target `upmt_tg` nel modulo. Alla riga 10 viene invocata la funzione `pid_to_exe_name()`, passandogli come parametro il TGID estratto dal socket buffer, per ottenere il nome dell'applicazione che ha aperto la connessione. Le righe 15 e 16 verificano che il pacchetto non abbia il marchio `NO_UPMT_MARK`, che indica che il pacchetto non deve essere inviato nei tunnel. La riga 18 calcola la key UPMT a partire dal socket buffer, valorizzando i campi della `struct upmt_key`, definita come:

```
1 struct upmt_key{
2   unsigned int proto:8;
3   unsigned int saddr:32;
4   unsigned int daddr:32;
5   unsigned int dport:16;
6   unsigned int sport:16;
7 };
```

Nel blocco 21 ÷ 29 si verifica se il pacchetto fa parte di una connessione che debba essere trattata secondo gli scenari Fixed Host o End to End. In particolare si verifica, per ogni

voce < ip dst, tid > della lista app_flow, se l'indirizzo IP di destinazione estratto dal pacchetto corrisponde a ip dst, allora deve essere inserita una regola nella PAFT (riga 25) per inoltrare tale pacchetto verso il tunnel tid. Il blocco successivo, alle righe 33 ÷ 46, verifica se per l'applicazione che ha aperto la connessione esistono politiche definite nella lista apps, ossia se questa contiene una voce < appname, tid >, dove appname è il nome di applicazione restituito dalla funzione pid_to_exe_name(). Se tale voce esiste, si inserisce una regola nella PAFT (riga 44) per inoltrare i pacchetti di tale connessione verso il tunnel tid. Infine, alla riga 45, si invia il messaggio netlink di nuova connessione all'upmt-appmon.

5.3.3 Gestione delle policy per applicazione

La lista apps permette al modulo di impostare i tunnel per ciascuna applicazione. Questa operazione di basso livello è di supporto al componente della Control Entity che effettua le decisioni su come instradare il traffico delle applicazioni in base alle politiche definite dall'utente e allo stato delle interfacce di rete.

All'avvio della Control Entity, questa esamina il file di configurazione delle politiche e, per ogni applicazione definita, invia un messaggio di aggiunta applicazione con il tunnel corrispondente all'interfaccia specificata.

Inoltre, ad ogni connessione o disconnessione di un'interfaccia di rete, il decision manager riesamina il file delle politiche per determinare di quali connessioni si può eseguire l'handover. In caso di disconnessione, in particolare, per le applicazioni che avevano delle connessioni aperte di cui è permesso (da politiche) effettuare l'handover, viene inviato un messaggio di aggiunta applicazione contenente il nuovo tunnel su cui instradare le future nuove connessioni dall'applicazione e, contestualmente, si modifica la PAFT tramite upmtconf per spostare i socket già stabiliti sul nuovo tunnel.

Un'altra classe di policy, impostata all'avvio della Control Entity, è quella relativa allo scenario Fixed Host. In questo scenario, si usa la lista app_flow, contenente un insieme di indirizzi IP e per ciascuno di essi un numero di tunnel. In questo scenario, infatti, si desidera che i socket creati verso lo stesso Fixed Host siano direttamente instradati verso quest'ultimo senza passare per gli Anchor Node. All'apertura di un socket avente uno dei Fixed Host presenti nella lista come indirizzo IP di destinazione, verrà inserita una regola nella PAFT per inviare i pacchetti relativi nel corretto tunnel.

Infine, nel caso di connessioni che non trovano corrispondenza in nessuna di queste liste, viene inserita una regola nella PAFT per instradare i pacchetti verso un tunnel di default, definito dalla Control Entity in fase di inizializzazione, e utilizzato per tutte le applicazioni che non si desidera gestire singolarmente.

La funzione di gestione della lista apps è riportata nel codice 5.1. Il codice delle funzioni di gestione delle liste no_upmt e app_flow è molto simile e omesso per brevità.

5.3.4 Eccezioni per applicazione e per traffico di rete locale

Come specificato in sezione 4.1, è richiesto che il traffico di rete locale, multicast e broadcast non sia instradato sui tunnel. Questo requisito è ragionevole: nel caso del traffico locale, infatti, l'Anchor Node non avrebbe modo di inoltrare un pacchetto indirizzato a un IP privato. Questa situazione è molto frequente nelle reti domestiche che utilizzano un modem/router

Codice 5.4 Target Netfilter per la gestione di nuove connessioni.

```
1  unsigned int upmt_tg(struct sk_buff **pskb, const struct xt_action_param *tp) {
2    struct sk_buff *skb = *pskb;
3    struct upmt_key key;
4    struct upmt_new_conn_data notif_data;
5    const char *appname;
6
7    if (default_tid == -1) return NF_ACCEPT; // do nothing until we receive a valid default
         tunnel
8    if (skb->tgid <= 0) return NF_ACCEPT;
9
10   if ((appname = pid_to_exe_name(skb->tgid)) == NULL) {
11     printk("upmt_tg: error finding exe name for pid %d\n", skb->tgid);
12     return NF_ACCEPT;
13   }
14
15   if (skb->mark == NO_UPMT_MARK)
16     return NF_ACCEPT;
17
18   calculate_upmt_key(&key, skb);
19
20   // Check whether the destination address of the socket is in the app_flow list. If yes,
         divert the app to the specified tunnel
21   {
22     struct app_flow_node *tmp;
23     list_for_each_entry(tmp, &app_flow.list, list) {
24       if (tmp->daddr == key.daddr) {
25         paft_insert_by_tid(&key, tmp->tid, 1);
26         return NF_ACCEPT;
27       }
28     }
29   }
30
31   // Check if the app is in the apps list. If yes, divert it to its tunnel, otherwise divert
         it to the default tunnel.
32   // Moreover, notify the appmon of the new app.
33   {
34     struct app_node *tmp;
35     notif_data.key = key;
36     strncpy(notif_data.appname, appname, MAX_APPNAME_LENGTH);
37     notif_data.tid = default_tid;
38     list_for_each_entry(tmp, &apps.list, list) {
39       if (strncasecmp(appname, tmp->appname, MAX_APPNAME_LENGTH) == 0) {
40         notif_data.tid = tmp->tid;
41         break;
42       }
43     }
44     paft_insert_by_tid(&notif_data.key, notif_data.tid, 1);
45     send_conn_notif(&notif_data);
46   }
47
48   return NF_ACCEPT;
49 }
```

ADSL, tipicamente dotato di un server DNS forwarding offerto via DHCP alla rete locale. In questo caso, i pacchetti UDP delle query DNS originati da un'applicazione gestita da UPMT verrebbero instradati nel tunnel, ma l'Anchor Node non potrebbe inviare i pacchetti di risposta essendo diretti a un host con indirizzo IP privato appartenente alla LAN originaria.

Il caso del traffico locale può essere facilmente trattato aggiungendo, per ogni interfaccia fisica, una rotta alla tabella di routing `upmt0_table` tramite il comando:

```
1 ip route add PREFIX/NETMASK dev IFACE table upmt0_table
```

dove `PREFIX` è il prefisso di rete, `NETMASK` è la subnet mask della rete espressa in decimale, e `IFACE` è l'interfaccia di rete. Ad esempio, se l'interfaccia di rete eth0 ha indirizzo IP `192.168.1.2` con subnet mask `255.255.255.0`, `PREFIX` varrà `192.168.1.0` e `NETMASK` `24`. Il comando quindi sarà:

```
1 ip route add 192.168.1.0/24 dev eth0 table upmt0_table
```

La tabella di routing `upmt0_table`, dunque, conterrà le seguenti rotte:

```
1 # ip route show table upmt0_table
2 192.168.0.0/24 dev eth0  scope link
3 default dev upmt0  scope link
```

La rotta appena inserita (riga 2), dunque, fa sì che tutti i pacchetti destinati a un host della sottorete attestata sull'interfaccia fisica `eth0` vengano inoltrati attraverso tale interfaccia. Poiché questa rotta ha un prefix certamente più lungo di quello della rotta di default, pari a 0, i pacchetti non vengono inoltrati tramite l'interfaccia virtuale `upmt0` e quindi nei tunnel.

Per la gestione del traffico broadcast, multicast e per le eccezioni per applicazione, invece, è stato implementato un hook Netfilter (vedi sezione 3.4) all'interno del modulo che venga invocato per tutti i pacchetti in uscita. Questo hook dovrà verificare se l'indirizzo di destinazione del pacchetto sia un indirizzo multicast o broadcast e se l'applicazione che ha originato il pacchetto sia presente nella lista `no_upmt`. Nel caso che una di queste condizioni risulti verificata, l'hook deve rieseguire il routing del pacchetto per inoltrarlo verso l'interfaccia di uscita corretta.

Per definire un hook, è necessario allocare un apposito descrittore:

```
1 static struct nf_hook_ops nf_ops_prefilter = {
2   .hook      = upmt_prefilter,
3   .pf        = PF_INET,
4   .hooknum   = NF_INET_LOCAL_OUT,
5   .priority  = NF_IP_PRI_FIRST
6 };
```

L'hook in questione, dunque, avrà come funzione di callback la funzione `upmt_prefilter()` e verrà registrato in `NF_INET_LOCAL_OUT` con priorità massima (`NF_IP_PRI_FIRST`). In fase di inizializzazione del modulo, tale descrittore dovrà essere registrato:

```
1 // Register pre-filter hook
2 if (nf_register_hook(&nf_ops_prefilter)) {
3   printk("\n\t prefilter_register_hook - unable to register prefilter hook");
4   goto fail_register_prefilter;
5 }
```

Codice 5.5 Hook Netfilter per la gestione delle eccezioni per applicazione e per traffico di rete locale.

```
1  static unsigned int upmt_prefilter(unsigned int hooknum, struct sk_buff *skb, const struct
       net_device *in, const struct net_device *out, int (*okfn)(struct sk_buff*)) {
2    const char *appname;
3    struct iphdr * ip = ip_hdr(skb);
4
5    if (skb->tgid <= 0) return NF_ACCEPT;
6    if (out != upmt_dev) return NF_ACCEPT;
7
8    // exception for broadcast
9    if (ipv4_is_lbcast(ip->daddr))
10     goto reroute_and_return;
11
12   // exception for multicast
13   if (ipv4_is_multicast(ip->daddr))
14     goto reroute_and_return;
15
16   if ((appname = pid_to_exe_name(skb->tgid)) == NULL)
17     return NF_ACCEPT;
18
19   {
20     struct no_upmt_node *tmp;
21     list_for_each_entry(tmp, &no_upmt.list, list) {
22       if (strncasecmp(appname, tmp->appname, MAX_APPNAME_LENGTH) == 0)
23         goto reroute_and_return;
24     }
25   }
26   return NF_ACCEPT;
27
28 reroute_and_return:
29   __reroute(skb, out);
30   return NF_ACCEPT;
31 }
```

L'implementazione della funzione `upmt_prefilter()` è riportata nel codice 5.5. La riga 6 verifica che l'interfaccia di uscita sia l'interfaccia virtuale `upmt0`. La riga 9 verifica se l'indirizzo IP di destinazione del pacchetto è un indirizzo broadcast. In caso affermativo, si riesegue il routing del pacchetto. La riga 13 verifica se l'indirizzo IP di destinazione del pacchetto è un indirizzo multicast. In caso affermativo, si riesegue il routing del pacchetto. La riga 16 ricava il nome dell'applicazione dal TGID estratto dal socket buffer. Nel blocco 19 ÷ 25 si itera sulla lista `no_upmt` alla ricerca di una voce con il nome di applicazione corrispondente a quello ricavato. Se tale voce esiste, si riesegue il routing del pacchetto (riga 23), altrimenti si termina accettando il pacchetto (riga 26). La riesecuzione del routing è effettuata dalla funzione `__reroute()` chiamata alla riga 29, e la cui implementazione è riportata nel codice 5.6. In seguito a tale operazione, si termina accettando il pacchetto (riga 30).

Esaminiamo ora il codice della funzione `__reroute()`. Tale funzione riceve in ingresso il socket buffer corrispondente al pacchetto non incapsulato (prefilter) Alla riga 2 si estrae

Codice 5.6 Funzione `__reroute` per eseguire il routing manuale dei pacchetti destinati alla rete locale.

```
1  static void __reroute(struct sk_buff *skb, const struct net_device *odev) {
2    struct iphdr * ip = ip_hdr(skb);
3    struct rtable *rt;
4    struct flowi fl = {
5      .mark = NO_UPMT_MARK,
6      .nl_u = {
7        .ip4_u = {
8          .daddr  = ip->daddr,
9        }
10     },
11   };
12   if (ip_route_output_key(dev_net(odev), &rt, &fl))
13     return;
14
15   if (rt->u.dst.dev == odev) {
16     ip_rt_put(rt);
17     return;
18   }
19
20   skb_dst_drop(skb);
21   skb_dst_set(skb, &rt->u.dst);
22 }
```

l'header IP del pacchetto dal socket buffer. Le righe 4 ÷ 11 costituiscono una serie di parametri da utilizzare per effettuare il routing lookup. La riga 5 forza il routing lookup ad essere eseguito sulla tabella `main`. La riga 12 effettua il routing lookup vero e proprio con i parametri appena preparati e scrive il risultato nella struttura `rt` di tipo `struct rtable`. La riga 15 verifica se l'interfaccia di uscita selezionata dalla decisione di routing sia uguale a quella originaria. In tal caso, il rerouting non ha apportato alcun cambiamento al pacchetto ed è necessario deallocare l'oggetto `rt` e ritornare. A questo punto, è necessario impostare la nuova destinazione nel socket buffer tramite la funzione `skb_dst_set()` (riga 21), non prima però di aver eliminato la destinazione precedente dalla cache delle destinazioni del socket buffer (riga 20).

5.4 Implementazione di `upmt-appmon`

Nella sezione 4.5 è stato introdotto `upmt-appmon`, che opera da message broker tra il modulo `xt_UPMT` e la Control Entity. Descriviamo ora in dettaglio la sua implementazione.

`upmt-appmon` mantiene aperti due socket UDP verso la Control Entity, uno in ricezione e l'altro in trasmissione, i cui descrittori sono, rispettivamente `frommgr_sd` e `tomgr_sd`, e un socket Netlink verso il modulo, il cui descrittore è `nl_sd`. Le porte utilizzate per i socket UDP devono essere passate come argomenti da riga di comando. Tali socket sono inizializzati, nella funzione `main()`, chiamando le funzioni, rispettivamente, `create_appmgr_listen_socket()` (codice 5.7), `create_appmgr_send_socket()` (codice 5.8) e `create_nl_socket()` (codice 5.9).

Codice 5.7 Funzione `create_appmgr_listen_socket()` per la creazione di un socket su cui ricevere i comandi della Control Entity.

```
1  int create_appmgr_listen_socket() {
2    int sock;
3    struct sockaddr_in addr;
4
5    if ( (sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
6      perror("create_appmgr_listen_socket: socket");
7      return -1;
8    }
9    memset((void *) &addr, 0, sizeof(addr));
10   addr.sin_family = AF_INET;
11   addr.sin_port = htons(appmgr_socket_port_frommgr);
12   inet_pton(AF_INET, LOCALHOST_ADDR, &addr.sin_addr);
13   if (bind(sock, (struct sockaddr *)& addr, sizeof(addr)) < 0) {
14     perror("create_appmgr_listen_socket: bind");
15     return -2;
16   }
17   frommgr_sd = sock;
18   return 0;
19 }
```

Codice 5.8 Funzione `create_appmgr_send_socket()` per la creazione di un socket su cui inviare le notifiche di nuove connessioni alla Control Entity.

```
1  int create_appmgr_send_socket() {
2    if ((tomgr_sd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
3      perror("create_appmgr_send_socket: socket");
4      return -1;
5    }
6
7    memset(&tomgr_addr, 0, sizeof(tomgr_addr));
8    tomgr_addr.sin_family = AF_INET;
9
10   if (inet_pton(AF_INET, LOCALHOST_ADDR, & tomgr_addr.sin_addr.s_addr) <= 0) {
11     printf("create_appmgr_send_socket: inet_pton");
12     return 1;
13   }
14   tomgr_addr.sin_port = htons(appmgr_socket_port_tomgr);
15
16   return 0;
17 }
```

Codice 5.9 Funzione `create_nl_socket()` per la comunicazione tramite Netlink con il modulo `xt_UPMT`.

```
1  int create_nl_socket() {
2    int fd;
3    struct sockaddr_nl local;
4
5    fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_GENERIC);
6    if (fd < 0) {
7      perror("Unable to create netlink socket");
8      return -1;
9    }
10
11   memset(&local, 0, sizeof(local));
12   local.nl_family = AF_NETLINK;
13   local.nl_groups = 0; /* no multicast */
14   local.nl_pid  = getpid();
15
16   if (bind(fd, (struct sockaddr *) &local, sizeof(local)) < 0) {
17     close(fd);
18     perror("Unable to bind netlink socket");
19     return -2;
20   }
21   nl_sd = fd;
22   return 0;
23 }
```

Una volta creati questi socket, è necessario ottenere l'identificativo della famiglia netlink assegnato dinamicamente dal controller generic netlink. A tale scopo, si invoca la funzione `init_family_id()` (codice 5.10). In particolare, alla riga 19 si usa la funzione `sendto_fd()` per inviare un messaggio sul socket netlink, e alla riga 22 si usa la funzione `handle_netlink_msg()` per ricevere di un messaggio netlink. Come si può vedere dalle loro implementazioni riportate, rispettivamente, nel codice 5.11 e nel codice 5.12, le funzioni standard `sendto()` e `recv()` possono essere utilizzate anche con socket netlink.

Perché il modulo `xt_UPMT` possa inviare in modo asincrono le notifiche di nuove connessioni sul socket netlink, è necessario che questo conosca il PID dell'istanza di `upmt-appmon` a cui inviare tali messaggi. A tale scopo, `upmt-appmon` è dotato di una procedura di associazione, che consiste nell'invio, tramite un apposito comando, di un magic number, ossia un valore numerico costante. Il modulo, dotato di una funzione handler per tale comando, estrarrà quindi il PID di `upmt-appmon` dal campo mittente del messaggio, e lo userà come destinatario delle future notifiche di nuove connessioni. L'implementazione, semplicissima, della procedura di associazione è riportata nel codice 5.13. Tale funzione invoca immediatamente la funzione `netlink_send_int()` (codice 5.14) per l'invio del magic number.

A questo punto, con tutti i socket pronti e l'associazione effettuata, `upmt-appmon` esegue la chiamata di sistema bloccante `poll` e viene sbloccato quando sul socket di ricezione dalla Control Entity o sul socket netlink sono in arrivo nuovi dati:

```
1 struct pollfd fds[SOCKETS_TO_POLL];
2 fds[0].fd = nl_sd;
3 fds[0].events = POLLIN;
```

Codice 5.10 Funzione `init_family_id()` per ottenere l'identificativo della famiglia assegnato dinamicamente dal controller generic netlink.

```
1 // Probe the controller in genetlink to find the family id for the UPMT_GNL_FAMILY_NAME
      family
2 int init_family_id() {
3   struct genl_msg family_req, ans;
4   int id = -10;
5   struct nlattr *na;
6
7   family_req.n.nlmsg_type = GENL_ID_CTRL;
8   family_req.n.nlmsg_flags = NLM_F_REQUEST;
9   family_req.n.nlmsg_seq = ++seqno;
10  family_req.n.nlmsg_pid = getpid();
11  family_req.n.nlmsg_len = NLMSG_LENGTH(GENL_HDRLEN);
12  family_req.g.cmd = CTRL_CMD_GETFAMILY;
13  family_req.g.version = 0x1;
14
15  na = (struct nlattr *) GENLMSG_DATA(&family_req);
16  set_nl_attr(na, CTRL_ATTR_FAMILY_NAME, UPMT_GNL_FAMILY_NAME, strlen(UPMT_GNL_FAMILY_NAME));
17  family_req.n.nlmsg_len += NLMSG_ALIGN(na->nla_len);
18
19  if (sendto_fd(nl_sd, (char *) &family_req, family_req.n.nlmsg_len) < 0)
20    return -2;
21
22  int ret = handle_netlink_msg(&ans);
23
24  if (ret < 0) return ret;
25  na = (struct nlattr *) GENLMSG_DATA(&ans);
26  na = (struct nlattr *) ((char *) na + NLA_ALIGN(na->nla_len));
27  if (na->nla_type == CTRL_ATTR_FAMILY_ID) {
28    id = *(__u16 *) GENLMSG_NLA_DATA(na);
29    upmt_family_id = id; // save the obtained family id
30    return 0;
31  }
32  return -1;
33 }
```

Codice 5.11 Funzione `sendto_fd()` per l'invio di un buffer di dati su un socket (UDP o netlink).

```
1  int sendto_fd(int s, const char *buf, int bufLen) {
2    struct sockaddr_nl nladdr;
3    int r;
4
5    memset(&nladdr, 0, sizeof(nladdr));
6    nladdr.nl_family = AF_NETLINK;
7
8    while ((r = sendto(s, buf, bufLen, 0, (struct sockaddr *) &nladdr, sizeof(nladdr))) <
         bufLen) {
9      if (r > 0) {
10       buf += r;
11       bufLen -= r;
12     } else if (errno != EAGAIN) return -1;
13   }
14   return 0;
15 }
```

Codice 5.12 Funzione `handle_netlink_msg()` per la ricezione di un messaggio netlink.

```
1  int handle_netlink_msg(struct genl_msg *ans) {
2    memset(ans->buf, 0, MAX_BUF_LEN);
3    int rep_len = recv(nl_sd, ans, sizeof(struct genl_msg), 0);
4
5    if (ans->n.nlmsg_type == NLMSG_ERROR) {
6      printf("Netlink: received NACK -- is module xt_UPMT inserted?\n");
7      return -1;
8    }
9    if (rep_len < 0) {
10     printf("Netlink: error receiving reply message\n");
11     return -2;
12   }
13   if (!NLMSG_OK((&ans->n), rep_len)) {
14     printf("Netlink: invalid reply message received\n");
15     return -3;
16   }
17   return 0;
18 }
```

Codice 5.13 Funzione `associate()` per l'associazione del modulo con un'istanza di `upmt-appmon`.

```
1  int associate() {
2    return netlink_send_int(UPMT_APPMON_C_ASSOCIATE, UPMT_APPMON_A_ASSOCIATE,
         UPMT_ASSOCIATE_MAGIC_NUMBER);
3  }
```

Codice 5.14 Funzione `netlink_send_int()` per l'invio di un comando netlink costituito da un unico attributo numerico.

```
1  int netlink_send_int(int cmd, int attr, int val) {
2    struct nlattr *na;
3
4    req.n.nlmsg_len   = NLMSG_LENGTH(GENL_HDRLEN);
5    req.n.nlmsg_type  = upmt_family_id;
6    req.n.nlmsg_flags   = NLM_F_REQUEST;
7    req.n.nlmsg_seq   = ++seqno;
8    req.n.nlmsg_pid   = getpid();
9    req.g.cmd       = cmd;
10
11   na = (struct nlattr *) GENLMSG_DATA(&req);
12   set_nl_attr(na, attr, &val, sizeof(int));
13   req.n.nlmsg_len += NLMSG_ALIGN(na->nla_len);
14
15   if (sendto_fd(nl_sd, (char *) &req, req.n.nlmsg_len) < 0) return -1;
16
17   return 0;
18 }
```

```
4  fds[1].fd = frommgr_sd;
5  fds[1].events = POLLIN;
6
7  while (poll(fds, SOCKETS_TO_POLL, -1) > 0) {
8    if (fds[0].revents & POLLIN) {      //received netlink message
9      receive_and_dispatch_netlink_msg();
10   }
11   else if (fds[1].revents & POLLIN) {   //received appmgr message
12     receive_and_dispatch_appmgr_msg();
13   }
14 }
```

In questo frammento di codice si è preferita la chiamata di sistema `poll()` rispetto alla più nota `select()` per la sua maggiore leggibilità: le due implementazioni, infatti, sono del tutto equivalenti [15].

Il codice 5.15 illustra la gestione di messaggi netlink in ingresso usando la funzione `handle_netlink_msg()` per la ricezione e la funzione `newconn_handler()` per la gestione dell'evento. Quest'ultima funzione, riportata nel codice 5.16, estrae l'oggetto di tipo `struct upmt_new_conn_data` dal messaggio generic netlink e assembla il messaggio JSON da inviare sul socket UDP verso la Control Entity. Notare come alle righe 10 e 11 il numero di protocollo contenuto nel campo `key.proto` della `struct upmt_new_conn_data` venga convertito da intero a stringa (ad es. 6 corrisponde a `tcp`, 17 a `udp`) tramite la funzione `getprotobynumber()`, mentre alla righe 17 e 21 si usa la funzione `inet_ntop` per convertire un indirizzo IP memorizzato come `int` nella sua rappresentazione dotted-decimal.

I codici 5.18, 5.19 e 5.20 mostrano la funzione `receive_and_dispatch_appmgr_msg()` che riceve e gestisce messaggi JSON provenienti dalla Control Entity, effettuandone il parsing e costruendo il comando netlink corrispondente. Questa funzione, internamente, usa la funzione `netlink_send()` che invia uno dei comandi definiti sul socket netlink dato il tipo

Codice 5.15 Funzione `receive_and_dispatch_netlink_msg()` per la ricezione di messaggi netlink dal socket.

```
1 void receive_and_dispatch_netlink_msg() {
2   struct genl_msg ans;
3   if (handle_netlink_msg(&ans) < 0) return;
4
5   parse_nl_attrs(&ans);
6
7   if (ans.g.cmd == UPMT_APPMON_C_NEW_CONN) {
8     newconn_handler(nl_attr[UPMT_APPMON_A_NEW_CONN]);
9   }
10 }
```

di messaggio e la struttura dati da utilizzare come payload.

5.5 Interventi sul modulo upmt

Durante la scrittura del modulo xt_UPMT, di upmt-appmon e della successiva integrazione dei componenti con la Control Entity, sono state apportate alcune modifiche al modulo upmt, sia per integrarlo con il modulo xt_UPMT, sia per correggere alcuni problemi riscontrati.

5.6 Dipendenze tra moduli

Poiché il modulo xt_UPMT deve inserire regole nella PAFT al rilevamento di una nuova connessione, è stato necessario esportare, nel modulo upmt, la funzione `paft_insert_by_tid()`:

```
1 EXPORT_SYMBOL(paft_insert_by_tid);
```

Inoltre, nella funzione `upmt_prefilter()` del modulo xt_UPMT, responsabile delle eccezioni per applicazione e traffico locale, è stato necessario esportare, nel modulo upmt, la variabile globale upmt_dev, che denota il descrittore dell'interfaccia virtuale upmt0:

```
1 EXPORT_SYMBOL(upmt_dev);
```

L'esportazione di tali simboli nel modulo upmt, e la loro dichiarazione come `extern` nel modulo xt_UPMT, comporta la nascita di una relazione di dipendenza tra i due moduli, in cui xt_UPMT dipende da upmt. Tali moduli, dunque, dovranno essere compilati come descritto nella sottosezione 3.2.1.

5.6.1 Riscrittura meccanismo di lock

La versione originaria del modulo upmt utilizzava una serie di spinlock per proteggere le strutture dati, in particolare la PAFT, la tabella dei TSA e delle interfacce, da modifiche concorrenti. Le invocazioni di tali spinlock erano spesso annidate, ad esempio, nel caso di aggiunta di una regola nella PAFT:

Codice 5.16 Funzione `newconn_handler()` per la gestione di notifiche di nuove connessioni.

```
1  void newconn_handler(struct nlattr *na) {
2    void *payload = GENLMSG_NLA_DATA(na);
3    struct upmt_new_conn_data *nd = (struct upmt_new_conn_data*) payload;
4
5    cJSON *root, *key;
6    root = cJSON_CreateObject();
7    key = cJSON_CreateObject();
8    cJSON_AddItemToObject(root, UPMT_CONN_NOTIF_JSON_KEY, key);
9
10   struct protoent *proto = getprotobynumber(nd->key.proto);
11   cJSON_AddStringToObject(key, UPMT_CONN_NOTIF_JSON_KEY_PROTO, proto->p_name);
12
13   char ipbuf[INET_ADDRSTRLEN];
14   struct in_addr ip;
15
16   ip.s_addr = nd->key.daddr;
17   inet_ntop(AF_INET, &ip, ipbuf, INET_ADDRSTRLEN);
18   cJSON_AddStringToObject(key, UPMT_CONN_NOTIF_JSON_KEY_DADDR, ipbuf);
19
20   ip.s_addr = nd->key.saddr;
21   inet_ntop(AF_INET, &ip, ipbuf, INET_ADDRSTRLEN);
22   cJSON_AddStringToObject(key, UPMT_CONN_NOTIF_JSON_KEY_SADDR, ipbuf);
23
24   cJSON_AddNumberToObject(key, UPMT_CONN_NOTIF_JSON_KEY_DPORT, nd->key.dport);
25   cJSON_AddNumberToObject(key, UPMT_CONN_NOTIF_JSON_KEY_SPORT, nd->key.sport);
26
27   cJSON_AddStringToObject(root, UPMT_CONN_NOTIF_JSON_APPNAME, nd->appname);
28
29   cJSON_AddNumberToObject(root, UPMT_CONN_NOTIF_JSON_TID, nd->tid);
30
31   char *jsonmsg = cJSON_PrintUnformatted(root);
32   if (sendto(tomgr_sd, jsonmsg, strlen(jsonmsg), 0, (struct sockaddr *) &tomgr_addr, sizeof(
         tomgr_addr)) < 0)
33     perror("newconn_handler: sendto");
34 }
```

Codice 5.17 Funzione `netlink_send()` per l'invio di un messaggio sul socket netlink.

```
1  int netlink_send(int msgtype, void *msg) {
2    struct nlattr *na;
3
4    req.n.nlmsg_len   = NLMSG_LENGTH(GENL_HDRLEN);
5    req.n.nlmsg_type  = upmt_family_id;
6    req.n.nlmsg_flags   = NLM_F_REQUEST;
7    req.n.nlmsg_seq   = ++seqno;
8    req.n.nlmsg_pid   = getpid();
9    switch (msgtype) {
10     case MSG_TYPE_APP:
11       req.g.cmd = UPMT_APPMON_C_APP;
12       na = (struct nlattr *) GENLMSG_DATA(&req);
13       set_nl_attr(na, UPMT_APPMON_A_APP, msg, sizeof(struct upmt_app_msg));
14       break;
15     case MSG_TYPE_NO_UPMT:
16       req.g.cmd = UPMT_APPMON_C_NO_UPMT;
17       na = (struct nlattr *) GENLMSG_DATA(&req);
18       set_nl_attr(na, UPMT_APPMON_A_NO_UPMT, msg, sizeof(struct upmt_no_upmt_msg));
19       break;
20     case MSG_TYPE_APP_FLOW:
21       req.g.cmd = UPMT_APPMON_C_APP_FLOW;
22       na = (struct nlattr *) GENLMSG_DATA(&req);
23       set_nl_attr(na, UPMT_APPMON_A_APP_FLOW, msg, sizeof(struct upmt_app_flow_msg));
24       break;
25   }
26   req.n.nlmsg_len += NLMSG_ALIGN(na->nla_len);
27
28   if (sendto_fd(nl_sd, (char *) &req, req.n.nlmsg_len) < 0) return -1;
29
30   return 0;
31 }
```

Codice 5.18 Funzione `receive_and_dispatch_appmgr_msg()` per la ricezione e la gestione di messaggi inviati dalla Control Entity (1/3).

```
1  void receive_and_dispatch_appmgr_msg() {
2    char recmsg[UPMT_MSG_WIRELEN];
3    int msgtype;
4    cJSON *elem;
5
6    memset(&recmsg, 0, UPMT_MSG_WIRELEN);
7    recv(frommgr_sd, &recmsg, UPMT_MSG_WIRELEN, 0);
8
9    cJSON *parsedmsg = cJSON_Parse(recmsg);
10   if (parsedmsg == NULL) goto malformed;
11
12   elem = cJSON_GetObjectItem(parsedmsg, UPMT_JSON_DEBUG);
13   if (elem != NULL) goto debug;
14
15   elem = cJSON_GetObjectItem(parsedmsg, UPMT_APP_MSG_JSON_MSGTYPE);
16   if (elem == NULL) goto malformed;
17   msgtype = elem->valueint;
18
19   switch (msgtype) {
20     case MSG_TYPE_APP: {
21       struct upmt_app_msg msg;
22       memset(&msg, 0, sizeof(struct upmt_app_msg));
23
24       elem = cJSON_GetObjectItem(parsedmsg, UPMT_APP_MSG_JSON_COMMAND);
25       if (elem == NULL) goto malformed;
26       msg.command = elem->valueint;
27
28       switch (msg.command) {
29         case CMD_ADD:
30           elem = cJSON_GetObjectItem(parsedmsg, UPMT_APP_MSG_JSON_APPNAME);
31           if (elem == NULL) goto malformed;
32           strncpy(msg.appname, elem->valuestring, MAX_APPNAME_LENGTH);
33
34           elem = cJSON_GetObjectItem(parsedmsg, UPMT_APP_MSG_JSON_TID);
35           if (elem == NULL) goto malformed;
36           msg.tid = elem->valueint;
37           break;
38
39         case CMD_RM:
40           elem = cJSON_GetObjectItem(parsedmsg, UPMT_APP_MSG_JSON_APPNAME);
41           if (elem == NULL) goto malformed;
42           strncpy(msg.appname, elem->valuestring, MAX_APPNAME_LENGTH);
43           break;
44
45         case CMD_FLUSH_LIST:
46           break;
47
48         case CMD_SET_DEFAULT_TID:
49           elem = cJSON_GetObjectItem(parsedmsg, UPMT_APP_MSG_JSON_TID);
50           if (elem == NULL) goto malformed;
```

Codice 5.19 Funzione `receive_and_dispatch_appmgr_msg()` per la ricezione e la gestione di messaggi inviati dalla Control Entity (2/3).

```
1           msg.tid = elem->valueint;
2           break;
3       }
4
5       if (netlink_send(msgtype, &msg)) goto netlinkerr;
6     }
7     break;
8
9     case MSG_TYPE_NO_UPMT: {
10      struct upmt_no_upmt_msg msg;
11      memset(&msg, 0, sizeof(struct upmt_no_upmt_msg));
12
13      elem = cJSON_GetObjectItem(parsedmsg, UPMT_APP_MSG_JSON_COMMAND);
14      if (elem == NULL) goto malformed;
15      msg.command = elem->valueint;
16
17      if (msg.command == CMD_ADD || msg.command == CMD_RM) {
18        elem = cJSON_GetObjectItem(parsedmsg, UPMT_APP_MSG_JSON_APPNAME);
19        if (elem == NULL) goto malformed;
20        strncpy(msg.appname, elem->valuestring, MAX_APPNAME_LENGTH);
21      }
22      else if (msg.command == CMD_FLUSH_LIST) {
23        // no extra fields required
24      }
25      else goto malformed;
26
27      if (netlink_send(msgtype, &msg)) goto netlinkerr;
28    }
29    break;
30
31    case MSG_TYPE_APP_FLOW: {
32      struct upmt_app_flow_msg msg;
33      memset(&msg, 0, sizeof(struct upmt_app_flow_msg));
34
35      elem = cJSON_GetObjectItem(parsedmsg, UPMT_APP_MSG_JSON_COMMAND);
36      if (elem == NULL) goto malformed;
37      msg.command = elem->valueint;
38
39      if (msg.command == CMD_ADD || msg.command == CMD_RM) {
40        elem = cJSON_GetObjectItem(parsedmsg, UPMT_APP_MSG_JSON_DADDR);
41        if (elem == NULL) goto malformed;
42        struct in_addr daddr;
43        inet_pton(AF_INET, elem->valuestring, &daddr);
44        printf("MSG_TYPE_APP_FLOW: %s --> %d\n", elem->valuestring, daddr.s_addr);
45        msg.daddr = daddr.s_addr;
46
47        elem = cJSON_GetObjectItem(parsedmsg, UPMT_APP_MSG_JSON_TID);
48        if (elem == NULL) goto malformed;
49        msg.tid = elem->valueint;
50      }
```

Codice 5.20 Funzione `receive_and_dispatch_appmgr_msg()` per la ricezione e la gestione di messaggi inviati dalla Control Entity (3/3).

```
      else if (msg.command == CMD_FLUSH_LIST) {
        // no extra fields required
      }
      else goto malformed;

      if (netlink_send(msgtype, &msg)) goto netlinkerr;
    }
    break;

    default:
      goto malformed;
  }
  return;

debug:
  netlink_send_int(UPMT_APPMON_C_DUMP_LIST, UPMT_APPMON_A_DUMP_LIST_ID, elem->valueint);
  return;

malformed:
  printf("appmgr message [%s] malformed\n", recmsg);
  return;

netlinkerr:
  printf("error while sending msg to netlink\n");
  return;
}
```

Codice 5.21 Funzione `set_nl_attr()` per impostare il valore di un attributo in un messaggio generic netlink.

```
void set_nl_attr(struct nlattr *na, const unsigned int type, const void *data, const unsigned
      int len) {
  int length = len + 2;
  na->nla_type = type;
  na->nla_len = length + NLA_HDRLEN; //message length
  memcpy(GENLMSG_NLA_DATA(na), data, length);
}
```

Codice 5.22 Funzione `parse_nl_attrs()` per estrarre i valori di tutti gli attributi da un messaggio generic netlink.

```
1 void parse_nl_attrs(struct genl_msg *ans) {
2   reset_nl_attrs();
3
4   unsigned int n_attrs = 0;
5   struct nlattr *na;
6   unsigned int data_len = GENLMSG_DATALEN(&ans->n);
7
8   na = (struct nlattr *) GENLMSG_DATA(ans);
9   nl_attr[na->nla_type] = na;
10  n_attrs++;
11  data_len = data_len - NLA_ALIGN(na->nla_len);
12
13  while(data_len > 0){
14    na = (struct nlattr *) GENLMSG_NLA_NEXT(na);
15    nl_attr[na->nla_type] = na;
16    n_attrs++;
17    data_len = data_len - NLA_ALIGN(na->nla_len);
18  }
19  if(n_attrs > UPMT_APPMON_A_MAX) printf("parse_nl_attrs - too many attributes");
20 }
```

Codice 5.23 Macro per la manipolazione di messaggi generic netlink.

```
1 #define GENLMSG_DATA(glh)       ((void *)(NLMSG_DATA(glh) + GENL_HDRLEN))
2 #define GENLMSG_DATALEN(glh)    (NLMSG_PAYLOAD(glh, 0) - GENL_HDRLEN)
3 #define GENLMSG_NLA_NEXT(na)    (((void *)(na)) + NLA_ALIGN(na->nla_len))
4 #define GENLMSG_NLA_DATA(na)    ((void *)((char*)(na) + NLA_HDRLEN))
5 #define GENLMSG_NLA_DATALEN(na)   (na->nla_len - NLA_HDRLEN - 1)
```

```
spin_lock(&paft_lock);
spin_lock(&tunt_lock);
/* ... add rule ... */
spin_unlock(&tunt_lock);
spin_unlock(&paft_lock);
```

Tale approccio, poco elegante, è esteso a numerosi altri casi nel modulo, con fino a quattro livelli di annidamento. Esso ha mostrato problemi di deadlock, particolarmente nel caso di handover di centinaia di socket contemporaneamente. Si è pertanto scelto di riscrivere il sistema di locking del modulo, adottando un unico lock implementato come un *rwlock* (reader-writer lock). Questo è una primitiva di sincronizzazione simile a uno spinlock, in quanto l'attesa di entrare in sezione critica è implementata come busy-waiting, ma che favorisce le letture rispetto alle scritture. Sono permesse, infatti, letture concorrenti da parte di più processi, ma i processi che effettuano scritture devono acquisire un lock esclusivo.

Il seguente codice, tratto dal file `upmt_locks.c` del modulo `upmt`, mostra la dichiarazione del rwlock `bul_mutex` (dove `bul` sta per Big UPMT Lock) e le funzioni usate per acquisire e rilasciare il lock, sia in lettura che in scrittura:

```
DEFINE_RWLOCK(bul_mutex);

inline void bul_read_lock() {
  read_lock(&bul_mutex);
}

inline void bul_read_unlock() {
  read_unlock(&bul_mutex);
}

inline void bul_write_lock() {
  write_lock(&bul_mutex);
}

inline void bul_write_unlock() {
  write_unlock(&bul_mutex);
}
```

La sostituzione dei vecchi spinlock con l'rwlock è stata piuttosto agevole, l'unica operazione che ha richiesto attenzione è stata la determinazione, per ogni sezione critica, se le funzioni chiamate al suo interno effettuassero scritture, così da richiedere il lock in scrittura, o solo letture, caso in cui è sufficiente il lock in lettura.

5.7 Implementazione del sistema di build

Con l'aggiunta del modulo `xt_UPMT`, la sua dipendenza da quest'ultimo e da Xtables-addons, e l'aggiunta di `upmt-appmon`, si è reso necessario estendere il sistema di makefile esistente, allo scopo di compilare correttamente questi nuovi moduli. Questi componenti sono stati organizzati in directory, ciascuna dotata di un proprio makefile. Nella loro directory padre, a sua volta, è stato creato un makefile che invoca ricorsivamente i makefile delle sottodirectory.

L'albero di directory ottenuto è il seguente:

```
1  upmt/
2    java/
3      upmt/
4        src/
5          Makefile
6    kernel/
7      upmt/
8        Makefile
9      upmt-conntracker/
10       Makefile
11     Makefile
12   upmt-appmon/
13     Makefile
14   upmtconf/
15     Makefile
16   Makefile
```

Il top-level makefile (riga 16) invoca ricorsivamente i makefile delle sottodirectory. Il makefile della directory `kernel/` (riga 11), a sua volta, invoca i makefile delle sue sottodirectory `upmt/` (riga 8) e `upmt-conntracker/` (riga 10) per compilare i moduli, Quest'ultimo makefile, inoltre, compila i plugin userspace per iptables relativi al target `UPMT`. Infine, vengono invocati i makefile in `upmt-appmon/` (riga 13) e in `upmtconf/` (riga 15).

Il top-level makefile dispone di un target `install`, utilizzabile per installare, ricorsivamente, tutti i componenti. I moduli vengono installati in `/lib/modules/VERSION/extra/`, dove `VERSION` è la versione del kernel in uso. Il plugin userspace per iptables viene installato nella directory appropriata, che dipende da come è stato compilato itpables e quindi dalla distribuzione Linux, determinata invocando `pkg-config`:

```
1  plugins_install:
2    install -m 0755 libxt_UPMT.so ‘pkg-config --variable=xtlibdir xtables‘
```

Infine, `upmt-appmon` e `upmtconf/` vengono installati in `/usr/bin/`.

Grazie alla stesura di questi makefile, è ora possibile compilare e installare i componenti di base di UPMT (modulo `upmt` e `xt_UPMT`, `upmt-appmon`, `upmtconf` e la libreria dinamica di `upmtconf` usata via JNI dalla Control Entity) semplicemente tramite i comandi:

```
1  $ make
2  # make install
```

6 Porting su piattaforma Android

> the definition of open: "mkdir android ; cd android ; repo init -u git://android.git.kernel.org/platform/manifest.git ; repo sync ; make"[1]
>
> *Andy Rubin*
> Vicepresident of Engineering presso Google e cofondatore di Android Inc.

La fase successiva del lavoro di tesi è stato il porting dei componenti realizzati su piattaforma Android. Il dispositivo a disposizione per lo sviluppo è un Google Nexus One. La macchina utilizzata per lo sviluppo è una macchina virtuale x86 con Ubuntu 10.10 i686.

6.1 Perché Android

La piattaforma mobile scelta per il porting di UPMT è stata Android. I motivi che hanno portato a questa scelta sono stati:

- Basata sul kernel Linux, seppure con qualche modifica
- Applicazioni scritte in Java
- Grande diffusione grazie a numerosi modelli di dispositivi
- Open-source (licenza GPLv2 per il kernel, Apache per la piattaforma)

Una piattaforma simile è MeeGo [16], sponsorizzata da Intel e Nokia. Questa, pur essendo basata sul kernel Linux, è ancora molto poco diffusa, esistendo solo pochi terminali, il più famoso dei quali è il Nokia N900. Inoltre, le applicazioni sono scritte in C++ utilizzando il framework Qt, il che avrebbe comportato la riscrittura completa della Control Entity.

[1] Tweet originale: http://twitter.com/#!/arubin/status/27808662429

Le altre piattaforme mobili quali iOS di Apple e Windows Phone 7 sono state immediatamente scartate per la loro chiusura, che impedisce ogni modifica non autorizzata, e per il fatto che non utilizzano il kernel Linux. Symbian (nelle sue varie versioni), pur essendo rilasciato con la licenza open source Eclipse Public License, utilizza un nano-kernel custom chiamato EKA2, e le applicazioni sono scritte in C++, pertanto è stato parimenti scartato.

6.2 Caratteristiche della piattaforma

Android è una piattaforma open source per dispositivi mobili inizialmente sviluppato da Android Inc., in seguito acquistata da Google, e attualmente mantenuto e sviluppato da Android Open Source Project (AOSP). Nel quarto trimestre del 2010, le vendite di dispositivi Android, pari a $32,9$ milioni di unità, hanno superato le vendite dei dispositivi Symbian, pari a 31 milioni di unità [17].

La piattaforma è basata sul kernel Linux ma non si può considerare una distribuzione Linux convenzionale: non è dotata di un X server né supporta le librerie standard GNU quali la GNU C Library (`glibc`). Android, infatti, fornisce una propria versione della C Library chiamata `bionic`, che implementa un sottoinsieme della GNU C Library.

Le applicazioni sono scritte in linguaggio Java utilizzando la ricca API fornita dalla piattaforma, e vengono eseguite tramite la Dalvik virtual machine, una Java virtual machine adattata per l'uso su dispositivi mobili. Tramite il tool `dx`, è possibile convertire i file `.class` prodotti da un tradizionale compilatore Java nel formato bytecode `.dex` utilizzato da Dalvik. Questa procedura assembla più classi nello stesso file .dex, includendo le stringhe e altre costanti duplicate una sola volta per risparmiare spazio.

L'SDK include gli strumenti di sviluppo, le librerie, un emulatore del dispositivo basato sul software di virtualizzazione QEMU, la documentazione, alcuni progetti di esempio, tutorial e altro. Al contrario di Windows Phone e iOS, è installabile su qualsiasi computer x86 compatibile che usi come sistema operativo Windows, Mac OS X o Linux. L'IDE ufficialmente supportato per lo sviluppo di applicazioni per Android è Eclipse, per il quale è fornito un plug-in che si interfaccia con l'emulatore.

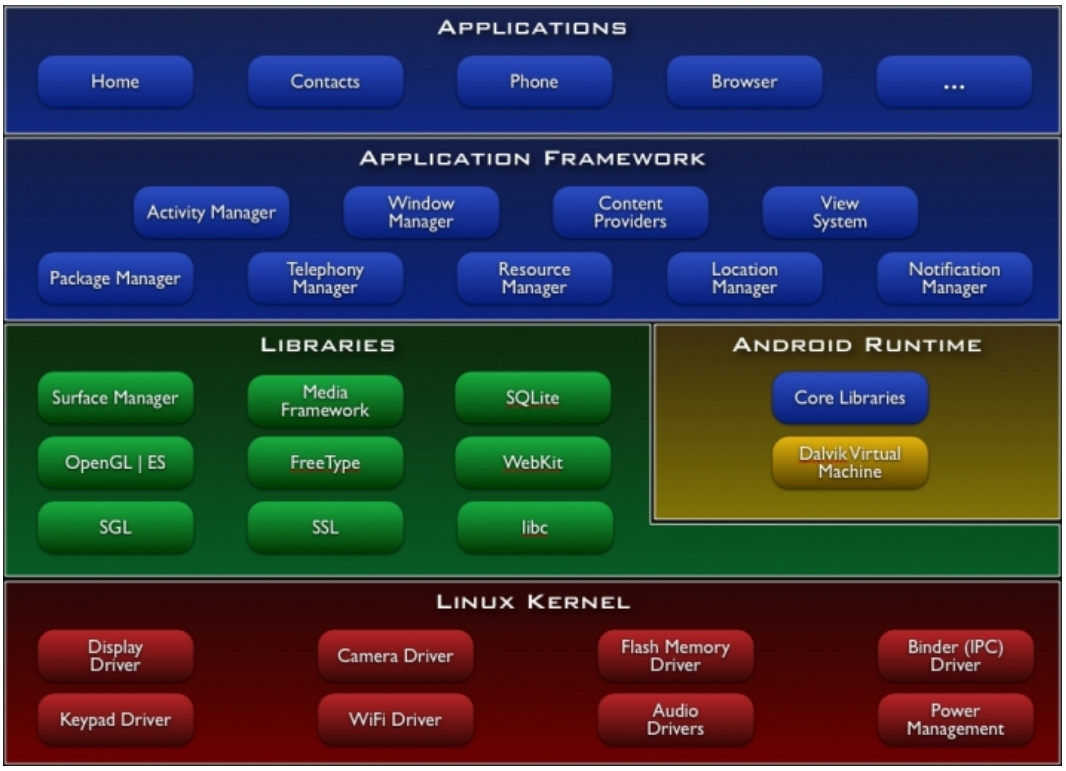
La figura 6.1 mostra i componenti principali che compongono la piattaforma e la loro stratificazione.

Il riquadro "Applications" racchiude le applicazioni preinstallate, come il programma di messaggistica, il client email, il calendario, il visualizzatore di mappe, il browser, la rubrica ecc. Tutte le applicazioni sono scritte in linguaggio Java.

Il riquadro "Application Framework" riassume tutte le facility di Android per la gestione del ciclo di vita delle applicazioni (activity manager, window manager, package manager), per disegnare e gestire l'interfaccia grafica (view system), per l'accesso e lo scambio di dati tra applicazioni (resource manager, content providers), per la gestione della telefonia (telephony manager), per la gestione della posizione (location manager) e delle notifiche (notification manager).

Nel riquadro "Libraries" sono elencate le librerie di basso livello implementate in C/C++ messe a disposizione delle applicazioni tramite l'application framework. Citiamo la C library (libc), le librerie multimediali per immagini e flussi audio e video (media framework), le librerie relative alla gestione dello schermo touch (Surface Manager), il motore di rendering

Figura 6.1 Architettura della piattaforma Android



per pagine web WebKit, librerie per la gestione di grafica 2D (SGL) e 3D (OpenGL), per il rendering dei font (FreeType), il database relazionale SQLite, il framework per la crittografia OpenSSL.

Il riquadro "Android runtime" comprende gran parte delle librerie standard messe a disposizione da Java (Core libraries). In Android, ogni applicazione viene eseguita in un processo, con una sua istanza privata della Dalvik Virtual Machine. Quest'ultima è un'implementazione della Java Virtual Machine basata su registri[2], ottimizzata in modo tale che un dispositivo mobile possa eseguire più VM contemporaneamente in modo efficiente. Dalvik esegue codice bytecode nel formato Dalvik Executable (.dex), una versione del bytecode Java ottimizzata per avere ridotte richieste di memoria. Dalvik si affida al kernel Linux sottostante per operazioni primitive quali la gestione di thread e processi e la gestione della memoria a basso livello.

6.2.1 Android e codice nativo

Android si basa sul kernel Linux nella versione 2.6 per i servizi di base come la sicurezza, la gestione della memoria e dei processi, lo stack di rete e il driver model. Google ha tuttavia

[2]La Java Virtual Machine classica è invece stack-oriented.

apportato diverse patch per migliorare l'efficienza energetica e per implementare alcuni requisiti di sicurezza riguardanti l'isolamento (*sandboxing*) di applicazioni.

I programmi e le librerie scritte in C devono essere compilate per produrre codice nativo ARM. L'Android Native Development Kit (NDK) fornisce una serie di API per l'invocazione di tali librerie da parte delle applicazioni Java. La compilazione in codice ARM può avvenire utilizzando una toolchain che può essere preparata tramite il NDK stesso. L'esecuzione di codice nativo, tuttavia, è complicata dal fatto che Android usa una libreria C non standard chiamata Bionic [18].

Questa libreria è stata sviluppata principalmente per questioni di:

- Licenza: non si desiderano componenti sotto GPL o LGPL in user space
- Dimensione: `glibc` è molto grande, mentre `uClibC`, sebbene molto più compatta, è rilasciata sotto licenza LGPL
- Velocità: poichè l'ambiente di esecuzione è embedded, è necessario che la libreria sia veloce. In pratica, questo si traduce nella rimozione di alcune componenti della libc standard.

Alcune importanti differenze con la `glibc` sono:

- Non sono supportate le eccezioni C++, né viene fornita una Standard Template Library (STL)
- L'implementazione di `libpthread` è completamente scritta da Google specificatamente per Android, quindi non tutte le funzioni sono disponibili
- Non viene offerto supporto ai caratteri multi-byte
- Esiste una regione di memoria condivisa per le proprietà di configurazione. Ad esempio, le impostazioni DNS sono memorizzate in tale area e non nel file usuale `/etc/resolv.conf`
- Alcune funzioni, quali `getprotobyname()` o `getprotobynumber()`, non sono implementate e attualmente stampano messaggi del tipo `FIX ME! implement getprotobyname() __FILE__:__LINE__`

6.2.2 Applicazioni e processi

I componenti di un'applicazione possono essere di quattro tipi:

Activity Presenta un'interfaccia grafica per eseguire un determinato compito. Un'applicazione può essere composta da una o più activity: una di esse è impostata come activity da caricare al lancio dell'applicazione.

Service Privo di interfaccia grafica, usato per eseguire codice in background.

Broadcast receiver Riceve e reagisce ad annunci in broadcast (broadcast announcements), che possono essere generati dal sistema (ad es. una chiamata in arrivo, batteria scarsa, connessione a una rete ecc.) o dalle applicazioni.

Content provider Utilizzati per mettere a disposizione di altre applicazione un insieme di dati.

Un principio cardine delle applicazioni Android è l'interoperabilità: ogni applicazione mette a disposizione elementi e servizi che possono essere utilizzati o invocati da altre applicazioni. Quando un'applicazione fa riferimento a parti esterne, il sistema si fa carico di avviare la parte di applicazione necessaria. Di conseguenza, contrariamente alle applicazioni tradizionali, le applicazioni Android non hanno un entry-point unico (in termini di applicazioni C/C++/Java non dispongono di una funziona `main()`).

Quando un componente di un'applicazione viene invocato per la prima volta, Android crea un processo Linux per l'applicazione, dotato di un unico thread di esecuzione. Di default, tutti i componenti di un'applicazione vengono eseguiti in tale processo e thread. Tuttavia, è possibile eseguire componenti in processi separati, e avviare thread addizionali per ciascuno di questi processi. Tali comportamenti sono concettualmente simili, rispettivamente, a creare un nuovo processo di un'applicazione invocando la chiamata di sistema `fork()` o a creare un thread tramite le librerie `libpthread`.

Finora si è parlato di processi Linux e di applicazioni Java. Il layer intermedio tra i due è la Dalvik Virtual Machine, di cui esiste un'istanza per ogni processo. Anche se la stessa applicazione ha componenti in esecuzione su più processi, dunque, esisterà un'istanza di Dalvik per ognuno di essi. La comunicazione tra VM, dunque, richiede un meccanismo di IPC, realizzato dal componente `Binder`. Anche se l'esistenza di queste VM multiple può sembrare uno spreco, Dalvik è stata progettata espressamente per questa circostanza. Ogni VM, infatti, è ottenuta tramite fork di una VM particolare, chiamata `zygote`, e fornita dall'omonimo processo. Al boot del dispositivo, infatti, il processo init esegue il programma `/system/bin/app_process`, e attribuisce al processo risultante il nome `zygote`, come testimoniato dal file `init.rc`:

```
1 service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
```

Questo processo costituisce l'ambiente di runtime di Dalvik, ed esegue il memory mapping di tutte le librerie Android comuni. Le VM figlie sfruttano il ben noto meccanismo di copy-on-write per condividere pagine di memoria in sola lettura con il processo `zygote`, evitando così di dover caricare copie multiple delle stesse librerie.

Quando l'ultimo componente di un'applicazione aperta termina il proprio ciclo di vita, i processi corrispondenti all'applicazione non vengono terminati, ma mantenuti in esecuzione. In tal modo si evita l'overhead introdotto dalla distruzione e dalla futura creazione di un nuovo processo. Le istanze della VM in esecuzione su di essi, però, vengono terminate per risparmiare memoria, in quanto l'allocazione di una nuova istanza di VM è, come visto, un'operazione poco costosa. Tali processi possono, comunque, essere terminati a discrezione del sistema operativo in caso di scarsità di risorse, in particolare di memoria.

6.3 Compilazione di programmi nativi

Il processo di compilazione di un programma su un'architettura allo scopo di ottenere codice eseguibile su un'altra architettura è chiamato *cross-compilazione*. Il compilatore utilizzato

per questo scopo è pertanto chiamato *cross-compilatore*. I manuali di `gcc` [8] definiscono tre tipi di piattaforma che possono intervenire nella cross-compilazione:

`build` La piattaforma su cui viene eseguita la compilazione

`host` La piattaforma per cui si sta compilando

`target` La piattaforma per cui `gcc` produrrà codice (di solito si applica solo alla compilazione di compilatori)

Usare, come nel nostro caso, un cross-compilatore ARM su una macchina x86, dunque, comporta che `build` sia la macchina x86, `host` sia la macchina ARM e `target` non sia specificata, essendo non rilevante in questo caso. Questa configurazione, quindi, sfrutta un cross-compilatore per generare codice nativo per un'architettura differente: tale configurazione viene definita host-x-host, crossed native o cross-built native.

Oltre al compilatore vero e proprio, sono anche necessari un assemblatore e un linker, oltre a diversi altri tool utilizzati nel processo, collettivamente denominati `binutils`. Tutti questi programmi sono raggruppati in una `toolchain`, e hanno il loro nome preceduto da un prefisso che indica l'architettura host, ad esempio `arm-none-linux-gnueabi-gcc`.

Nel caso di Android, non è tanto problematico compilare programmi e librerie native, quanto eseguirne il linking alla libreria `bionic` che, come visto, non offre completa compatibilità con la usuale `glibc`. Una possibile soluzione consiste nel linkare staticamente la `glibc` ai programmi, ma ciò aggiunge circa 2 MB alle dimensioni degli eseguibili ottenuti.

Per l'architettura ARM esistono numerose toolchain già pronte e di facile installazione, la più famosa delle quali è prodotta da CodeSourcery [19]. Questa, tuttavia, è una toolchain general-purpose per dispositivi ARM e utilizza la `glibc` in fase di linking. Nel dicembre 2010, Google ha rilasciato l'Android NDK versione 5, che include una toolchain ottimizzata per l'uso in Android e che utilizza `bionic` in fase di linking. Questa toolchain però, come si vedrà più avanti, non è ancora matura ed è stato necessario apportare, ove possibile, alcuni adattamenti.

6.4 Il dispositivo utilizzato: Google Nexus One

Nexus One è il primo telefono cellulare presentato con il marchio Google. È prodotto da HTC ed è stato presentato il 5 gennaio 2010. Nexus One ha rappresentato il developer phone per eccellenza fino al 6 dicembre 2010, data di presentazione del successore Nexus S, prodotto da Samsung.

Il telefono supporta le reti con tecnologia HSDPA/HSUPA, è dotato di un processore Qualcomm Snapdragon 1GHz, RAM pari a 512MB, ROM pari a 512MB, memoria interna da 512MB, memoria esterna Micro SD espandibile fino a 32GB, fotocamera da 5 megapixel con flash, schermo AMOLED da 3,7 pollici touch-screen capacitivo multi-touch, A-GPS con bussola digitale, Bluetooth 2.1, connessione Wi-Fi b/g/n, accelerometro, sensore di luce ambientale e di prossimità.

La memoria interna è suddivisa in diverse partizioni:

```
1 # cat /proc/mtd
2 dev:    size   erasesize  name
```

Figura 6.2 Google Nexus One.

```
3 mtd0: 000e0000 00020000 "misc"
4 mtd1: 00400000 00020000 "recovery"
5 mtd2: 00380000 00020000 "boot"
6 mtd3: 09100000 00020000 "system"
7 mtd4: 05f00000 00020000 "cache"
8 mtd5: 0c440000 00020000 "userdata"
```

La partizione `boot` contiene il kernel e il relativo ramdisk; la partizione `system` contiene l'albero delle directory contenenti tutti i file e programmi di sistema, montato in `/system`; la partizione `userdata` contiene i dati delle applicazioni installate, montata in `/data`; la partizione `recovery` contiene un tool utilizzabile da una speciale modalità, chiamata appunto recovery, per eseguire il flashing di immagini sulle partizioni, applicare aggiornamenti e altre operazioni di sistema.

Le partizioni `system`, `cache`, `userdata` e `misc` contentono file system di tipo `yaffs2`. Le partizioni `boot` e `recovery`, invece, sono organizzate secondo un formato custom di Android che prevede un header di 2 KB, seguito dall'immagine compressa con `gzip` del kernel, seguita dal ramdisk e da un second stage loader (poco diffuso). Questa struttura è delineata nel file `mkbootimg.h`:

```
1 +-----------------+
2 | boot header     | 1 page
3 +-----------------+
4 | kernel          | n pages
5 +-----------------+
6 | ramdisk         | m pages
7 +-----------------+
8 | second stage    | o pages
9 +-----------------+
10
11 n = (kernel_size + page_size - 1) / page_size
12 m = (ramdisk_size + page_size - 1) / page_size
13 o = (second_size + page_size - 1) / page_size
14
15 0. all entities are page_size aligned in flash
16 1. kernel and ramdisk are required (size != 0)
17 2. second is optional (second_size == 0 -> no second)
```

Il ramdisk è un piccolo file system contenente i file principali di sistema necessari alla sua inizializzazione. Tra gli altri, include il programma `init` e il suo file di configurazione `init.rc`. I seguenti file sono tipicamente contenuti in un ramdisk:

```
1 ./init.trout.rc
2 ./default.prop
3 ./proc
4 ./dev
5 ./init.rc
6 ./init
7 ./sys
8 ./init.goldfish.rc
9 ./sbin
10 ./sbin/adbd
11 ./system
12 ./data
```

All'accensione del telefono, viene invocato l'HBOOT (Hardware Bootloader), un equivalente embedded del BIOS dei PC, che rileva e rende disponibile la RAM, carica il bootloader primario nella RAM e inizializza i dispositivi hardware. In seguito, il bootloader cerca un kernel da avviare leggendo la partizione `boot`. Una volta copiato il kernel nella RAM, si salta alla prima istruzione, che avvia la decompressione del kernel e l'esecuzione della sequenza di inizializzazione del kernel. In seguito, si carica dal ramdisk il processo `init` che, in base al contenuto del file `init.rc`, avvia i servizi di sistema specificati.

Particolari combinazioni di tasti effettuate all'accensione portano il telefono in alcune modalità di servizio: tasto power + Volume Giù portano il telefono in modalita *HBOOT*, in cui è possibile flashare una RUU, ovvero una ROM rilasciata dal produttore, che ripristina il telefono alle condizioni di fabbrica; tasto power + Trackball premuta, invece, portano il telefono in modalità *fastboot*, in cui è possible flashare immagini sulle partizioni attraverso la connessione USB, servendosi del protocollo omonimo. Tale protocollo permette di flashare immagini non firmate: per tale motivo è spesso disabilitato nei telefoni in vendita tramite la disattivazione della connettività USB nel bootloader.

6.5 Sblocco del bootloader e rooting del telefono

Essendo il Nexus One un developer phone, Google ha implementato un semplice comando per sbloccare il bootloader e permettere di flashare immagini auto-prodotte. Tale comando è:

```
1 fastboot oem unlock
```

da eseguire con il telefono in modalità fastboot. Questa operazione è irreversibile e comporta l'automatico decadimento della garanzia. Su altri dispositivi dove tale comando non è disponibile, lo stesso effetto si ottiene tipicamente sfruttando vulnerabilità nel kernel, nel protocollo fastboot o nel protocollo di comunicazione USB.

In Android, a nessun processo tranne quelli di sistema è permesso di essere eseguito con privilegi di root. Eseguire il *rooting* del telefono significa rimuovere questa limitazione. In [20] è riportata la procedura seguita per eseguire il rooting del Nexus One a disposizione, la cui ROM ufficiale è Android 2.2.2 con numero di build FRG83G.

6.6 Compilazione e flashing di un kernel custom

Come su una distribuzione Linux per desktop, anche su Android è possibile sostituire il solo kernel Linux senza modificare il resto del sistema. Il kernel, infatti, risiede nella partizione `boot`, ed è sostituibile producendo un'immagine da flashare in tale partizione tramite fastboot.

Tuttavia, il kernel deve essere cross-compilato: a tale scopo, Google mette a disposizione una toolchain nei sorgenti di Android. In [21] è riportata la procedura per ottenerli e i pacchetti da installare sulla propria distribuzione per eseguire la compilazione. In breve, è necessario scaricare il tool `repo` sviluppato da Google per la gestione simultanea di più repository `git`. Tramite questo tool, si avvia il checkout del branch `froyo`, corrispondente ad Android 2.2, la versione attualmente installata sul Nexus One a disposizione. La directory `prebuilt/` contiene alcuni file precompilati, disponibili per varie architetture e SO, tra cui la toolchain utilizzata durante il processo di compilazione, in `prebuilt/linux-x86/toolchain/arm-eabi-4.4.0`,

e alcune immagini del kernel già compilate per l'emulatore. La directory external/ contiene, nelle sottodirectory corrispondenti, alcuni tool esterni alla piattaforma ma usati da Android, quali `iptables`, `ping`, `wpa_supplicant` ecc. Nel seguito, ci si riferirà alla directory in cui si è eseguito il checkout dei sorgenti di Android come `$ANDROID`.

I sorgenti del kernel non sono compresi nei sorgenti di Android e devono, pertanto, essere scaricati separatamente tramite i comandi seguenti:

```
1 git clone git://android.git.kernel.org/kernel/msm.git nexus_kernel
2 cd nexus_kernel
3 git checkout android-msm-2.6.32
```

È necessario eseguire il checkout del branch `msm` poiché il Nexus One, come molti altri telefoni Android assemblati da HTC, dispone della famiglia di chipset Qualcomm MSM per gestione della rete cellulare CDMA/GSM/UMTS. La versione del kernel 2.6.32 corrisponde a quella fornita con la ROM originale di Android 2.2: l'uso di questa versione permette di riutilizzare il file di configurazione `.config` del kernel originale. Nel seguito, ci si riferirà alla directory in cui si è eseguito il checkout di questo ramo come `$KERNEL`.

Una volta ottenuto il tree dei sorgenti del kernel, è necessario applicare la patch riportata nel codice 4.1 per aggiungere il TGID al socket buffer. Tale patch, fornita in formato unified diff, è applicabile tramite il comando `patch`.

A questo punto, è necessario configurare il kernel oppure, come è stato fatto in questo lavoro di tesi, copiare il file di configurazione direttamente dal Nexus One, quando era ancora installato il kernel originale. Ciò è stato fatto tramite i comandi, impartiti sulla macchina di sviluppo:

```
1 adb pull /proc/config.gz ./config_n1_2.2.2.gz
2 gunzip ./config_n1_2.2.2.gz
3 cp ./config_n1_2.2.2 $KERNEL/.config
```

Questa configurazione, tuttavia, ha il supporto alle tabelle di routing multiple, al policy routing e al packet mangling disabilitati. Per abilitarli è necessario aprire l'interfaccia di configurazione del kernel, non prima di aver definito alcune variabili d'ambiente per eseguire la cross-compilazione verso l'architettura ARM:

```
1 ARCH=arm CROSS_COMPILE=$ANDROID/prebuilt/linux-x86/toolchain/arm-eabi-4.4.0/bin/arm-eabi-
      make menuconfig
```

e abilitare le seguenti opzioni:

```
1  Symbol: IP_ADVANCED_ROUTER [=y]
2    Prompt: IP: advanced router
3      Defined at net/ipv4/Kconfig:17
4      Depends on: NET [=y] && INET [=y]
5      Location:
6        -> Networking support (NET [=y])
7          -> Networking options
8            -> TCP/IP networking (INET [=y])
9
10 Symbol: IP_MULTIPLE_TABLES [=y]
11   Prompt: IP: policy routing
12     Defined at net/ipv4/Kconfig:101
13     Depends on: NET [=y] && INET [=y] && IP_ADVANCED_ROUTER [=y]
```

```
14    Location:
15      -> Networking support (NET [=y])
16        -> Networking options
17          -> TCP/IP networking (INET [=y])
18    Selects: FIB_RULES [=y]
19
20
21 Symbol: IP_NF_MANGLE [=y]
22   Prompt: Packet mangling
23     Defined at net/ipv4/netfilter/Kconfig:287
24     Depends on: NET [=y] && INET [=y] && NETFILTER [=y] && IP_NF_IPTABLES [=y]
25     Location:
26       -> Networking support (NET [=y])
27         -> Networking options
28           -> Network packet filtering framework (Netfilter) (NETFILTER [=y])
29             -> IP: Netfilter Configuration
30               -> IP tables support (required for filtering/masq/NAT) (IP_NF_IPTABLES [=y])
```

Per compilare il kernel si usa il tradizionale comando make:

```
1 ARCH=arm CROSS_COMPILE=$ANDROID/prebuilt/linux-x86/toolchain/arm-eabi-4.4.0/bin/arm-eabi-
     make
```

La variabile `ARCH` indica che è l'architettura desiderata è ARM, pertanto il kernel utilizzerà, ove necessario, i sorgenti specifici per ARM (ad esempio, includerà i file assembler definiti in `include/asm-arm`). La variabile `CROSS_COMPILE` costituisce il prefisso da anteporre agli strumenti di compilazione, quali `gcc`, `ld`, `readelf`, `objcopy` ecc. I valori risultanti puntano ai corrispondenti strumenti forniti dalla toolchain scaricata precedentemente.

Il file di config predefinito prevede la compilazione come modulo del solo driver della scheda wireless, ogni altro componente è compilato staticamente. Pertanto, la compilazione del kernel produce i due seguenti file:

```
1 $KERNEL/arch/arm/boot/zImage
2 $KERNEL/drivers/net/wireless/bcm4329/bcm4329.ko
```

A partire dal file `zImage`, corrispondente all'immagine del kernel ottenuta, deve essere prodotta un'immagine da flashare utilizzando fastboot. Ciò è possibile utilizzando il comando `mkbootimg`, i cui sorgenti sono disponibili nel tree di Android. Questo può essere compilato come segue:

```
1 cd $ANDROID
2 . build/envsetup.sh
3 lunch full_passion-userdebug
4 make mkbootimg
```

Questo tool richiede quattro parametri: la command line da fornire al kernel, il percorso dell'immagine del kernel, il percorso dell'immagine del ramdisk e, obbligatorio per il Nexus One ma non in generale, un parametro `--base` che rappresenta l'indirizzo fisico a cui l'immagine del kernel deve essere posta. La command line del kernel e il ramdisk possono essere estratti dalla ROM ufficiale, reperibile in molti mirror offerti da modder quali MoDaCo[3]

[3]Si noti che questa ROM, pur provenendo da siti non ufficiali, è firmata digitalmente da Google. Infatti, è possibile installarla su telefoni completamente privi di ogni sblocco, senza far decadere la garanzia.

[22]. Tali ROM sono fornite sotto forma di file in formato ZIP. Scompattando tale archivio, si ottengono i seguenti file:

```
1 android-info.txt
2 boot.img
3 recovery.img
4 system.img
5 userdata.img
```

Il file che ci interessa è `boot.img`. Il sito [23] mette a disposizione uno script in Perl per l'estrazione di file dall'immagine di boot, chiamato `split_bootimg.pl`.

```
1 $ ./split_bootimg.pl boot.img
2 Page size: 2048 (0x00000800)
3 Kernel size: 2226200 (0x0021f818)
4 Ramdisk size: 164628 (0x00028314)
5 Second size: 0 (0x00000000)
6 Board name:
7 Command line: no_console_suspend=1 msmsdcc_sdioirq=1 wire.search_count=5
8 Writing boot.img-kernel ... complete.
9 Writing boot.img-ramdisk.gz ... complete.
```

La riga 7 riporta la command line passata al kernel compreso nell'immagine, la riga 9 indica il nome del file in cui è stato estratto il ramdisk.

La nostra immagine di boot personalizzata, dunque, può essere creata con il seguente comando:

```
1 cp $KERNEL/arch/arm/boot/zImage ~/android/
2 cd ~/android/
3 $ANDROID/out/host/linux-x86/bin/mkbootimg --cmdline 'no_console_suspend=1 msmsdcc_sdioirq=1
      wire.search_count=5' --kernel zImage --ramdisk boot.img-ramdisk.gz --base 0x20000000 -o
      boot-upmt.img
```

Questo produrrà il file immagine `boot-upmt.img`. Per provare questa immagine senza flasharla, fastboot mette a disposizione il seguente comando, da impartire con il telefono in modalità fastboot:

```
1 fastboot boot boot-upmt.img
```

In caso di problemi, è sufficiente riavviare il telefono per tornare al kernel precedente. Una volta accertato che il nuovo kernel funziona correttamente, è necessario flashare la nuova immagine:

```
1 fastboot flash boot boot-upmt.img
```

Una volta avviato il telefono, rimane da copiare il modulo per la scheda wireless:

```
1 adb push bcm4329.ko /data/local/tmp/
2 adb shell
3 # mount -o remount,rw /dev/block/mtdblock3 /system
4 # cat /data/local/tmp/bcm4329.ko > /system/lib/modules/bcm4329.ko
5 # rm /data/local/tmp/bcm4329.ko
6 # mount -o remount,ro /dev/block/mtdblock3 /system
```

Codice 6.1 Patch applicata ai sorgenti di busybox preparati dal team CyanogenMod.

```
diff --git a/Android.mk b/Android.mk
index c8cb674..3660241 100644
--- a/Android.mk
+++ b/Android.mk
@@ -17,7 +17,6 @@ KERNEL_MODULES_DIR?=/system/modules/lib/modules
 BUSYBOX_SRC_FILES = $(shell cat $(LOCAL_PATH)/busybox-$(BUSYBOX_CONFIG).sources) \
        libbb/android.c

-ifeq ($(strip $(CYANOGEN_BIONIC)),true)
     ifeq ($(TARGET_ARCH),arm)
       BUSYBOX_SRC_FILES += \
         android/libc/arch-arm/syscalls/adjtimex.S \
@@ -27,7 +26,6 @@ ifeq ($(strip $(CYANOGEN_BIONIC)),true)
         android/libc/arch-arm/syscalls/swapoff.S \
         android/libc/arch-arm/syscalls/sysinfo.S
     endif
-endif

 BUSYBOX_C_INCLUDES = \
        $(LOCAL_PATH)/include-$(BUSYBOX_CONFIG) \
```

6.7 Compilazione di iproute2 e iptables

La ROM stock manca di alcuni comandi, il più importanti dei quali è `ip` fornito dalla suite `iproute2`. Senza di esso, non sarebbe possibile creare e operare su tabelle di routing multiple, né di ottenere informazioni sulle interfacce di rete.

Questo tool è compreso in BusyBox [24], una suite di vari programmi comuni in Linux specificatamente destinata a sistemi embedded. La compilazione di `busybox` può essere agevolmente effettuata scaricando i sorgenti preparati dal team della CyanogenMod [25] per la loro ROM custom, una delle più famose e utilizzate. Questo ramo, infatti, contiene alcune modifiche volte ad assicurare la compatibilità con la libreria bionic, che viene linkata dinamicamente. Inoltre, viene fornito un apposito makefile che consente la compilazione tramite l'Android build system: è sufficiente, infatti, copiare la directory dei sorgenti in `$ANDROID/external/`. In tal modo, il file `system.img` prodotto alla fine della compilazione del tree di Android conterrà l'eseguibile `busybox` e i link simbolici per i comandi da esso forniti.

Il checkout dei sorgenti viene effettuato dal repository della CyanogenMod su github [26]:

```
cd $ANDROID/external/
git clone git://github.com/CyanogenMod/android_external_busybox.git busybox -b froyo
```

A tali sorgenti, è stata applicata la patch riportata nel codice 6.1 per permetterne la compilazione su una ROM stock. Busybox può quindi essere compilato tramite i comandi:

```
cd $ANDROID
. build/envsetup.sh
lunch full_passion-userdebug
make busybox
```

Un altro problema riscontrato con i programmi forniti dalla ROM stock è il mancato supporto al packet marking in iptables. In questo caso, l'uso dei sorgenti di iptables preparati dalla CyanogenMod non è di aiuto: la versione fornita è 1.3.7, troppo vecchia per supportare Xtables-addons e quindi il target `UPMT` sviluppato, inoltre, poiché né il kernel originale né il kernel della CyanogenMod supportano il packet mangling, iptables non offre il target `MARK`. Per questi motivi si è optato per la compilazione manuale dell'ultima versione di iptables, linkando staticamente il plugin userspace del target `UPMT`. Per la compilazione, è stato necessario utilizzare la toolchain fornita da CodeSourcery, poiché sia la toolchain presente nel tree di Android sia quella preparata tramite l'NDK mancavano di numerosi file header. Poiché tale toolchain usa glibc, è stato necessario linkare staticamente anch'essa. Si è comunque provveduto a contattare il team di Android tramite mailing-list per segnalare i problemi con la toolchain NDK ma, al momento della stesura di questo documento, non si è giunti ad una soluzione.

L'inclusione del plugin userspace del target `UPMT` consiste nell'aggiunta del file sorgente `libxt_UPMT.c`, riportato di seguito, alla directory `extensions/`.

```
1  #include <xtables.h>
2
3  static struct xtables_target upmt_tg_reg = {
4    .version       = XTABLES_VERSION,
5    .name          = "UPMT",
6    .family        = AF_UNSPEC,
7  };
8
9  void _init(void) {
10   xtables_register_target(&upmt_tg_reg);
11 }
```

Per la compilazione vera e propria, sono necessari i seguenti comandi:

```
1 ./configure --host=arm-none-linux-gnueabi --disable-shared --enable-static
2 make LDFLAGS="-all-static"
```

L'opzione `--enable-static` alla riga 1 forza la compilazione statica di tutti i plugin nell'eseguibile finale, mentre l'uso della variabile d'ambiente `LDFLAGS` alla riga 2 impone al linker di linkare staticamente la libreria glibc fornita dalla toolchain. Si ottiene così l'eseguibile `iptables-multi`.

6.8 Adattamento dei componenti sviluppati

Durante il processo di sviluppo dei componenti sono state fatte scelte progettuali tali da massimizzare la compatibilità con Android. Tuttavia, per le caratteristiche e le limitazioni intrinseche della piattaforma Android, alcune porzioni necessitano inevitabilmente di riscritture o adattamenti.

Un esempio emblematico è l'implementazione della funzione `pid_to_exe_name()`. Come si può vedere dal codice 6.2, questa è radicalmente diversa dalla versione in ambiente Linux desktop. Come è prevedibile, l'implementazione presentata nel codice 4.2 su Android restituisce sempre `app_process`, ovvero il nome del file eseguibile del processo da cui sono ottenute tutte le istanze di VM Dalvik per forking (vedi sottosezione 6.2.2). Tuttavia, esaminando

Codice 6.2 Funzione `pid_to_exe_name()` che restituisce il package name dell'applicazione in esecuzione su un processo in esecuzione dato il suo PID (ambiente Android).

```
1  // buffer for pid_to_exe_name
2  char cmdline_buffer[PAGE_SIZE];
3
4  const char* pid_to_exe_name(int pid) {
5    struct pid* spid;
6    struct task_struct* task;
7    int len;
8
9    spid = find_get_pid(pid);
10   if (spid == NULL) return NULL;
11   task = pid_task(spid, PIDTYPE_PID);
12   if (task == NULL) return NULL;
13
14   len = proc_pid_cmdline(task, cmdline_buffer);
15   if (len > 0)
16     return cmdline_buffer;
17   else
18     return NULL;
19 }
```

il proc file system, si nota che la voce `cmdline` di un processo riporta il package name dell'applicazione. Si tratta dunque di riscrivere la funzione `pid_to_exe_name()` per leggere tale valore, invocando la stessa funzione chiamata quando il proc file system deve presentarlo all'utente. Tale funzione è `proc_pid_cmdline()`, definita in `fs/proc/base.c` ma purtroppo non esportata dal kernel. Per aggirare questo ostacolo, si è copiata l'implementazione di questa funzione e della funzione `access_process_vm()`, da essa invocata e anch'essa non esportata, definita in `mm/nommu.c`.

Un altro adattamento richiesto è stato causato dalla mancanza delle funzioni di rete `getprotobyname()` e `getprotobynumber()` in bionic. Queste funzioni sono usate da `upmt-appmon` per convertire il numero di protocollo, usato nei messaggi netlink, nel corrispondente nome, usato nei messaggi JSON. Essendo il numero di protocolli supportati limitato, si è provveduto ad effettuare manualmente questa conversione tramite un semplice `if`, usando il mapping numero $\leftrightarrow$ nome protocollo riportata nel file `/etc/protocols` della macchina di sviluppo.

Per facilitare la compilazione, i file sorgenti che compongono il modulo `compat_xtables` di Xtables-addons sono stati copiati nella directory `upmt/kernel/upmt-conntracker/`, così che tale modulo sia compilato congiuntamente al modulo `xt_UPMT`. Questo ha permesso di eliminare la dipendenza esterna da Xtables-addons e di semplificare i makefile eliminando l'uso della variabile d'ambiente `KBUILD_EXTRA_SYMBOLS` (vedi sottosezione 3.2.1).

Sia `upmt-appmon` che `upmtconf` sono stati compilati con la toolchain NDK: questo ha permesso loro di linkare dinamicamente la libreria bionic e, di conseguenza, ciò ha portato a eseguibili di ridotte dimensioni. La toolchain NDK, tuttavia, mancava del file header `include/linux/genetlink.h` richiesto da `upmt-appmon`: si è risolto copiando tale file dall'albero dei sorgenti del kernel nella directory `include/` della toolchain.

Infine, si è provveduto a compilare il plugin userspace per il target UPMT in modo statico

durante la compilazione di iptables, per le motivazioni esposte sopra.

6.9 Mantenere più interfacce connesse contemporaneamente

Rimane un ultimo ostacolo da affrontare: Android cerca di mantenere sempre al più un'interfaccia di rete, tra 3G e Wifi, connessa. In caso siano disponibili nello stesso momento sia la connessione 3G che via Wifi, quest'ultima ha priorità maggiore e causa la disconnessione dell'interfaccia 3G. Questo comportamento è legato sia a motivi di tariffazione del traffico sia, soprattutto a motivi di risparmio energetico. Tuttavia, questo approccio è un problema per la gestione dei flussi applicativi attuata da UPMT: avere sempre solo una sola interfaccia attiva impedisce gli handover automatici basati sulla quality of experience descritti in precedenza.

La strada più diretta per intervenire su questo comportamento indesiderato è quella di modificare il codice sorgente di Android, individuando il punto in cui, all'associazione con una rete Wifi, venga eseguita la disconnessione dell'interfaccia 3G. Dopo l'esame di parecchio codice, si è individuato il metodo `handleConnect()` della classe ConnectivityService, che gestisce le variazioni di connettività. Questo metodo, alla connessione di un'interfaccia, verifica se ci sono altre interfacce connesse e, in tal caso, le disconnette. Le righe $28 \div 31$ sono di notevole interesse. Il metodo `teardown()`, in particolare, si occupa della disconnessione vera e propria dell'interfaccia ma, come specificato in `handleConnect()`, esiste la possibilità per la rete di rifiutare la disconnessione. È questo il caso dell'uso dell'Assisted GPS che, anche quando l'interfaccia wifi è connessa, deve connettersi alla rete 3G per acquisire la posizione, e questa non deve essere disconnessa prima del termine dell'operazione. La linea d'azione, dunque, è di modificare il metodo `teardown()` per restituire sempre `false`, ossia rifiutando sempre la disconnessione.

Per testare la modifica effettuata è necessario compilare l'intero tree di Android: è stata così creata una ROM personalizzata. Alla fine del processo, si otterrà un file `system.img` da flashare sul telefono tramite fastboot.

Una volta installata questa ROM custom sul telefono, si è avuto conferma dell'efficacia di questa soluzione. Come si può vedere dalla screenshot riportata in figura 6.3, entrambe le interfacce 3G e Wifi sono connesse. Inoltre, il log di sistema riporta:

```
1 V/ConnectivityService(   83): Policy requires mobile teardown
2 E/ConnectivityService(   83): Network declined teardown request
```

Nella tabella di routing sono presenti due default gateway, uno per interfaccia:

```
1 # ip r
2 109.52.30.32/30 dev rmnet0  src 109.52.30.33
3 192.168.0.0/24 dev eth0  src 192.168.0.50
4 default via 192.168.0.1 dev eth0
5 default via 109.52.30.34 dev rmnet0
```

L'effettivo funzionamento di entrambe le connessioni può essere provato tramite il comando ping. Nella tabella di routing mostrata sopra, il primo default gateway corrisponde all'interfaccia wifi, quindi tutto il traffico verso l'esterno viene instradato attraverso di essa.

Codice 6.3 Metodo `handleConnect()` della classe ConnectivityService.

```
1  private void handleConnect(NetworkInfo info) {
2    int type = info.getType();
3
4    // snapshot isFailover, because sendConnectedBroadcast() resets it
5    boolean isFailover = info.isFailover();
6    NetworkStateTracker thisNet = mNetTrackers[type];
7
8    // if this is a default net and other default is running
9    // kill the one not preferred
10   if (mNetAttributes[type].isDefault()) {
11     if (mActiveDefaultNetwork != -1 && mActiveDefaultNetwork != type) {
12       if ((type != mNetworkPreference &&
13           mNetAttributes[mActiveDefaultNetwork].mPriority >
14           mNetAttributes[type].mPriority) ||
15           mNetworkPreference == mActiveDefaultNetwork) {
16           // don't accept this one
17           if (DBG) Slog.v(TAG, "Not broadcasting CONNECT_ACTION " +
18               "to torn down network " + info.getTypeName());
19           teardown(thisNet);
20           return;
21       } else {
22         // tear down the other
23         NetworkStateTracker otherNet =
24             mNetTrackers[mActiveDefaultNetwork];
25         if (DBG) Slog.v(TAG, "Policy requires " +
26             otherNet.getNetworkInfo().getTypeName() +
27             " teardown");
28         if (!teardown(otherNet)) {
29           Slog.e(TAG, "Network declined teardown request");
30           return;
31         }
32         if (isFailover) {
33           otherNet.releaseWakeLock();
34         }
35       }
36     }
37     mActiveDefaultNetwork = type;
38   }
39   thisNet.setTeardownRequested(false);
40   thisNet.updateNetworkSettings();
41   handleConnectivityChange();
42   sendConnectedBroadcast(info);
43 }
```

Codice 6.4 Metodo `handleConnect()` della classe ConnectivityService.

```
1 private boolean teardown(NetworkStateTracker netTracker) {
2   if (netTracker.teardown()) {
3     netTracker.setTeardownRequested(true);
4     return true;
5   } else {
6     return false;
7   }
8 }
```

Aggiungendo una rotta statica, è possibile utilizzare anche l'interfaccia 3G: ad esempio, per far uscire i pacchetti verso l'indirizzo IP `79.52.247.83` dall'interfaccia 3G:

```
1 # ip r add 79.52.247.83 dev rmnet0
2
3 # ip r
4 79.52.247.83 dev rmnet0
5 109.54.119.108/30 dev rmnet0  src 109.54.119.110
6 192.168.0.0/24 dev eth0  src 192.168.0.50
7 default via 109.54.119.109 dev rmnet0
8 default via 192.168.0.1 dev eth0
9
10 # ping 79.52.247.83
11 PING 79.52.247.83 (79.52.247.83) 56(84) bytes of data.
12 64 bytes from 79.52.247.83: icmp_seq=1 ttl=49 time=331 ms
13 64 bytes from 79.52.247.83: icmp_seq=2 ttl=49 time=310 ms
14 64 bytes from 79.52.247.83: icmp_seq=3 ttl=49 time=307 ms
15 64 bytes from 79.52.247.83: icmp_seq=4 ttl=49 time=295 ms
```

Wireshark aperto e in sniffing sull'interfaccia dell'host `79.52.247.83` conferma che i pacchetti ICMP provengono dall'indirizzo dalla rete mobile.

Da prove pratiche effettuate, si evince che il metodo `teardown()` viene invocato solo quando l'interfaccia di rete viene disconnessa "per policy": tale modifica, infatti, non interferisce con i controlli manuali che permettono di attivare/disattivare le interfacce a discrezione dell'utente.

La figura 6.3 testimonia che si sta usando il kernel custom, la cui versione è `2.6.32.9-upmt`, compilato dall'utente `venator` sulla macchina di sviluppo `ubuntu-android`. L'uso della ROM custom è testimoniato dalla stringa `full-passion_userdebug` seguita dallo username della macchina di sviluppo (`venator`).

6.10 Test di funzionamento

Poiché la Control Entity non è stata ancora portata su Android, si è testata la piattaforma caricando manualmente i moduli, creando i tunnel iniziali per ciascuna interfaccia, predisponendo le tabelle di routing, regole e rotte necessarie, configurando iptables con marchi e target, avviando l'`upmt-appmon`. Non esistendo una Control Entity che possa inviare i messaggi di configurazione all'`upmt-appmon`, né ricevere le notifiche di nuove connessioni, sono stati predisposti due semplicissimi programmi client e server per la comunicazione

Figura 6.3 Screenshot del telefono durante l'utilizzo del kernel e della ROM custom. Si può vedere nella barra superiore come sia la rete mobile sia la rete wireless siano connesse.

su socket UDP. Sebbene `busybox` fornisca `netcat`, questo è compilato senza il supporto ai socket UDP utilizzati dall'`upmt-appmon`.

La rilevazione di nuove connessioni funziona correttamente. Il nome di applicazione restituito dal modulo `xt_UPMT` è un package name, ad esempio `com.android.browser`, che corrisponde all'applicazione "Browser". La conversione da package name a nome di applicazione può essere facilmente effettuata sfruttando le API di Android, in particolare la classe PackageManager, pertanto risulta conveniente implementarla direttamente nella Control Entity, alla ricezione della notifica di connessione. Questo messaggio JSON, attualmente, viene semplicemente stampato sullo standard output dal server UDP.

L'impostazione di politiche per applicazioni è avvenuta costruendo manualmente il messaggio JSON corrispondente e inviandolo tramite il client UDP.

La gestione dei tunnel e le operazioni sulla PAFT per l'esecuzione di handover sono state effettuate manualmente tramite il tool `upmtconf`. Allo stato attuale, chiaramente, non è prevista alcuna funzionalità di handover automatico alla connessione/disconnessione di un'interfaccia, come implementato dalla Control Entity sulla versione desktop.

7 Valutazione delle prestazioni

In questo capitolo sono illustrate due diverse valutazioni delle prestazioni della soluzione UPMT, sia a livello di sistema che di utente, tramite esperimenti effettuati con la versione desktop di UPMT.

7.1 Valutazione delle prestazioni a livello di sistema

In questa sezione viene presentata la valutazione delle prestazioni della soluzione UPMT, basata sulle misure effettuate nel nostro testbed. Lo scopo principale di questa analisi è valutare l'impatto in termini di carico di elaborazione addizionale dovuto alle procedure di incapsulamento e decapsulamento UPMT.

Per differenti motivi, è importante che il carico di elaborazione sia il più limitato possibile sia nel Mobile Host che nell'Anchor Node. Sull'Anchor Node, il motivo è che la scalabilità della soluzione, ossia quanti flussi e quanti utenti possono essere supportati contemporaneamente, dipende dal carico di elaborazione. Sul Mobile Host, un carico di elaborazione eccessivo causato della gestione della mobilità può limitare le performance delle applicazioni, considerata anche la limitata capacità del processore. Inoltre, il carico di elaborazione ha diretto impatto sui consumi energetici e, quindi, sulla durata delle batterie dei terminali mobili. Le misure effettuate mostrano che il meccanismo di tunneling proposto e implementato in kernel space è molto efficiente rispetto al carico di elaborazione.

Si è usato una metodologia sperimentale, sia per l'analisi del carico di elaborazione sull'Anchor Node, che sul Mobile Host. Si confrontano tre soluzioni, come mostrato in figura 7.1:

1. invio di pacchetti senza il tunneling, usando le sole funzionalità NAT dell'Anchor Node (plain-NAT)

2. invio di pacchetti usando il tunneling UPMT (UPMT)

3. invio di pacchetti usando OpenVPN, una soluzione di tunneling user-space, senza crittografia (*null cipher*) (ULT)

Tabella 7.1 Configurazione hardware delle macchine del testbed.

Host	Configurazione
Mobile Host (sorgente e destinazione iperf)	PC Intel Core 2 Quad 2.66 GHz, NIC Ethernet 100 Mb/s
Anchor Node	PC VIA C7-D Processor 1.5 GHz, 2 NICs Ethernet 100 Mb/s
Correspondent Host (sorgente e destinazione iperf)	PC Intel Core 2 Duo 1.83 GHz, NIC Ethernet 100 Mb/s

Nella tabella 7.1 è riportata la configurazione hardware delle macchine che compongono il testbed utilizzato.

Per quanto riguarda l'Anchor Node, si sono generati flussi sintetici tramite il generatore di traffico iperf utilizzando due host esterni e analizzando il comportamento dell'Anchor Node, sia in termini di perdita di pacchetti che di carico CPU (misurati utilizzando il tool Linux `mpstat`). Si è stato in grado di stimare il carico di saturazione della CPU dell'Anchor Node aumentando il traffico generato. La figura 7.2 mostra la perdita di pacchetti dovuta al sovraccarico della CPU nell'Anchor Node in questi tre scenari. Sull'asse x è riportato il tasso di pacchetti in ingresso (pacchetti/s). Per ciascun scenario è possibile individuare un carico di saturazione: quando il traffico in ingresso è minore del carico di saturazione non c'è perdita di pacchetti, quando il traffico in ingresso è maggiore del carico di saturazione, il traffico in eccesso viene scartato. Il carico di saturazione per UPMT, pari a circa 60000 pacchetti/s, non è molto più basso di quello del plain-NAT, pari a circa 70000 pacchetti/s. Questo mostra che il carico di elaborazione addizionale introdotto dall'incapsulamento e decapsulamento ha impatti limitati sulla scalabilità dell'Anchor Node. D'altro canto, il carico di saturazione della soluzione di tunneling user-space OpenVPN è molto più basso, pari a circa 20000 pacchetti/s. Ciò è dovuto all'inefficienza delle operazioni di tunneling in user-space. Questo risultato è confermato dall'analisi del carico CPU per alcuni intervalli di traffico di input (figura 7.3). Quando il tasso del traffico in ingresso cresce e si avvicina al carico di saturazione, il carico CPU cresce e si avvicina al 100%. Dal carico di saturazione in poi, il carico CPU rimane al 100%.

Per il carico di elaborazione sull'Mobile Host, si configura iperf per generare pacchetti a diversi tassi. Ciò dovrebbe rappresentare una generica applicazione che produca un flusso di pacchetti. Si è misurato il carico di elaborazione complessivo del device sotto i tre scenari menzionati sopra (plain-NAT, UPMT e OpenVPN), allo scopo di valutare l'impatto della soluzione UPMT e di confrontarlo con quello di meccanismi di tunneling in user-space. La figura 7.4 mostra che, come negli esperimenti per l'Anchor Node, la differenza tra lo scenario plain-NAT (niente tunneling) e UPMT (tunneling in kernel) è limitato. In questo caso la differenza relativa è ancora minore, dato che il carico CPU è dominato dalle operazioni user-space effettuate dall'applicazione e in particolare dalle chiamate di sistema necessarie a inoltrare i dati dall'user-space al kernel (lato trasmissione) e viceversa (lato ricezione). D'altro canto, le differenze tra il carico di elaborazione del plain-NAT e di OpenVPN è relativamente alta.

Figura 7.1 Scenari di valutazione delle prestazioni

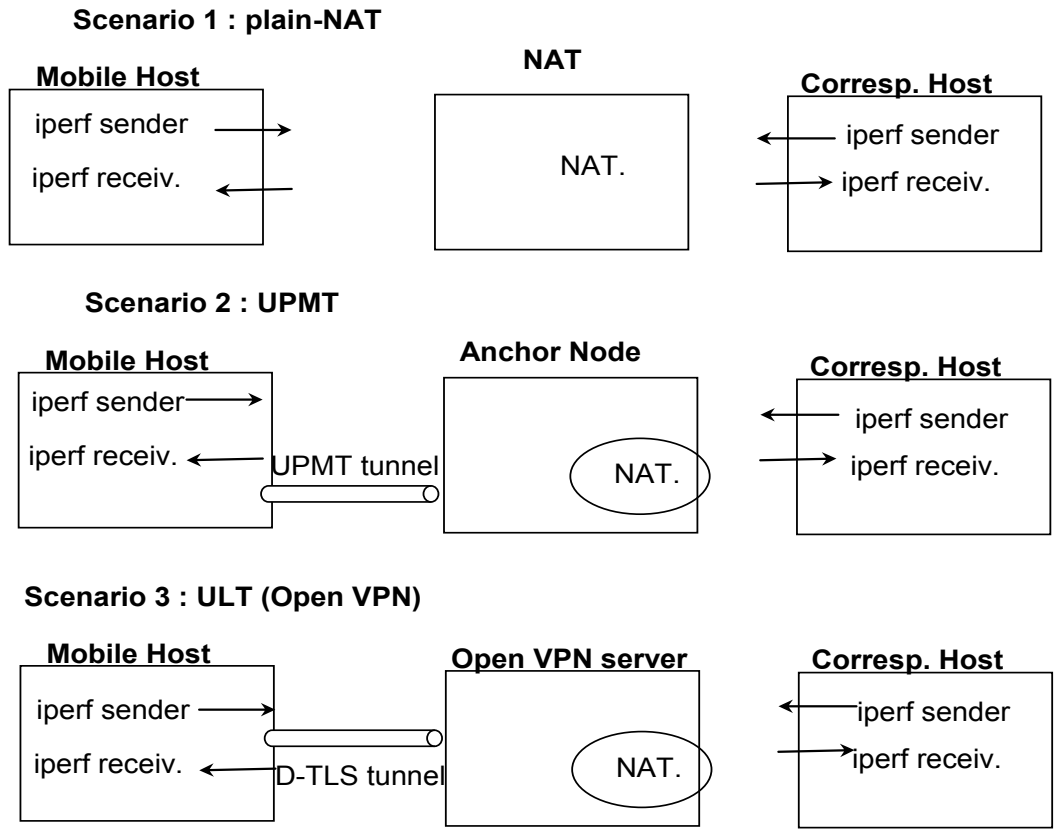


Figura 7.2 Confronto sulla perdita di pacchetti all'Anchor Node.

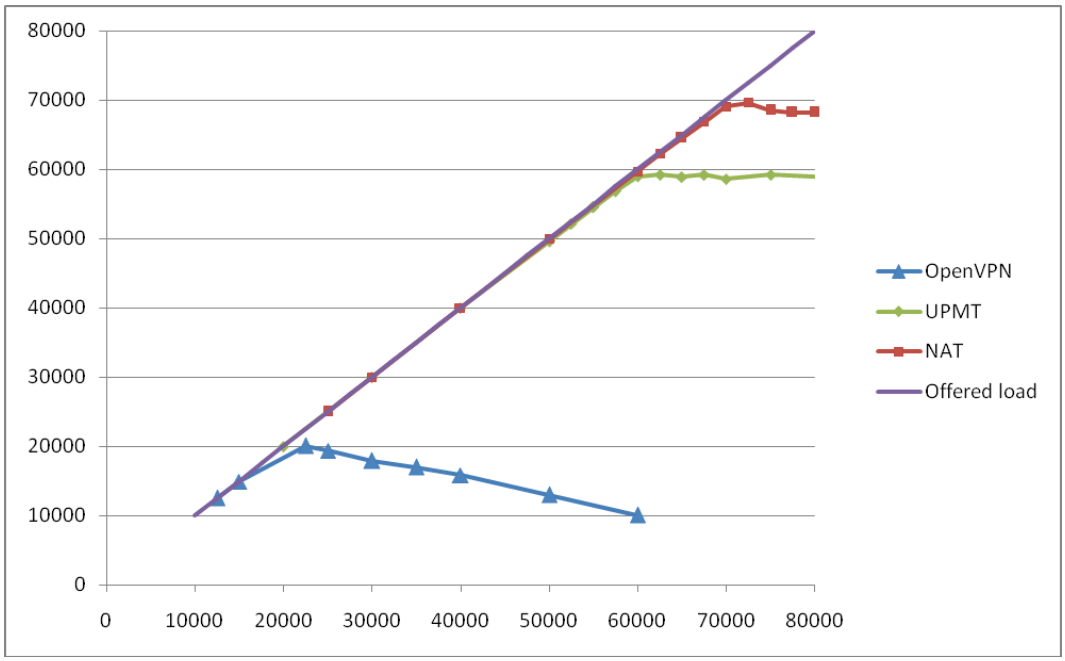


Figura 7.3 Confronto sul carico di elaborazione all'Anchor Node.

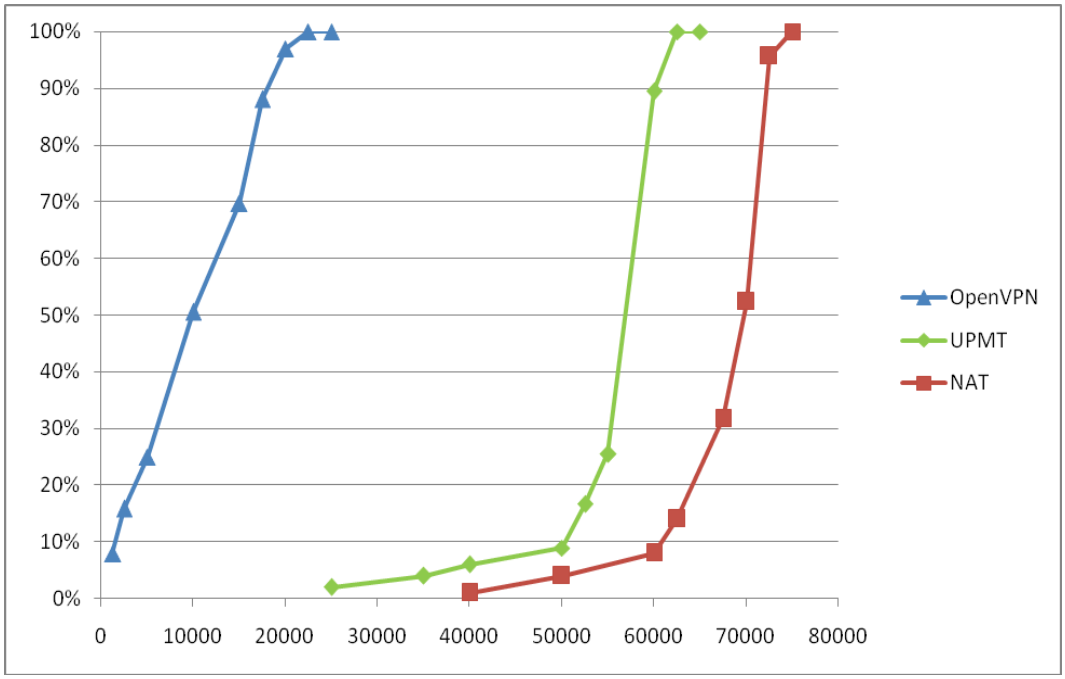
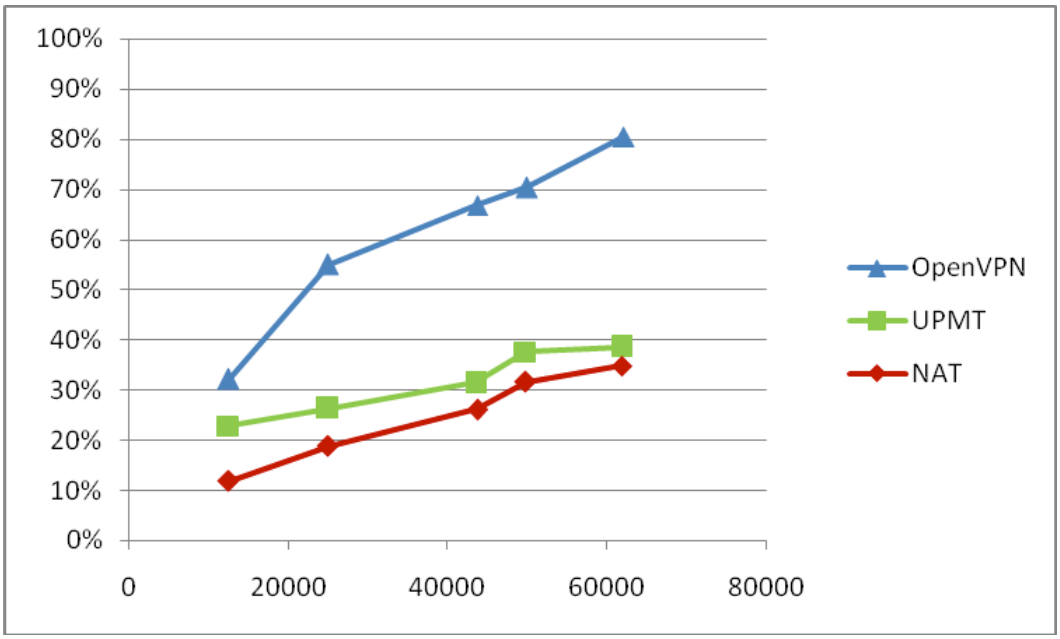


Figura 7.4 Confronto sul carico di elaborazione al Mobile Host.



7.2 Valutazione delle prestazioni a livello utente

In questa sezione vengono valutate le prestazioni percepite dall'utente, concentrandosi sull'impatto degli handover sul throughput. Il testbed è composto dai seguenti componenti:

- un Anchor Node (AN) all'indirizzo IP pubblico 160.80.103.66. Questo è un PC con un processore Pentium M 1.2 GHz, 1.256 GB di memoria RAM e il kernel Linux personalizzato 2.6.35.4-upmt.

- un Mobile Host (MH) con tre interfacce di rete. Questo è un notebook con kernel Linux 2.6.35.4-upmt, processore Core 2 Duo 1.83 GHz, 2 GB di RAM. È equipaggiato con 3 interfacce di rete: `wlan0`, una scheda di rete wireless 802.11g connessa a un access-point nella subnet 192.168.100.0/24; `eth0`, una scheda di rete Ethernet 100 Mbs sulla LAN 192.168.100.0/24; `ppp0`, una chiavetta HSPA USB connessa tramite un link PPP e con indirizzo IP 95.75.196.58. In aggiunta, MH ha un'interfaccia virtuale UPMT chiamata `upmt0` con un VIPAfix 5.6.7.8.

- un numero di Correspondent Host legacy in Internet. In particolare, durante l'esperimento, MH si connetterà a: un web server `www.torvergata.tv` (CH1), un backup file server in esecuzione su un server privato (CH2), un server streaming RTP `vipnrj.yacast.net` (CH3), un server FTP `ftp.archlinux.org` (CH4), un notebook in un'altra LAN con Skype in esecuzione.

Tabella 7.2 Lista di applicazioni e sequenza di handover.

Applicazione	Attività	Tipo di traffico	Handover	CH
Skype	video conference	RT video/audio (UDP) + control	ppp0 → eth0	CH5 + altri
VLC	video streaming	MMS streaming (TCP)	wlan0 → ppp0	CH3
SCP	remote backup	SSH (TCP)	eth0 → wlan0	CH2
Chrome	FTP download	FTP (TCP)	wlan0 → eth0	CH4
Firefox	web tv streaming	HTTP (TCP)	eth0 (no UPMT)	CH1

La dimostrazione descritta nel resto di questa sezione intende mostrare la capacità di UPMT di effettuare handover indipendenti su base applicazione, e può essere riassunta come segue:

1. La UPMT Control Entity viene avviata. Il MH si associa con l'AN e ottiene il VIPA 1.2.3.104.

2. Per ciascuna interfaccia di rete attiva, il MH crea automaticamente un tunnel con l'AN.

3. Cinque applicazioni sono avviate nella stessa sequenza riportata in tabella 7.2. Quattro applicazioni sono poste sotto il controllo di UPMT, un'applicazione (Firefox) non è gestita da UPMT e il suo traffico è instradato da eth0 per l'intera durata della demo. L'interfaccia iniziale per ogni applicazione è la prima interfaccia riportata nella colonna Handover della tabella.

4. Si procede agli handover per le applicazioni sotto il controllo di UPMT, come descritto in tabella, nella sequenza in cui sono riportati.

Si è monitorato il tempo di esecuzione dell'handover tramite cattura di tracce di pacchetti tramite il tool tcpdump. I risultati sono riportati nelle figure 7.5, 7.6, 7.7 e 7.8. Queste mostrano, per ogni applicazione, la somma del traffico ricevuto e trasmesso in bit/s sulla prima e sulla seconda interfaccia di rete usate durante la demo (il periodo iniziale in cui le applicazioni sono avviate in sequenza è stato tagliato). La traccia relativa al traffico di Firefox, non controllato da UPMT, è stata omessa.

I risultati ottenuti sono del tutto soddisfacenti:

- non c'è virtualmente ritardo nell'esecuzione dell'handover; la differenza nei timestamp dei pacchetti ricevuti è dovuta unicamente ai diversi ritardi di rete tra le interfacce

- non c'è perdita di pacchetti e, in effetti, la somma del bitrate per le due interfacce in prossimità dell'istante di handover mantiene il trend della banda aggregata

- le oscillazioni presenti nelle tracce non sono causate dal tunneling UPMT, ma dalle perdite della rete e dal meccanismo di controllo della congestione di TCP: si è osservato, infatti, un identico comportamento senza UPMT.

Figura 7.5 Handover di una videoconferenza Skype.

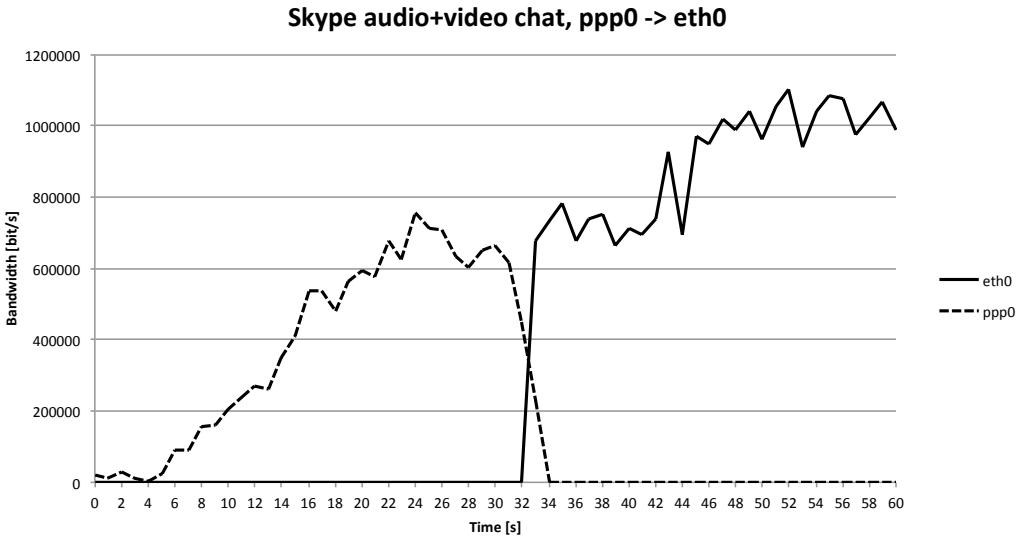


Figura 7.6 Handover di una sessione di video streaming tramite VLC.

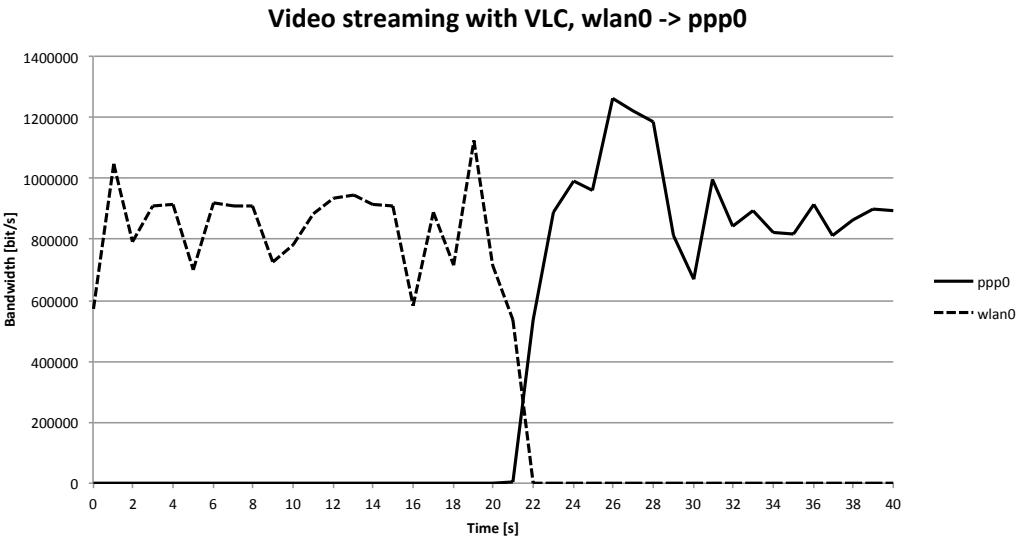


Figura 7.7 Handover di una sessione di backup tramite SCP (upload verso un server remoto).

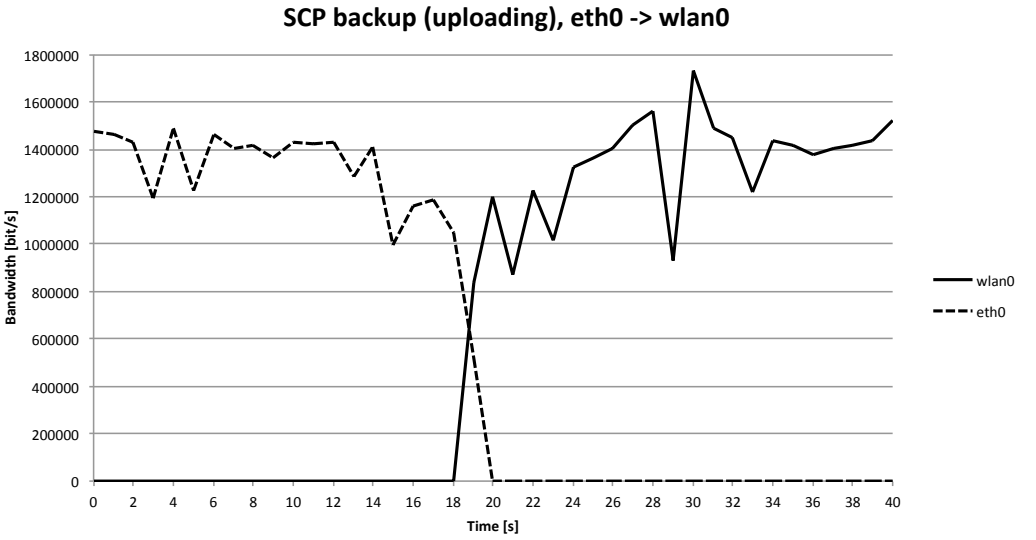
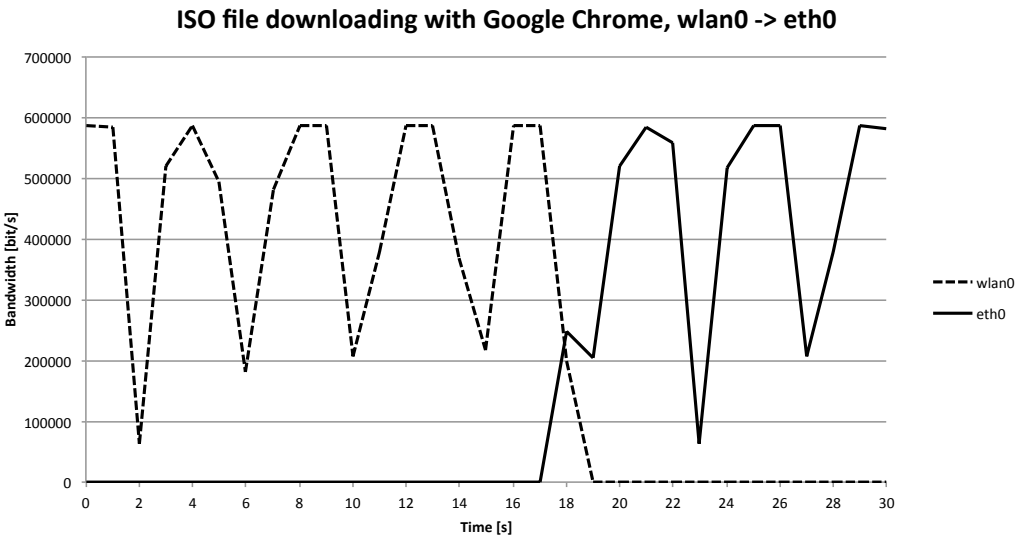


Figura 7.8 Handover di una sessione di download di un file ISO tramite Google Chrome.



8

Conclusioni e sviluppi futuri

I componenti realizzati in questo lavoro di tesi hanno permesso a UPMT di distinguere e operare separatamente sui flussi di ciascuna applicazione. I testbed approntati hanno mostrato l'efficacia della soluzione UPMT, comprensiva delle estensioni sviluppate, tramite esperimenti pratici basati su situazioni di uso comune. Il porting dell'architettura verso Android mostra quanto la soluzione presentata sia adatta ad un contesto mobile reale. Esistono, tuttavia, alcune aree di futura espansione.

Innanzitutto, è necessario iniziare il porting della Control Entity verso Android. Data la sua natura di applicazione Java, questa operazione è piuttosto semplice. Le uniche aree in cui è necessario un intervento più marcato sono quelle più dipendenti dal sistema operativo, ovvero l'interazione con il Network Manager e l'interfaccia grafica. Per quanto riguarda il primo, la gestione delle connessioni di rete di Android è fondamentalmente diversa da quella attuata dal Network Manager. È tuttavia disponibile un'API apposita, che le applicazioni possono implementare. Per quanto riguarda la GUI, sarebbe concettualmente sbagliato tentare di portare la stessa GUI della versione desktop su Android. Oltre a non supportare Swing (il toolkit grafico utilizzato sul desktop), le interfacce dei dispositivi mobili e la loro integrazione nell'ambiente operativo presentano una serie di problematiche inedite, quali usabilità, immediatezza, ecc., molte delle quali legate alla ridotta dimensione degli schermi dei dispositivi mobili.

Disponendo di un prototipo funzionante, grazie ai componenti sviluppati e al lavoro di integrazione nella piattaforma, è naturale cercare di ottimizzarlo quanto più possibile. Una ottimizzazione sicuramente possibile è un sistema di caching per la corrispondenza tra PID e nome dell'applicazione. Sebbene l'implementazione attuale sia piuttosto efficiente, dovendo solo "seguire" una catena di puntatori in diverse strutture dati del kernel, si può fare di meglio. La sfida principale è rappresentata dalla determinazione di un protocollo di invalidazione usato per rimuovere le entry obsolete dalla cache. Tale protocollo dovrà essere il più "reattivo" possibile, in modo tale da minimizzare i falsi positivi, ossia i processi terminati ma ancora presenti in cache, ma al tempo stesso implementabile, idealmente, tramite funzioni utilizzabili da moduli del kernel, senza richiedere ulteriori modifiche al kernel vero e proprio.

In questa tesi non si è coperto affatto il problema della sicurezza in UPMT, che copre temi quali l'autenticazione dei Mobile Host e degli indirizzi IP virtuali, la riservatezza,

autenticazione e integrità sia del traffico dati che del traffico di segnalazione, o la creazione dinamica di chiavi. È già disponibile un'implementazione, chiamata S-UPMT, che utilizza IPSec per cifrare i pacchetti interni inviati sui tunnel, indipendentemente dall'incapsulamento esterno IP su UDP. Uno dei vantaggi dell'approccio proposto consiste nel fatto che l'indirizzo IP all'interno dei tunnel instaurati tra un Mobile Host e un Anchor Node è fisso, pertanto non è richiesta una nuova Security Association IPsec collegata a questo indirizzo IP interno quando il Mobile Host si muove tra differenti reti di accesso. Questo rappresenta un vantaggio nel confronto con altre soluzioni di gestione della mobilità basate su IPsec e nelle quali ogni handover comporta una nuova Security Association per lo scambio di chiavi.

Il progetto UPMT è in rapida evoluzione: gli ultimi aggiornamenti sullo stato dell'implementazione sono disponibili sul sito ufficiale http://netgroup.uniroma2.it/UPMT.

Bibliografia

Libri e articoli

[1] Deguang Le, Xiaoming Fu e Dieter Hogrere. *A Review of Mobility Support Paradigms for the Internet*. Volume 8, No. 1. IEEE Communications surveys. 1st quarter 2006 (citato alle pagg. 2, 4).

[2] Moonjeong Chang, Meejeong Lee e Hyunjeong Lee. *Per-Application Mobility Management with Cross-Layer Based Performance Enhancement*. IEEE WCNC. 2008 (citato a pag. 2).

[3] Marco Bonola, Stefano Salsano e Andrea Polidoro. "UPMT: universal per-application mobility management using tunnels". In: *Proceedings of the 28th IEEE conference on Global telecommunications*. GLOBECOM'09. Honolulu, Hawaii, USA: IEEE Press, 2009, pagg. 2811–2818. ISBN: 978-1-4244-4147-1. URL: `http://portal.acm.org/citation.cfm?id=1811681.1811847` (citato a pag. 2).

[4] Linux kernel documentation. *Documentation/kbuild/modules.txt* (citato a pag. 22).

[5] Daniel P. Bovet e Marco Cesati. *Understanding the Linux kernel*. 3ª ed. ISBN: 0-596-00213-0. O'Reilly, 2003 (citato a pag. 23).

[6] Jan Engelhardt e Nicolas Bouliane. *Writing Netfilter Modules*. 2010. URL: `http://jengelh.medozas.de/documents/Netfilter_Modules.pdf` (citato a pag. 28).

[7] J. Salim et al. *Linux Netlink as an IP Services Protocol*. Request for Comments 3549. Internet Engineering Task Force. IETF, lug. 2003. URL: `http://www.ietf.org/rfc/rfc3549.txt` (citato a pag. 31).

[8] GNU. *Configure Terms and History – GCC Internals Manual*. 2009. URL: `http://gcc.gnu.org/onlinedocs/gccint/Configure-Terms.html` (citato a pag. 76).

Siti web

[9] *Kernel Korner – Why and How to Use Netlink Socket.* URL: http://www.linuxjournal.com/article/7356 (citato a pag. 31).

[10] *Linux 2.6.14 changelog.* 2005. URL: ftp://ftp.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.14 (citato a pag. 34).

[11] *RockSaw – Java library for RAW IP socket.* URL: http://www.savarese.org/software/rocksaw/ (citato a pag. 38).

[12] *JSON.* URL: http://www.json.org/ (citato a pag. 39).

[13] *JavaBean2JSON.* URL: https://minerva.netgroup.uniroma2.it/sms/browser/Javabean2JSON/ (citato a pag. 39).

[14] *cJSON.* URL: http://sourceforge.net/projects/cjson/ (citato a pag. 39).

[15] *poll vs select vs event-based.* URL: http://daniel.haxx.se/docs/poll-vs-select.html (citato a pag. 60).

[16] *MeeGo.* URL: http://meego.com/ (citato a pag. 71).

[17] *Google's Android becomes the world's leading smart phone platform.* URL: http://www.canalys.com/pr/2011/r2011013.html (citato a pag. 72).

[18] *Anatomy & Physiology of an Android.* URL: http://sites.google.com/site/io/anatomy--physiology-of-an-android (citato a pag. 74).

[19] *Sourcery G++ Lite 2010.09-50 for ARM GNU/Linux.* URL: http://www.codesourcery.com/sgpp/lite/arm/portal/release1600 (citato a pag. 76).

[20] *Procedura di rooting per build FRGxxx.* URL: http://forum.xda-developers.com/showpost.php?p=8315805&postcount=2 (citato a pag. 79).

[21] *Get Android Source Code.* URL: http://source.android.com/source/download.html (citato a pag. 79).

[22] *ROM ufficiale Android build FRG83.* URL: http://loadbalancing.modaco.com/download.php?url=paul/nexusone/signed-passion-img-FRG83_0923.zip (citato a pag. 82).

[23] *HOWTO: Unpack, Edit, and Re-Pack Boot Images.* URL: http://android-dls.com/wiki/index.php?title=HOWTO:_Unpack,_Edit,_and_Re-Pack_Boot_Images (citato a pag. 82).

[24] *BusyBox.* URL: http://www.busybox.net/ (citato a pag. 83).

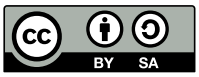
[25] *CyanogenMod.* URL: http://www.cyanogenmod.com/ (citato a pag. 83).

[26] *CyanogenMod on GitHub*. URL: `https://github.com/CyanogenMod` (citato a pag. 83).

Contenuti sotto licenza

[27] Jan Engelhardt. *Netfilter Packet Flow*. LICENZA: Creative Commons Attribution - ShareAlike 3.0 Unported. URL: `http://en.wikipedia.org/wiki/File:Netfilter-packet-flow.svg` (citato a pag. 26).

[28] Jan Engelhardt. *Netfilter components*. LICENZA: Creative Commons Attribution - ShareAlike 3.0 Unported. URL: `http://en.wikipedia.org/wiki/File:Netfilter-components.svg` (citato a pag. 29).



Alessio Bianchi <venator85@gmail.com>

Made on a Mac with LaTeX 2ε

Thesis sources available on:
https://bitbucket.org/venator85/tesi-magistrale

Versione finale: 15 febbraio 2011.