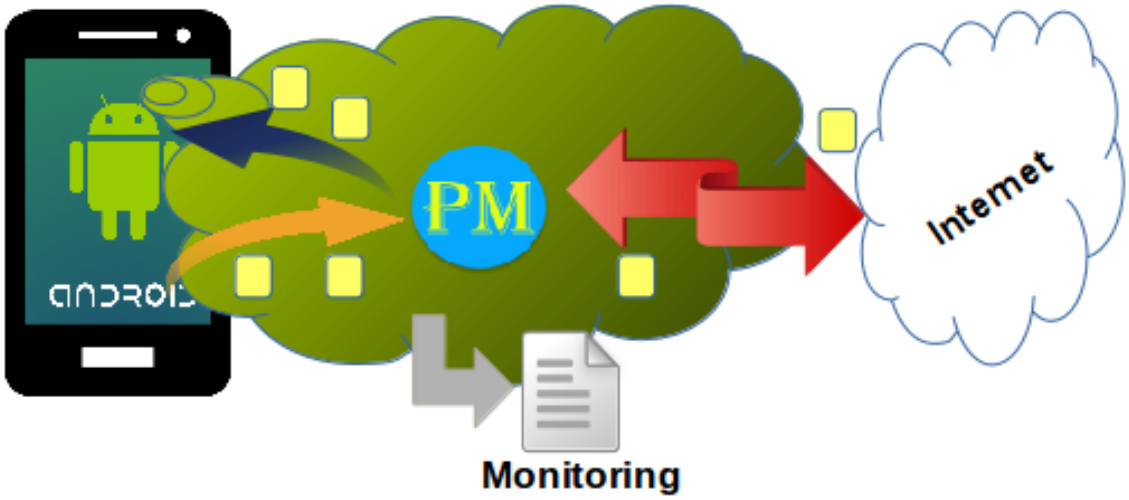
FACOLTA' DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Progetto di Sicurezza informatica e Internet

Traccia A1:
"Proxy in the middle"



Docenti:
Professore F.G.Italiano
Ing. G.Bottazzi

Studenti:
Fabio Alberto Coira
Gabriele Belli
Leonardo De Laurentiis

Anno Accademico: 2014/2015

Indice

Introduzione **1**

1 Il modello del proxy adottato **2**
1.1 Concetto di Proxy Server . . . 2
1.1.1 NAT . . . 3
1.1.2 Proxy server . . . 5
1.2 Modellazione del proxy server ad alto livello . . . 5
1.3 Configurazione proxy e possibili soluzioni su sistema Android . . . 7
1.3.1 Utilizzo di "iptables" su sistema Linux . . . 7
1.3.2 Conoscenze preliminari su VPN . . . 11
1.3.3 User-space VPN . . . 12
1.4 Cos'è il root e come si diventa Super user . . . 13
1.4.1 Vantaggi e svantaggi . . . 13
1.5 Scelte progettuali . . . 15
1.5.1 Scelte generali relative al modulo proxy . . . 16
1.5.2 Scelte relative all'attività di monitoraggio legate all'utilizzo di VPN-Service . . . 17
1.5.3 Riassunto . . . 17

2 Le scelte implementative del Proxy **19**
2.1 Premessa . . . 19
2.1.1 Le classi implementate e la gestione dei log . . . 19
2.1.2 Le classi viste rispetto ai moduli software . . . 21
2.1.3 Richiami teorici . . . 21
2.2 Utilizzo di VpnService . . . 21
2.3 Uso di Java NIO Selector . . . 24
2.3.1 Perché conviene usare un selettore . . . 24
2.3.2 Creazione di un Selector . . . 24
2.3.3 Registrazione dei Channels con il Selector . . . 25
2.3.4 SelectionKey . . . 25
2.3.5 Selezionare i Channels attraverso un Selector . . . 27
2.3.6 selectedKeys() . . . 27
2.3.7 wakeUp() . . . 28
2.3.8 close() . . . 28
2.4 Uso di Java NIO Buffer . . . 28
2.4.1 Buffer . . . 28
2.4.2 Utilizzo generale di un Buffer . . . 28
2.4.3 Buffer Capacity, Position e Limit . . . 29
2.4.4 Buffer types . . . 29
2.4.5 Allocare un Buffer . . . 30
2.4.6 Scrivere dati su un Buffer . . . 30

2.4.7 Metodo flip() . . . 30
2.4.8 Leggere dati da un buffer . . . 31
2.4.9 Metodo rewind() . . . 31
2.4.10 I metodi clear() e compact() . . . 31
2.5 La classe ByteBufferPool . . . 31
2.6 L'Activity Principale: LocalVPN . . . 32
2.7 La classe VPNLocalService . . . 33
2.7.1 Utilizzo di 6 thread . . . 35
2.7.2 VPNRunnable e VPNOutput . . . 36
2.8 La classe Packet . . . 37
2.9 Il tracciamento delle connessioni . . . 38
2.9.1 La classe LinkedHashMap . . . 39
2.9.2 La classe LRUCache . . . 39
2.10 Gestione del protocollo UDP . . . 40
2.10.1 Utilizzo di Java NIO DatagramChannel . . . 40
2.10.2 La classe UDB . . . 41
2.10.3 La classe UDPoutput . . . 42
2.10.4 La classe UDPinput . . . 44
2.11 Gestione del protocollo TCP . . . 45
2.11.1 Utilizzo di Java NIO SocketChannel . . . 45
2.11.2 Gestione della connessione TCP . . . 47
2.11.3 Gli stati TCP . . . 48
2.11.4 La classe TCB . . . 49
2.11.5 La classe TCPoutput . . . 50
2.11.6 La classe TCPinput . . . 55

3 L'attività di Monitoring 57
3.1 Riconoscimento dei protocolli basato sulle porte . . . 57
3.2 Un esempio: il logging di HTTP . . . 57

4 Esempi di funzionamento e di logging 60
4.1 Esempio realizzato tramite utilizzo di un browser . . . 60
4.2 Esempio di log di pacchetti DNS . . . 62
4.3 Esempio di log di pacchetti POP3 . . . 62
4.3.1 La sessione di richieste SMTP tramite il TelnetClient . . . 63
4.3.2 I pacchetti POP3 nel file di log . . . 65
4.4 Ancora sui protocolli di posta . . . 66
4.5 Protocollo "unknown" . . . 66
4.6 Esempio ulteriore: un ping . . . 67

5 Limitazioni riscontrate 69

6 Conclusioni ed eventuali sviluppi futuri 71

7 Breve Manuale d'uso 72

Bibliografia 78

Introduzione

In tema di sicurezza informatica la difesa di uno o più sistemi dall'entità cosiddetta "man in the middle", e dai suoi possibili attacchi ai sistemi informatici tra i quali egli (o esso) si interpone, rappresenta da sempre una tematica di studio di notevole interesse. Ciò è giustamente motivato dalla pericolosità dell'eventualità in cui l'attaccante riesca ad entrare in possesso di informazioni strettamente confidenziali, relative ad una comunicazione tra due interlocutori, intercettando, ad esempio, il traffico Internet scambiato tra i comunicanti, per poi riuscire, in un secondo momento, a decifrarli, o peggio ancora che lo stesso attaccante riesca a mettere in atto attacchi di impersonificazione o alterazione dei messaggi. L'attività di eavesdropping, ovvero l'ascolto della comunicazione privata di due interlocutori, è spesso una azione malevola, nociva per un sistema, atta a compiere talvolta azioni di pirateria informatica. Questa situazione in realtà, può anche essere auspicabile, e la presenza di un intermediario può, in alcuni casi, essere voluta: è il caso, ad esempio, del proxy server.

Un proxy server è un server (un computer system o una vera e propria applicazione) che agisce da intermediario per richieste provenienti da vari clients in cerca di risorse ubicate su altri server nella rete differenti dal proxy server contattato. In una situazione generale, quindi, un client si connette ad un proxy server, richiedendo un qualche servizio, o una qualche pagina, offerti da un altro server ubicato altrove all'interno della rete internet globale; è il proxy server successivamente ad evadere tali richieste per conto del client ed a fornire le risposte al rispettivo client richiedente. Oggi la maggior parte dei proxy server appartengono alla categoria cosiddetta dei "web proxies", i quali facilitano l'accesso al contenuto del World Wide Web ed in alcune situazioni, e con diverse modalità, provvedono all'anonimato.

L'obiettivo principale del progetto svolto consiste nella realizzazione di una applicazione che implementi un modulo software proxy per dispositivi mobili dotati di sistema operativo Android, in grado di interporsi, in maniera trasparente ed efficiente, tra il dispositivo e tutte le comunicazioni da e verso Internet. Tale modulo software proxy non ha, in questo progetto, né lo scopo di semplificare la navigazione, né tantomeno di garantire l'anonimato all'utente, ma ha l'obiettivo di comportarsi come un vero e proprio "man in the middle", con il proposito di monitorare il traffico Internet del dispositivo da e verso la rete, intercettando correttamente sia le sessioni del browser che quelle di tutte le App installate sul dispositivo stesso. Tale proxy inoltre si comporterà da "packet analyzer", analizzando i pacchetti inoltrati e tenendo traccia delle varie sessioni proxate, mediante un file di log. Le informazioni memorizzate in tale file (destination URL, date, time, protocol, method (eventuale), protocol response, user-agent, DNS usato, tipo diconnettività usata se 3/4G o Wifi, etc.) saranno dunque lo strumento di analisi della navigazione Internet del device interessato.

Capitolo 1

Il modello del proxy adottato

Prima di procedere con la presentazione del lavoro svolto è necessario chiedersi quale sia il reale contesto in cui si deve operare, nonché cercare di comprendere a fondo il cuore del problema che si sta affrontando. In questo capitolo, dunque, si cerca di illustrare una serie di aspetti teorici di analisi e di compiere una astrazione ad alto livello, per gettare le basi su quanto verrà trattato nei capitoli successivi, dove si illustrerà l'implementazione di tale modello astratto. Dapprima ci si pone perciò un quesito relativo a cosa sia realmente un proxy server, focalizzando l'attenzione sulle sue caratteristiche essenziali e peculiari che hanno portato ad adottare precise scelte progettuali, alcune delle quali dettate anche dal semplice buon senso. In seguito si illustrano diverse possibili strade che avrebbero potuto essere percorse per affrontare il problema. Infine si riassumono le scelte effettuate per offrirne una visione generale e sintetica.

1.1 Concetto di Proxy Server

Nelle reti di calcolatori, un proxy server è un server (un sistema informatico o una applicazione) che agisce come intermediario per richieste da parte di vari client alla ricerca di risorse possedute da altri server. In generale un client si connette al proxy server, richiedendo qualche servizio, come un file o una pagina web, e il proxy server evade tali richieste per suo conto. Di conseguenza un proxy si interpone tra un client ed un server, vagliando e decidendo se inoltrare o meno le richieste e le risposte dall'una all'altra entità oppure addirittura se reindirizzarle da qualche altra parte all'interno della rete. Il client, come implicitamente sottolineato più volte, si collega al proxy invece che al server "finale", ed invia ad esso le proprie richieste. Il proxy, a sua volta, si collega al server e inoltra la richiesta del client, ne riceve l'eventuale risposta ed infine la re-inoltra al client richiedente. Tutto questo è riassunto in figura 1.1.

L'utilizzo di un proxy offre i seguenti vantaggi:

- possibilità di effettuare **caching**, cioè di immagazzinare per un certo tempo i risultati delle richieste di un utente, così da evadere le analoghe richieste di un altro utente effettuate a breve distanza di tempo, permettendo così un miglioramento delle prestazioni ed una riduzione del consumo di ampiezza di banda;

- possibilità di effettuare un **monitoraggio**, dunque di tenere traccia di tutte le operazioni effettuate (ad esempio, tutte le pagine web visitate), offrendo statistiche ed osservazioni sull'utilizzo della rete;

- possibilità di imporre un **controllo** cioè applicare un set di regole definite dall'amministratore di sistema per determinare quali richieste inoltrare e quali rifiutare,

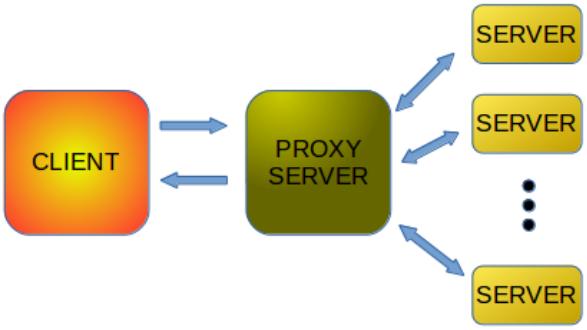


Figura 1.1: Schema di funzionamento di una interazione Client-Server mediante un Proxy Server

oppure limitare l'ampiezza di banda utilizzata dai client, o anche filtrare le pagine Web in transito, ad esempio bloccando quelle il cui contenuto è ritenuto offensivo;

- capacità di garantire la **privacy** e l'**anonimato**, mascherando il vero indirizzo IP di un client in modo tale che il server finale non possa risalire all'identità di chi ha effettuato la richiesta originale.

Oltre alle caratteristiche elencate, esistono tante altre caratteristiche che rendono i server proxy tra loro differenti; ad esempio il fatto che essi siano appartenenti alla categoria cosiddetta dei **transparent proxy**, i quali non modificano il contenuto del traffico Internet inoltrato, oppure appartenenti alla categoria cosiddetta dei **non trasparent proxy**, i quali invece intervengono "chirurgicamente" sui pacchetti. Sono menzionabili inoltre altre distinzioni, legate ad esempio alla gestione di un particolare protocollo di rete, (es. **web proxy**), oppure le categorie cosiddette di tipo **reverse** o **forwarding** proxy.

Si sottolinea infine come, a differenza di bridge e router che lavorano ad un livello ISO/OSI più basso, nella maggior parte dei casi il concetto di "proxy" si riferisca ad un sistema software che opera a livello 7 della pila OSI: è il caso, ad esempio, dei servers proxy che realizzano meccanismi di caching, i quali sono implementati al livello 7, e processano talvolta specifici protocolli Internet, come HTTP e FTP (generalmente il numero dei protocolli applicativi gestito è limitato). Inoltre, a differenza di un packet filter che opera a livello 3 e 4 della pila OSI, un proxy server è capace di analizzare i pacchetti a livello 7, fornendo quindi la possibilità di avere granularità più fine ed agire da filtro sui contenuti di livello applicativo. Tuttavia, un'altra modalità di implementazione di un proxy è quella di "inserirlo" al livello 3 della pila OSI: tale configurazione è meglio nota come Network Address Translation (NAT). La differenze tra queste tecnologie di proxy sono il livello al quale essi operano e la procedura differente di configurazione per client proxy e server proxy. Nella configurazione del client, in un proxy al livello 3 (NAT), la configurazione del gateway [1] è sufficiente.

1.1.1 NAT

La NAT è una tecnica che consiste nel modificare gli indirizzi IP dei pacchetti in transito su un sistema che agisce da router all'interno di una comunicazione tra due o più host. Il

[1] Un gateway è un dispositivo di rete che opera a livello di rete, e ai livelli superiori, del modello ISO/OSI. Il suo scopo principale è quello di veicolare i pacchetti di rete all'esterno di una rete locale (LAN). Spesso i gateway non si limitano a fornire la funzionalità di base di routing, ma integrano altri servizi offerti da e verso la rete locale quali ad esempio proxy DNS, firewall, NAT,etc., che sono appunto servizi propri di uno strato di rete più elevato, ovvero quello applicativo.

NAT è spesso implementato dai router e dai firewall. Si può distinguere tra:

- **source NAT (SNAT)**: le connessioni effettuate da uno o più computer vengono alterate in modo da presentare verso l'esterno uno o più indirizzi IP diversi da quelli originali. Quindi chi riceve le connessioni le vede provenire da un indirizzo diverso da quello utilizzato da chi effettivamente le genera (cioè viene modificato l'indirizzo sorgente del pacchetto).
- **destination NAT (DNAT)**: le connessioni effettuate da uno o più computer vengono alterate in modo da venire redirette verso indirizzi IP diversi da quelli originali. Quindi chi effettua le connessioni si collega in realtà ad un indirizzo diverso da quello che seleziona. (viene modificato l'indirizzo di destinazione del pacchetto che inizia una nuova connessione).

I pacchetti che viaggiano in senso opposto verranno modificati in modo corrispondente, in modo da dare, ad almeno uno dei due computer che stanno comunicando, l'illusione di comunicare con un indirizzo IP diverso da quello effettivamente utilizzato dalla controparte. Per implementare il NAT, un router ha quindi bisogno di effettuare il **tracciamento delle connessioni**, ovvero di tenere traccia di tutte le connessioni che lo attraversano. Per "connessione" in questo contesto si intende un flusso bidirezionale di pacchetti tra due host, identificati da particolari caratteristiche a livelli superiori a quello di rete (IP):

- nel caso di **TCP** è una **connessione TCP in senso proprio**, **caratterizzata da una coppia di porte**;
- **nel caso di UDP**, per quanto UDP sia un protocollo di trasporto senza connessione, **viene considerata connessione uno scambio di pacchetti UDP tra due host che usino la stessa coppia di numeri di porta**;
- altri protocolli vengono gestiti in modo analogo, usando caratteristiche del pacchetto a livelli superiori a quello IP per identificare i pacchetti che appartengono ad una stessa connessione.

Come detto, in genere,in questa soluzione ci si ferma al livello 3 o 4 della pila OSI.

La NAT dunque si basa su di una tabella contenente la corrispondenza tra le socket interne e le socket esterne in uso (ovvero l'insieme di indirizzi IP e porte di comunicazione). Ad esempio, quando un client va a visitare una pagina web su un server esterno, il suo indirizzo e la sua porta di origine vengono tradotti, e la corrispondenza viene registrata nella tabella. Quando poi arriva la risposta dal server web esterno, la tabella permette di capire chi è il legittimo richiedente di quei dati, quindi viene effettuata la traduzione inversa e mandata al client. Tutte le comunicazioni dall'esterno che non sono in tabella vengono scartate. Di seguito si riporta un esempio esplicativo:

*"Il client **192.168.0.1** utilizza la porta **1234** per aprire una sessione WEB col server **212.41.203.1** sulla porta **80**. Il gateway che sfrutta il NAT intercetta la richiesta, e modifica l'intestazione dei pacchetti TCP in modo che risultino generati da lui, quindi per esempio **99.102.13.1** porta **1435**. Nella tabella viene inserita questa corrispondenza:*

Origine	Traduzione	Destinazione
192.168.0.1:1234	**99.102.13.1:1435**	**212.41.203.1:80**

*La destinazione riceverà i pacchetti da **99.102.13.1:1435**, e a questa socket invierà i dati delle pagine WEB. Il gateway a questo punto riceve la risposta da **212.41.203.1:80**, consulta la tabella*

*e modifica l' intestazione dei pacchetti ricevuti per mandarli verso **192.168.0.1:1234**; se arriva qualche pacchetto da indirizzi IP non richiesti, e quindi non presenti in tabella, questi vengono scartati.*
La traduzione può essere di vari tipi:

- **statica**: consiste nell'inserire in tabella una corrispondenza **fissa** che consente di rendere disponibile all' esterno un servizio che gira su un host interno alla rete (ad esempio far diventare pubblici dei servizi che girano su macchine locali, sfruttando un solo indirizzo IP pubblico);
- **dinamica**: (o **IP Masquerade**): è la traduzione utilizzata nell' esempio esposto, dove la tabella gestisce le connessioni. Tali connessioni hanno associato un certo timeout, scaduto il quale viene rimosso il record dalla tabella: tutto quello che non è in tabella viene eliminato a priori, quindi non esistono strade per arrivare dall' esterno su un PC interno, a meno che non sia presente una NAT statica.

Il funzionamento di queste tabelle è semplice ed intuitivo, ed è per questo che le implementazioni della NAT sono quasi sempre affidabili e in linea generale prive di "bachi". E' importante sottolineare che tale tecnica lavora ad un livello basso dello stack OSI, operando modifiche esclusivamente sulle intestazioni dei pacchetti TCP e UDP, motivo per cui non è in grado di analizzare il contenuto delle informazioni di livello superiore, e quindi di poter proteggere la rete interna in modo completo, e di fornire delle "policies" per le varie utenze.

1.1.2 Proxy server

I server proxy lavorano ad un livello ISO/OSI più alto (rispetto alle già citate NAT), il livello applicativo, e ciò significa che tali proxy non si occupano dei livelli TCP e IP, ma **operano, con granularità più fine delle NAT, con dei protocolli applicativi** come HTTP, FTP, SMTP, POP3. Tali server proxy ricevono dai client le richieste di servizio come se fossero i server a cui tali richieste sono destinate. A questo punto **rigenerano da zero le richieste** verso i server esterni, funzionando proprio come dei "mediatori" tra i client della rete interna e il mondo esterno. Un esempio pratico di funzionamento di una interazione fra un client ed un server proxy è il seguente:

*"Il client chiede attraverso il suo browser il sito **www.uniroma2.it**. Tale richiesta viene inoltrata al server proxy, che ne prende carico e la inoltra, come se fosse lui stesso un browser, sulla sua interfaccia esterna verso internet. Quando arrivano i dati richiesti, il proxy li invia al client che li riceve come se il proxy fosse il server WEB."*

La differenza con la NAT è notevole: in questo schema non si considerano porte e indirizzi IP! Attraverso il proxy vengono considerati ed analizzati solo i dati inerenti a protocolli di alto livello (nel caso dell'esempio HTTP, ma in generale si possono implementare proxy server per quasi tutti i protocolli di livello applicativo).

1.2 Modellazione del proxy server ad alto livello

Sulla base delle specifiche progettuali, è possibile osservare il proxy server come una "black box", e individuare le caratteristiche salienti che lo modellano. Le principali features individuate sono le seguenti:

- il proxy gestisce e supporta scambio dati su Internet;

- **tutto** il traffico Internet che proviene dal client passa per il proxy server;
- **tutto** il traffico Internet diretto al client passa per il proxy server;
- il proxy tiene traccia delle connessioni, o comunque delle richieste effettuate;
- il proxy gestisce richieste concorrenti;
- il proxy deve riuscire ad effettuare un **monitoraggio** della rete, esteso su **tutto** il traffico Internet generato dal dispositivo;
- vengono chiuse/aperte le connessioni in maniera trasparente, perciò non ci si accorge della presenza del proxy server,in questo senso si intende il requisito di **trasparenza all'utente**;
- il lavoro del proxy è svolto in **tempi ragionevoli** e con grande riguardo verso la **power consumption** in ogni operazione, requisito importantissimo nel campo del **"mobile software engineering"**;
- l'implementazione del proxy si compone di uno o più moduli software, e non si tratta di un componente sito all'interno della rete, ma di un' **applicazione implementata su un dispositivo** con sistema operativo **Android**.

Come illustrato nel paragrafo precedente esistono diverse soluzioni a tale problema. Per come è concepito un proxy server in linea generale, si è già detto che esso opera al livello applicativo della pila OSI, e si potrebbe pensare in prima analisi di implementare in tal modo il modulo software proxy richiesto da specifica. Ad esempio si potrebbero far cooperare diversi moduli software, ciascuno dei quali si occupa di una parte della gestione del server proxy a livello applicativo, ed utilizzarli per gestire le connessioni del dispositivo e del traffico Internet di ciascun protocollo. Il lavoro tuttavia è abbastanza oneroso, ed oltretutto il servizio richiesto da specifica è esclusivamente quello di **monitoraggio** e **logging** della attività di rete, azioni da intraprendere in modo comune per tutti i protocolli. Non sono richiesti dei servizi di gestione di specifici protocolli applicativi, ad esempio di "packet-filtering", motivo per cui non si rende necessario salire così in alto nella pila OSI. Per i detti motivi **una soluzione di questo tipo non è stata considerata nella progettazione del modulo realizzato**.

La soluzione più congeniale al soddisfacimento di tutte le specifiche progettuali richieste consiste **nell'operare ai livelli 3 e 4 della pila OSI**, scendendo quindi più in basso, ed operando con datagrammi IP e segmenti TCP e UDP, in modo da poter avere una **gestione completa e uniforme** del traffico di rete e poter loggare tutte le informazioni interessanti con **semplicità**. Oltre a **realizzare una NAT software** l'obiettivo è quello di analizzare i pacchetti a livello applicativo, per capire a quale protocollo appartengano i messaggi scambiati dal dispositivo con la rete e viceversa, e registrare in un file di log le informazioni ritenute più essenziali.

In altre parole quindi, il proxy deve ricevere traffico internet, quindi deve essere in grado di riconoscere ed utilizzare i protocolli su cui si basa la rete Internet. Inoltre, nel caso in cui una App o un client siano configurati per dialogare con il proxy tutto il traffico Internet che normalmente sarebbe diretto nella rete dovrà essere convogliato verso questo intermediario, che dovrà analizzare ogni pacchetto, guardare, ad esempio, le intestazioni, effettuare esso stesso una comunicazione con il server a cui le comunicazioni originali sono dirette, ricevere eventuali messaggi di riscontro o comunque traffico di dati diretti al client originario, ed infine reinoltrarli allo stesso mittente originario (tutto questo attraverso una NAT a livello 3 e 4 della pila OSI). In uno schema di questo tipo molteplici

applicazioni lato client potrebbero effettuare richieste simultanee, di conseguenza deve esserci una gestione di tipo concorrente delle richieste, con particolare attenzione alle "race conditions" su strutture dati condivise. Deve essere presente infine una serie di moduli software da eseguire per garantire i servizi aggiuntivi di logging operando a livello 7 della pila OSI.

1.3 Configurazione proxy e possibili soluzioni su sistema Android

Come del resto già osservato, ciò che la specifica del progetto richiede non è l'impiego di un server proxy remoto, localizzato in qualche parte del mondo, a cui collegarsi per usufruire di alcuni servizi, connettendo dunque semplicemente il proprio dispositivo a quello specifico server proxy.

Il cuore del problema risiede, invece, nella realizzazione di un vero e proprio server proxy che risieda su un dispositivo fisico mobile. La prima iniziale difficoltà consiste nel dirottare tutto il traffico Internet del dispositivo verso il detto proxy. Dunque, sfruttando le API messe a disposizione dal sistema operativo per la comunicazione remota tra processi, il client deve poter comunicare con il proxy, e anche gli stessi server remoti che inviano dati al dispositivo devono poter inviare dati al proxy, che si frappone tra i due interlocutori. Come detto, il proxy presenterà una struttura tale da comportarsi da intermediario, disponendo esso stesso dunque di un proprio indirizzo IP. Il punto cruciale nella realizzazione del sistema in esame consiste dunque nel modificare le rotte dei pacchetti in modo da inviare tutto il traffico in uscita ad un preciso indirizzo IP, quello del proxy server, differente dall'indirizzo dei reali destinatari dei datagrammi IP. Il traffico in uscita dal proxy server verso il dispositivo interno, e quello diretto dalla rete al proxy, transita attraverso canali di comunicazioni aperti dal proxy stesso verso la rete e verso il dispositivo. L' indirizzo IP dei pacchetti inoltrati all'esterno è quello del proxy server, che saprà gestire al meglio le risposte provenienti dall' esterno reindirizzandole al mittente originario, semplicemente guardando la propria tabella interna delle corrispondenze degli indirizzi delle connessioni. Diverso è l'invio dei pacchetti dal dispositivo al proxy, o dal proxy al dispositivo. Qui si dovrà intervenire in qualche modo, per inoltrare pacchetti con precisi header, verso un destinatario che di default non è quello autentico. In sostanza quello che si richiede è di realizzare una sorta di NAT, di cui già si è discusso.

La natura del problema porta a due possibili soluzioni nel "mondo Android":

- l'utilizzo di direttive impartite al kernel del proprio sistema operativo (e.g. iptables) per dirottare i pacchetti verso un proxy software installato a bordo del dispositivo;
- una soluzione software basata sull'utilizzo di un servizio Android, pensato originariamente per realizzare soluzioni VPN, denominato VPNService.

1.3.1 Utilizzo di "iptables" su sistema Linux

Su di una distribuzione Linux, ad esempio Ubuntu oppure lo stesso kernel di Android (basato a sua volta su kernel Linux), basterebbe semplicemente modificare le regole di gestione del traffico di rete per risolvere il problema discusso; infatti si potrebbe pensare di "loggarsi" come Superuser e di lanciare da terminale dei comandi per redirigere il traffico diretto su porte note. Rediretto il traffico Internet diretto ad una specifica porta (ad esempio la porta 80) su un'altra porta (ad esempio la porta 8080) quello che si farà

è realizzare un modulo software in un qualche linguaggio di programmazione (C, Java, Python), e si gestiranno tramite esso le sockets, la concorrenza, verrà realizzata e consultata la tabella delle connessioni, e ci si servirà di diverse strutture dati necessarie alla corretta implementazione di un autentico proxy server, il quale, come si intuisce, verrà messo in ascolto sulla porta specifica verso cui tutto il traffico è stato dirottato.

Panoramica su iptables

"Iptables" è un'applicazione che permette agli amministratori di un sistema di configurare le tabelle, le catene e le regole di netfilter [2]. Nello specifico "iptables" si applica a IPv4 (per IPv6 è utilizzata la versione "ip6tables").

Dato che iptables modifica il funzionamento del sistema operativo, per essere eseguito è necessario entrare nel sistema in qualità di utente amministratore, che nei sistemi di tipo Unix è il cosiddetto utente root, il quale ha i permessi per compiere qualsiasi tipo di operazione. Sulla maggior parte dei sistemi Linux, iptables è installato come **/usr/sbin/iptables**. La lista completa delle funzionalità del comando è consultabile nella relativa documentazione, che può essere visualizzata con il comando "man iptables". Possono essere definite differenti tabelle. Ogni tabella contiene un numero di catene built-in e un certo numero di catene definito dall'utente, dove ogni catena è una lista di regole che può fare "match" con un set di pacchetti. Ogni regola specifica cosa fare con un pacchetto che realizzi un "match". Tale operazione è in gergo chiamata "target", e implica un salto ad una catena definita dall'utente nella stessa tabella. Ciascuna regola del firewall specifica dei criteri per un pacchetto e per un "target". Se il pacchetto non realizza un "match", viene esaminata la prossima regola nella catena: se quest'ultima realizza il "match", allora lo prossima regola è specificata dal valore del "target" associato, che può essere il nome di una catena definita dall'utente o un valore speciale: **ACCEPT**,**DROP**, **QUEUE**, or **RETURN**. Esistono diverse tabelle indipendenti, due di loro sono:

- **filter**, che è la tabella di default, (se non è passata l'opzione -t), e che contiene le catene di built-in: **INPUT** (per pacchetti destinati a sockets locali), FORWARD (per pacchetti che devono essere routati attraverso la box), e OUTPUT (per pacchetti localmente generati).

- **nat**, tabella consultata quando un pacchetto che crea una nuova connessione viene incontrato. Presenta le catene di built-in: **PREROUTING** (per alterare i pacchetti non appena entrano), **OUTPUT** (per alterare i pacchetti generati localmente prima di effettuare il routing sulla rete), e **POSTROUTING** (per alterare i pacchetti che sono in procinto di uscire).

Un comando come il seguente:

```
# iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT
--to-port 8080
```

dove:

[2] E' un componente del kernel del sistema operativo Linux, che permette l'intercettazione e manipolazione dei pacchetti che attraversano il computer. Netfilter permette di realizzare alcune funzionalità di rete avanzate come la realizzazione di firewall basata sul filtraggio stateful dei pacchetti o configurazioni anche complesse di NAT, un sistema di traduzione automatica degli indirizzi IP, tra cui la condivisione di un' unica connessione Internet tra diversi computer di una rete locale, o ancora la manipolazione dei pacchetti in transito. Per configurare netfilter si usa il programma iptables, che permette di definire le regole per i filtri di rete e il reindirizzamento NAT. Spesso con il termine iptables ci si riferisce all'intera infrastruttura, incluso netfilter

- **-t** *table* specifica la tabella;
- **-A** è un opzione di iptables, un comando che dice di appendere una o più regole alla catena selezionata;
- **PREROUTING**, ha il significato già illustrato relativo a **nat**;
- **-p** *protocol* indica il protocollo, che può essere tcp, udp, icmp, oppure tutti questi;
- **-dport** *port* indica il numero di porta di destinazione originale;
- **-j** *target* specifica il target della regola;
- **–to-port** *port* specifica la porta di destinazione verso cui dirottare.

consente ad esempio di redirigere tutto il traffico diretto sulla porta 80, che di default è TCP ed è dedicata al protocollo applicativo HTTP, sulla porta 8080. E' evidente dunque che attraverso comandi di questo tipo, in qualità di Superuser, si può facilmente configurare il routing del traffico Internet del proprio dispositivo, e quindi gestire lo stesso traffico che arriva sulla porta scelta.

Poiché la specifica del progetto prevede l'utilizzo di un dispositivo mobile con sistema operativo Android, il quale poggia le sue basi su di un kernel Linux, è dunque teoricamente possibile operare secondo la logica illustrata eseguendo da Superuser dei comandi speciali (e.g. iptables) da terminale. Tuttavia non è affatto semplice ottenere i privilegi di Superuser su un device mobile, e, inoltre, può comportare degli svantaggi, come verrà illustrato più avanti. Tale scelta (detta in gergo operazione di "rooting" del device) limiterebbe implicitamente il pubblico a cui una possibile release (i.e. sul market di Android) potrebbe essere destinata, dato che la procedura di "rooting" non è praticabile facilmente da un cosiddetto "utente medio".

Realizzazione di un semplice NAT con iptables su sistema Linux

Il NAT, come già discusso nei precedenti paragrafi, è una tecnica che permette di manipolare l' header dei datagrammi IP quando viaggiano nel nodo di interconnessione tra le reti, e può essere o SNAT o DNAT.
Prima di configurare una regola NAT, è bene ricordarsi di attivare l'IP Forwarding:

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

- **Possibili esempi di utilizzo di DNAT(DestinationNAT)**, ricordando che DNAT deve essere utilizzato nella fase di pre-routing.
  - *Reindirizzamento del traffico diretto all'indirizzo di rete pubblico **80.23.45.178** e porta **80** verso l'IP di un server web **192.168.0.254** ed il traffico verso lo stesso indirizzo pubblico verso il terminal server **192.168.0.253** con porta **RDP 3389**, nel caso in cui il protocollo utilizzato sia RDP appunto.*

    ```
    # iptables -t nat -A PREROUTING -p tcp -d 80.23.45.178 -dport 80
     -j DNAT --to-destination 192.168.0.254:80
    # iptables -t nat -A PREROUTING -p tcp -d 80.23.145.178 -dport 3389
     -j DNAT --to-destination 192.168.0.253:3389
    ```

  - *Reindirizzamento di tutto il traffico dell'interfaccia associata a **eth1** verso l'indirizzo **192.168.0.1***

```
# iptables -t nat -A PREROUTING -i eth1 -j DNAT
 --to-destination 192.168.0.1
```

– *Reindirizzamento del traffico HTTP proveniente dalla rete interna sull'interfaccia eth0 verso un Trasparent-Proxy installato all'indirizzo **192.168.0.1** porta **8080***

```
# iptables -t nat -A PREROUTING -p tcp --dport 80 -i eth0
 -j DNAT --to-destination 192.168.0.1:8080
```

– *Reindirizzamento di tutto il traffico per l'indirizzo **192.168.0.1** verso l'IP **192.168.20.1***

```
# iptables -t nat -A OUTPUT -d 192.168.0.1
 -j DNAT --to-destination 192.168.20.1
```

In sequenza a tali comandi bisogna poi inserire quanto segue per abilitare il forwarding:

```
#iptables -t filter -P FORWARD ACCEPT
#echo 1 > /proc/sys/net/ipv4/ip_forward
```

- **Possibili esempi di utilizzo di SNAT(SourceNAT)**, ricordando che SNAT deve essere eseguito in fase di post-routing, un attimo prima che il pacchetto venga trattato dal processo di routing e pronto per essere immesso sulla rete. Questo è un dettaglio importante, perché solo così qualsiasi altra componente nella Linux box (instradamento, filtraggio dei pacchetti) vedrà il pacchetto come invariato. In questo schema inoltre si potrà utilizzare l'opzione -o (interfaccia uscente).

  – *Il **Masquerading** (o Mascheramento) è anch'essa una tecnica di SNAT dove tutti i pacchetti provenienti dai client interni alla rete privata vengono mascherati con un solo IP address. Si possono mascherare (Masquerading) tutti i pacchetti in uscita dall'interfaccia **eth1**(interfaccia collegata ad internet) con lo stesso l'IP address assegnato all'interfaccia **eth1** attraverso il comando*:

  ```
  # iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
  ```

  – *Alterazione in **192.168.1.254** di qualsiasi indirizzo IP sorgente proveniente dall'interfaccia **eth0***:

  ```
  # iptables -t nat -A POSTROUTING -o eth0
   -j SNAT --to-source 192.168.1.254
  ```

  – *Impostare un pool di indirizzi con cui mascherare tutti i pacchetti in uscita dall'interfaccia eth0 in un range che va da **192.168.0.1** a **192.168.0.5***:

  ```
  # iptables -t nat -A POSTROUTING -o eth0
   -j SNAT --to-source 192.168.0.1-192.168.0.5
  ```

Infine si possono utilizzare i seguenti comandi per:

- visualizzare le tabelle NAT

  ```
  # iptables -t nat -L
  ```

- ripristinare le tabelle NAT

  ```
  # iptables -t nat -F
  ```

NAT su Android attraverso iptables

Poiché iptables è un modulo presente di default in AOSP [3], è possibile utilizzare **netfilter** e scrivere del codice in linguaggio C per gestirlo. Ad esempio, si potrebbe creare un progetto Android, e scrivere uno o più JNI file, utilizzare ndk-build per compilarli, e successivamente utilizzando **adb** effettuare un push dell'eseguibile sul file system del dispositivo Android per eseguirlo. Inoltre dal dispositivo è possibile utilizzare la shell **adb shell** ed utilizzare direttamente i comandi di iptables come un utente root, equivalentemente a come si farebbe da un terminale Linux. In linea generale non tutti i device supportano iptables, e comunque richiede i permessi di Superuser su Android.

Per capire se il proprio device mobile abbia installato a bordo e supporti iptables basta lanciare il seguente comando da adb shell o nel Terminal Emulator:

```
iptables -L -t nat
```

E' possibile consultare alternativamente la guida su **Droidwall**, una interfaccia applicativa front-end che richiede i permessi di root e che supporta delle librerie apposite che consentono di fare il NAT, oltre a supportare il programma iptables.

1.3.2 Conoscenze preliminari su VPN

Per meglio comprendere come sfruttare eventualmente il servizio Android denominato VPNService, si è distillato un insieme di conoscenze generali sulle soluzioni VPN. Attraverso tali conoscenze si è riusciti a comprendere **come** e soprattutto **se** convenisse utilizzare il detto servizio messo a disposizione dal sistema operativo, posto che **l'obiettivo della specifica di progetto è ben diverso da quello della realizzazione di una soluzione VPN**.

Una rete VPN (Virtual Private Network) permette a computer ubicati in sedi fisiche differenti di stabilire un collegamento tramite Internet. Questa soluzione elimina la necessità di dover ricorrere a costose linee dedicate. Poiché le connessioni a Internet sono connessioni pubbliche, e quindi non protette per definizione, sono esposte al rischio che i pirati informatici possano intercettare e modificare i dati trasmessi sul Web. Con l'utlizzo di una rete VPN è eventualmente possibile crittografare i dati e inviarli solo ad un computer (o gruppo di computer) specifico e, in altre parole, creare una rete privata accessibile solo agli utenti autorizzati. La rete in questione è però una **rete virtuale** poiché il collegamento tra i computer remoti non è fisico (i.e. punto-punto) ma basato sulla rete Internet. Nella pratica, una rete VPN è utile sia in ambito aziendale che in ambito privato. Scopo delle reti VPN è offrire alle aziende, a un costo inferiore, le stesse possibilità delle linee private in affitto ma sfruttando reti condivise pubbliche: si può vedere dunque una VPN come l'estensione, a scala geografica, di una rete locale privata aziendale che colleghi tra loro host "interni" all'azienda stessa variamente dislocati su un ampio territorio, sfruttando l'instradamento tramite IP per il trasporto su scala geografica e realizzando di fatto una rete LAN, detta appunto virtuale e privata, logicamente del tutto equivalente a un'infrastruttura fisica di rete (ossia con collegamenti fisici) appositamente dedicata. La feature principale che viene offerta da una VPN riguarda la sicurezza e la protezione dei dati personali, che vengono sempre mantenuti al sicuro da sguardi indiscreti. Solo chi ha accesso alla VPN può eventualmente vedere i dati che vengono scambiati, condivisi e utilizzati dagli utenti. Il modello di riferimento generale di una VPN prevede la presenza

[3] Acronimo di Android Open Source Project, è un progetto open source di Google che permette la compilazione dai sorgenti di Android "Vanilla" (= puro, come sui Nexus). Tutte le ROM più celebri (Cyanogenmod, AOKP, Paranoid Android, o AOSPA, e molte altre) sono basate sull'AOSP, solo che hanno pesantemente modificato i sorgenti apportando molte più features e personalizzazioni.

di un collegamento sicuro su Internet tra gli end-points della VPN, ove tale collegamento è chiamato **tunnel** ed il **tunnelling** utilizzato consiste in un insieme di tecniche per cui un generico protocollo viene incapsulato in un protocollo dello stesso livello o di livello inferiore per realizzare configurazioni particolari. Le due tipologie di tunnel più utilizzate sono:

- tunnel IP;
- tunnel UDP.

. La sicurezza del tunnel è ottenuta attraverso la cifratura dei dati che sono inseriti all'interno del tunnel, e l'autenticazione degli end-point. Le reti costruite utilizzando la cifratura dei dati sono chiamate "Secure VPN".

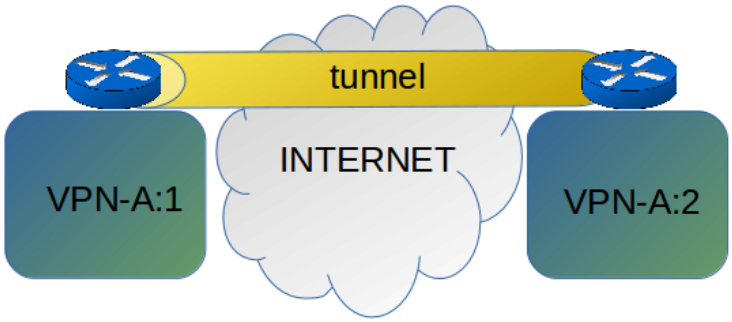


Figura 1.2: Per realizzare soluzioni VPN è solitamente utlizzato un cosidetto tunnel

Un tunnel può essere reso visibile o meno al livello applicativo. Quando visibile, è spesso presentato come una scheda virtuale di livello 3 che trasferisce/riceve pacchetti IP (TUN driver) o come scheda virtuale di livello 2 (TAP driver) che trasferisce/riceve trame Ethernet. Con un TUN si trasferiscono pacchetti (in questo caso si parla di Routed VPN) mentre con un TAP si trasferiscono trame Ethernet (Bridged VPN). Oltre che sulla scheda virtuale, un TUN/TAP driver può ricevere/trasmettere pacchetti su una zona comune di memoria (detta "character device"); una applicazione che scrive sul character device invoca una ricezione sul device di rete virtuale, mentre una applicazione che scrive sul device di rete virtuale invoca una ricezione sul character device.
Il già citato servizio Android VPNService, una volta avviato, inizializza una autentica interfaccia virtuale della tipologia "TUN driver" appena descritta.
Nella figura seguente è illustrato un esempio di trasferimento/ricezione per mezzo di un TUN/TAP driver.

1.3.3 User-space VPN

Nell'insieme di possibili soluzioni VPN è presente quella di realizzare una cosiddetta "User-Space VPN", i cui link sono dei tunnel UDP o TCP gestiti da uno specifico **tool di livello applicativo**. La sicurezza su questi tunnel è garantita da (Datagram) Transport Layer Security TLS (DTLS). Esse sono dette "User-Space VPN" poichè sono basate sui sockets, che sono controllabili dallo user-space. Alla ricezione di un pacchetto proveniente da una applicazione locale, il tool "User-Space VPN" controlla l'indirizzo IP di destinazione e decide su quale socket UDP (TCP) incapsulare il pacchetto cifrato. Pertanto il tool "User-Space VPN" possiede una sua **tabella di routing (overlay)** svincolata dalla tabella di routing dell'OS.

Le entry di questa tabella overlay sono del tipo:

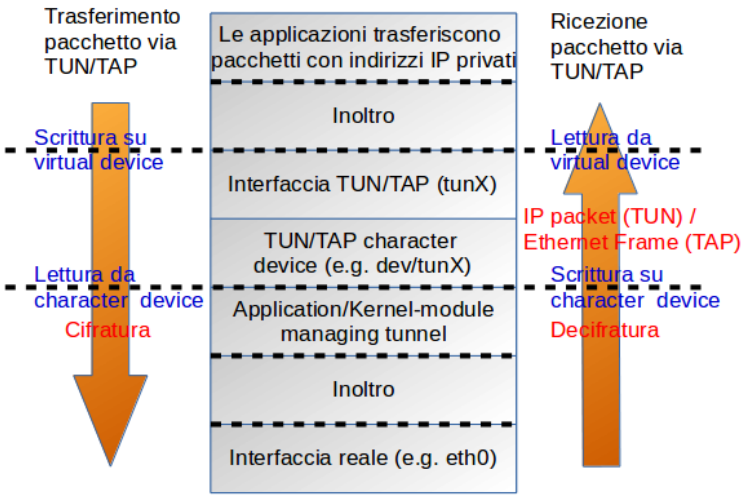


Figura 1.3: Schema di attraversamento dei vari livelli di cui si compone un tunnel basato su TUN/TAP.

`<netid, mask, public_ip_da, udp_port>`

Il tool "User-Space VPN" remoto decifra, autentica e decapsula i pacchetti IP (Ethernet) entranti e li inietta sul TUN/TAP driver (i.e., scrive sul TUN/TAP character device). Il forwarding dell' OS provvederà alle successive operazioni di forwarding intra-VPN.

1.4 Cos'è il root e come si diventa Super user

Come già discusso, una delle possibili soluzioni per implementare il modulo proxy richiesto da specifica prevederebbe la procedura di "rooting" del dispositivo. Il root, dall'inglese "radice", è un processo derivato dai sistemi Linux e applicato anche su Android, a sua volta basato su kernel Linux, per ottenere i diritti di Super-user ed avere così accesso senza limiti al sistema. Il processo di root passa attraverso l'**installazione di un nuovo kernel** nel sistema. Installando un nuovo kernel modificato si avrà dunque a disposizione una gamma di soluzioni più ampia e talvolta migliore per la gestione dei processi sullo smartphone.

1.4.1 Vantaggi e svantaggi

Ottenere i permessi di root sul proprio dispositivo Android ha indubbiamente dei pro e dei contro.

Pro

1. **Un gran numero di app in più disponibili per l'installazione**: quando l'utente richiede di installare una App che preveda l'utilizzo di una qualche operazione un po' più avanzata, vengono richiesti i permessi di root per far interagire più in profondità le app col sistema. Il root, tra le altre cose, permette di installare anche **programmi per disinstallare bloatware** (app preinstallate sui dispositivi acquistati), **firewall**, **sistemi di gestione multitouch** per controllare a fondo il dispositivo con i gesti e tanto altro.

2. **Personalizzazione indefinita**: i dispositivi Android sono famosi per la loro possibilità di personalizzazione (es. cambiare launcher, icon pack, sfondi live, suonerie,

suoni della tastiera), tuttavia si tratta pur sempre di una personalizzazione limitata ad un cambiamento estetico superficiale. Grazie ai permessi di root è possibile modificare i file di sistema a piacere, compresi i suoni di sistema e l'animazione di avvio.

3. **Prestazioni oltre ogni limite**: esistono già molte app in grado di liberare la RAM e di velocizzare il telefono ma con i metodi "normali" non è possibile superare i limiti imposti dall'hardware. I permessi di root permettono, ad esempio, di superare la frequenza di clock limite della CPU tramite **overclocking**, affidandosi ad un kernel modificato, spesso "tunable" con opportune modifiche a dei file di configurazione.

4. **Migliore gestione del power-saving**: nonostante gli smartphone continuino ad evolversi, il tallone d'Achille della limitatezza della batteria sussiste. Ottenendo i permessi di root è possibile accedere a diverse funzioni che permettono di regolare il processore ad un setting di opportuno risparmio energetico, impostando ad esempio un limite massimo di clock inferiore a quello originalmente previsto. Tante altre modifiche sono rese possibili (comprese procedure di undervolt, proprie dell'hardware) affidandosi ad un kernel modificato.

5. **Automatizzazione generale**: applicazioni come Tasker permettono di automatizzare praticamente qualsiasi funzione dello smartphone. Ad esempio ad una determinata condizione (come l'inserimento delle cuffie, il luogo in cui ci si trova o un determinato orario) è possibile far corrispondere un'azione (il lancio di un'app, invio sms, riproduzione di una canzone, etc). Questa applicazione specifica, ad essere precisi, è funzionante anche senza i permessi di root, ma una volta ottenuti questi ultimi è possibile accedere alla lista completa delle possibili funzionalità implementate, come l'attivazione automatica di internet, del GPS, accensione dello schermo, tuning della velocità della CPU e tanto altro ancora.

6. **Permette di "flashare" una custom ROM sul dispositivo mobile** [4]: questo è sicuramente il motivo per il quale la maggior parte degli utenti Android decide di voler ottenere i permessi di root. Oltre alla loro feature principale rappresentata dalla personalizzazione radicale di uno smartphone, i maggiori sviluppatori di ROM Custom hanno più volte dimostrato come spesso le prestazioni generali delle ROM Custom oltrepassino quelle delle corrispondenti ROM stock (i.e. per funzionalità, fluidità e performance).

7. **Niente più pubblicità**: sebbene per molti sviluppatori l'utilizzo delle pubblicità nelle app sia l'unica fonte di guadagno, alcune app contengono però pubblicità invasive che non solo consumano un gran numero di dati ma che spesso attivano abbonamenti senza il nostro consenso. Perciò l'utilizzo di app come **AdBlock Plus**, permettono di visualizzare solo i normali ad, bloccando quelli che penalizzano l'esperienza d'uso delle applicazioni. L'utilizzo di questa applicazione e di quelle simili è possibile solo per chi ha ottenuto i permessi di root.

[4] E' una versione del sistema operativo spesso più completa, con funzionalità estese e personalizzazione e fluidità spesso maggiori della versione stock ("stock" è il termine con cui viene indicata la ROM che si ha a disposizione quando si acquista il telefono,possibilmente sviluppate ad esempio dalle aziende Samsung, HTC, LG, Sony...). Le custom rom talvolta sono necessarie anche per tenere costantemente aggiornato il proprio sistema operativo, poichè le case produttrici, per motivi spesso commerciali, rilasciano aggiornamenti per il proprio dispositivo fino ad una specifica versione di android; attraverso le custom rom si può continuare a mantenere aggiornato il proprio dispositivo. Nella quasi totalità dei casi le custom ROM sono rilasciate e mantenute da comunità di sviluppatori autonomi

8. **Effettuare backup completi**: ogni dispositivo Android ha la possibilità di fare un backup delle applicazioni e di alcuni dati, all'interno delle impostazioni o tramite l'utilizzo di software gestionali come **Kies**. Anche se oltre a questi si decidesse di utilizzare i vari servizi cloud, nulla potrebbe competere con le funzioni offerte dai permessi di root: ad esempio con l'utilizzo dell'app **Titanium Backup**, si possono effettuare copie delle immagini di sistema del dispositivo e spostarle su SD o computer, oltre ad automatizzarne il procedimento.

9. **Ottenere funzioni nascoste**: uno dei punti di forza di Android, è l'incredibile varietà di dispositivi sui quali è installabile. Un ulteriore motivo per ottenere i permessi di root è che talvolta permettono di godere delle fantastiche caratteristiche esclusive di un gruppo di smartphone (e.g. la fascia dei "top di gamma").

10. **Reversibilità del processo**: in qualsiasi momento è possibile fare un passo indietro, togliendo i permessi di root e tornando al firmware originale del dispositivo e, nella maggior parte dei casi, ripristinando anche la garanzia.

Contro

Come ogni cosa, anche l'operazione di ottenere i permessi di root ha dei contro.

1. **Sicurezza a rischio**: poiché tali permessi permettono di comunicare in modo avanzato col proprio dispositivo e di installare nuove app altrimenti non utilizzabili, nulla impedisce ad **applicazioni volutamente dannose** di agire negativamente sul nostro sistema, danneggiando il dispositivo o, in circostanze peggiori, mettendo a repentaglio la privacy e i dati sensibili dell'utente.

2. **Invalidazione della garanzia**: i permessi di root **invalidano la garanzia**.

3. **Per dispositivi di case non molto note è più complesso**: i procedimenti per ottenere i permessi di root, come detto, quasi sempre non arrivano per vie ufficiali ma vengono forniti da altri utenti o gruppi più o meno estesi, che si uniscono in team per cercare di specificarli e renderli pubblici; è quindi normale che essi si concentrino maggiormente sui dispositivi di brand più noti. Di conseguenza, se si possiede un device di un brand poco noto, molto spesso si deve ricorrere a procedimenti più complessi e ad hoc, che aumentano la possibilità di commettere errori e danneggiare il device.

4. **Problemi con gli aggiornamenti**: non è più possibile aggiornare il dispositivo a seconda del produttore e del terminale usato. I meno fortunati dovranno quindi dimenticarsi della comodità degli aggiornamenti via OTA [5] e attraverso software gestionali (come Kies per Samsung ad esempio), anche se si potranno comunque installare manualmente.

1.5 Scelte progettuali

Il cuore del lavoro svolto consiste nella realizzazione dell' applicazione **Proxy In The Middle** per dispositivi mobili equipaggiati con sistema operativo Android, e nell'implementazione di moduli software atti a garantire lo svolgimento dell'attività di **monitoraggio e logging** del traffico Internet, per una profilazione quanto più completa e accurata del dispositivo, in conformità a quanto già discusso nei paragrafi precedenti.

[5] "over the air", cioè in remoto, con una notifica inviata direttamente dal centro aggiornamenti

1.5.1 Scelte generali relative al modulo proxy

Per realizzare una applicazione quanto più portabile e flessibile, **si è preferito non vincolare l'applicazione stessa all'uso di un kernel "rootato"**. Sebbene certamente fosse questa la soluzione più semplice e immediata da perseguire,la necessità di utilizzo dei diritti di Super-user è stata sin da subito percepita più come un difetto che non come un pregio. Perciò è stata adottata la seconda delle soluzioni già menzionate nel **paragrafo 1.3**, ovvero la soluzione che prevede l'utilizzo del servizio Android denominato VPNService, costituendo una serie di moduli software che operano con delle sockets TCP e UDP, e utilizzando una interfaccia di tipo TUN (e quindi una routed VPN che gestisce pacchetti IP) predisposta da VPNService, realizzando così con questa soluzione una autentica NAT che richiederebbe invece permessi di Superuser se realizzata, ad esempio, attraverso iptables.

Il servizio VPNService consente di metter su una interfaccia virtuale avente **un indirizzo IP virtuale, e modificabile, differente da quello del device**, e nel caso in cui venga effettuato un corretto setting si ottiene un'importante feature: tutto il traffico di rete viene convogliato verso l'interfaccia virtuale; si noti che per "traffico di rete" si intende sia il traffico in ingresso che quello in uscita dal dispositivo. Utilizzando tale interfaccia, ed opportuni moduli software che aprono dei nuovi canali di comunicazione (sockets) verso i servers originariamente destinatari del traffico di rete generato dal device, l'effetto ottenuto è quello di aver realizzato un modulo software che assolva a tutte le funzioni tipiche di un vero e proprio server proxy. Attraverso un **meccanismo di caching** si tiene traccia dei canali di comunicazione instaurati, ciascuno connesso verso un preciso IP di destinazione ed una specifica porta di destinazione. All'interno della struttura del modulo software proxy realizzato, sono previsti diversi thread, a cui viene demandato un compito di gestione ben preciso, e per permettere la comunicazione tra i sotto-moduli si utilizzano delle **code**. Le code sono state previste come risorse condivise, il cui accesso è sincronizzato tramite meccanismi di locking. Un sotto-modulo gestisce la comunicazione con l'esterno e l'apertura dei canali, mentre un altro si occupa della comunicazione tra le risorse e l'interfaccia della VPN. Infine, un ulteriore sotto-modulo software offre il servizio di **monitoring e logging**, registrando la cronologia delle sessioni proxate.

Si è scelto, senza perdita di generalità, di supportare unicamente **IPv4** come protocollo a livello di rete, e solamente sockets **TCP** e **UDP** per quanto riguarda i protocolli a livello di trasporto.

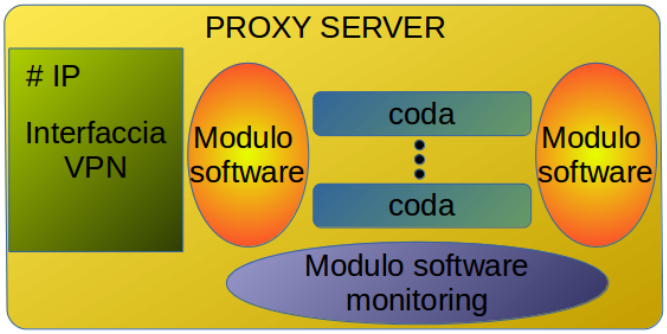


Figura 1.4: L'implementazione della specifica di progetto prevede la presenza di vari moduli software interagenti tra loro attraverso delle strutture condivise di tipo coda

Come feature aggiuntiva, si è scelto di realizzare delle activity nelle quali l'utente potrà scegliere **quali app escludere dal meccanismo** messo su tramite VPNService. Le app escluse pertanto utilizzeranno la rete come se non fosse presente alcun meccanismo di dirottamento e monitoring. Si noti che, di default, tutte le app utilizzeranno il meccanismo;

l'unica possibilità di escludere una app è di aggiungerla alla lista delle app escluse dal monitoring.

1.5.2 Scelte relative all'attività di monitoraggio legate all'utilizzo di VPNService

Analizzando la specifica del progetto, si è scelto di monitorare e di effettuare attività di logging di pacchetti afferenti a diversi protocolli di livello applicativo. Ciò ha comportato problemi non banali, legati allo "spacchettamento" delle informazioni di livello applicativo da datagrammi di livello 3 (livello IP). Difatti il servizio Android scelto (VPNService) permette di interfacciarsi con una interfaccia virtuale che si "inietta" a livello 3 dello stack ISO/OSI. A tale interfaccia virtuale vengono dirottati, utilizzando un setting specifico, tutti i pacchetti destinati (provenienti) alla (dalla) rete. L'interfaccia virtuale si è rivelata essenziale per realizzare, senza privilegi di root, il dirottamento di tutto il traffico, ed ha rappresentato, nello schema progettuale, una sotto-specie di proxy server parziale. Una volta destinato tutto il traffico sulla interfaccia virtuale, il problema è diventato differente. E' stato necessario effettuare operazioni di prelievo (di push) dei pacchetti dalla (sulla) interfaccia virtuale, a cui appunto sono convogliati i pacchetti stessi, per poterli "spacchettare", analizzare ed estrarre informazioni di livello superiore. Successivamente al prelievo e all'analisi dei pacchetti dall'interfaccia virtuale si è proceduto a instaurare nuove connessioni con i server originariamente destinatari delle informazioni (cosa affatto semplice, data la mancanza di Java al supporto di raw socket [6]) e a memorizzare le associazioni tra i pacchetti uscenti e gli indirizzi a cui essi fossero destinati. Ottenute risposte dai detti server sui canali di comunicazione aperti in precedenza, si è poi dovuto procedere seguendo un percorso inverso, estraendo cioè informazioni di livello applicativo e fornendo pacchetti di livello 3 opportunamente forgiati all'interfaccia virtuale, in modo che essa li potesse fornire indietro alle applicazioni legittime destinatarie dei determinati pacchetti di risposta. Come detto l'attivita di "spacchettamento" si è resa necessaria in quanto l'interfaccia virtuale opera al livello 3 della pila OSI. In particolare, prelevato un pacchetto di livello 3 si è dovuta effettuare una analisi di tutti i campi dell'intestazione per capirne la dimensione e lo specifico protocollo di livello superiore, per poi quindi estrarre un pacchetto di livello di trasporto. Avendo a disposizione un pacchetto di livello 4, analogamente, si è proceduto ad analizzare ed estrarre tutte le informazioni concernenti al payload di livello applicativo. Estratto infine il payload di dati di livello applicativo si è proceduto ad analizzarne il contenuto tramite parser costruiti ad-hoc per lo specifico protocollo di livello applicativo.

In particolare si è scelto di supportare **l'analisi ed il logging** di pacchetti dei seguenti protocolli di livello applicativo: **HTTP, HTTPS, DNS, SMTP, POP3, IMAP**.

1.5.3 Riassunto

Si è visto cos'è un proxy server in generale, e si è discusso delle due possibili soluzioni adottate per realizzarlo, una a livello 7 della pila OSI ed una a livello 3 e 4. Si è visto come la soluzione ai livelli più bassi si presti meglio al servizio di **monitoring** richiesto da speci-

[6] In ogni livello della pila OSI, un generico pacchetto presenta due sezioni disgiunte: Intestazione e Payload. In una socket non Raw, ovvero standard, può essere modificato esclusivamente il Payload a livello di trasporto, mentre è compito del sistema operativo creare intestazioni ai livelli di trasporto, di rete e Data Link . In una Raw socket invece è possibile determinare ogni sezione del pacchetto, quindi anche l'intestazione, oltre al payload.

fica. Dovendo realizzare un proxy su un device mobile equipaggiato con Android si è poi illustrata una possibile scelta alternativa tra due soluzioni che permettessero l'implementazione dello stesso modulo richiesto da specifica: una prima scelta, che vincola l'utente che voglia ottenere i permessi di root ad utilizzare una versione del kernel modificata invalidando così la garanzia, ed una seconda, basata sul servizio Android denominato VPNService, la quale non prevede alcun tipo di modifica al dispositivo. Nella volontà di realizzare una applicazione **portabile** e più **ad alto livello** (ovvero senza dover realizzare codice sorgente del kernel Android), si è optato per una soluzione del secondo tipo (che utilizza, cioè, VPNService).

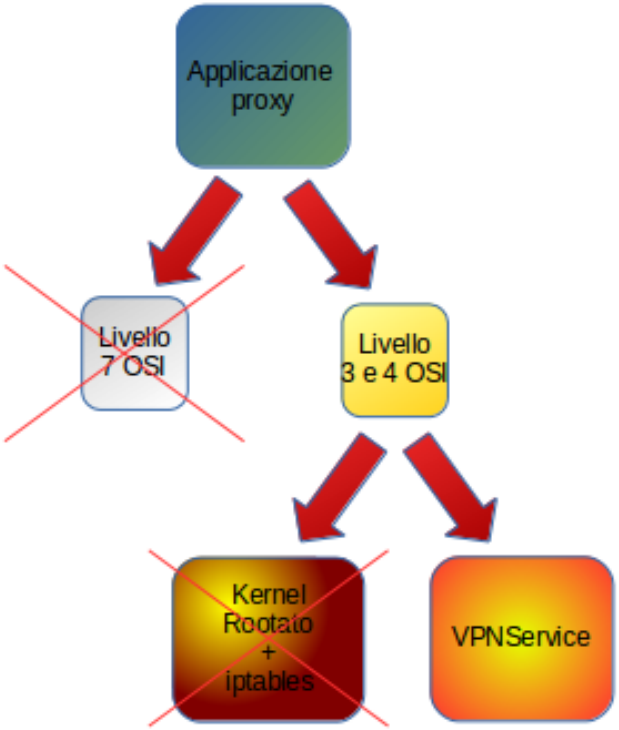


Figura 1.5: Schema concettuale riassuntivo delle scelte effettuate per l'implementazione della specifica di progetto

Capitolo 2

Le scelte implementative del Proxy

Si passa d'ora in avanti ad analizzare i dettagli implementativi del lavoro svolto. I moduli software concettualmente previsti in precedenza sono stati modellati in uno o più file combinati per fornire in armonia una data funzionalità. Per meglio comprendere il funzionamento delle classi realizzate si è deciso inoltre di fare dei richiami teorici ove necessario.

2.1 Premessa

La concezione ad alto livello dei moduli software da realizzare è stata effettuata nel precedente capitolo, in cui si è anche motivata la scelta di lavorare a livello 3 e 4 della pila OSI mediante l'utilizzo di VPNService. Uno specifico modulo, come detto, gestirà la comunicazione tra l'interfaccia virtuale di tipo TUN ed il dispositivo interno; un altro modulo software gestirà l'interazione tra il modulo software proxy e la rete; infine, un ulteriore modulo gestirà il monitoring. La logica con cui è strutturata l'applicazione, nelle sue classi che la compongono, è illustrata in figura 2.1. Tale logica sarà chiarita nelle sezioni successive.

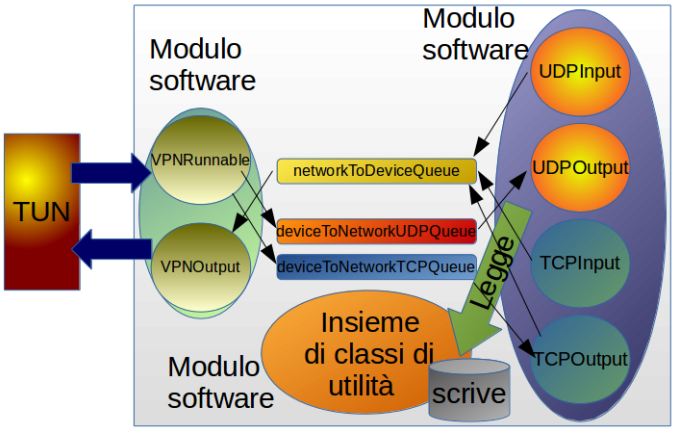


Figura 2.1: Schema logico del proxy realizzato e i moduli software che lo realizzano.

2.1.1 Le classi implementate e la gestione dei log

Il progetto realizzato si compone di un set di classi che si occupano di realizzare l'architettura necessaria al corretto funzionamento del modulo software proxy:

- ByteBufferPool;

- LocalVPN, che è la Main Activity;
- LocalVPNService;
- LRUCache;
- Packet;
- TCB;
- TCPInput;
- TCPOutput;
- UDB;
- UDPInput;
- UDPOutput;

e di un set di classi che si occupano di gestire il monitoring ed il logging delle informazioni interessanti di livello applicativo tra cui:

- Utils
- MyDB

Sono state realizzate inoltre due activity per permettere all'utente di scegliere quali app escludere dal meccanismo messo su per il monitoring:

- AllAppsActivity,per visualizzare e scegliere tra le app installate, quelle da escludere
- AppChoosedActivity, per visualizzare le app scelte per l'esclusione ed eventualmente resettare il set di quelle escluse

La memorizzazione delle App scelte per l'esclusione è realizzata mediante l'utilizzo della classe:

- PreferencesManager

Si è scelto di effettuare in due modalità differenti il logging dei pacchetti di livello applicativo e di livello 4 (TCP). In un log file creato nella cartella "Documents" del device Android si è scelto di riportare un log delle informazioni TCP, le quali riassumono lo scambio dati tra device e rete a livello 4, sotto forma di informazioni relative alla macchina a stati TCP. In un Database si è poi scelto di tenere memoria dei pacchetti di livello applicativo intercettati. Si noti che durante il funzionamento del modulo software proxy, i pacchetti vengono temporaneamente memorizzati in una lista, e solo nel momento in cui viene stoppata una **sessione** essi sono definitivamente scritti in memoria persistente. L'utente potrà quindi visionare i pacchetti loggati durante una sessione in una scrollview creata ad-hoc dopo aver stoppato la sessione di monitoring.
In **Utils** sono presenti inoltre metodi che permettono di fare il "discovering" della rete utilizzata (Mobile o WiFi), oltre ai metodi che si occupano di fare il "parsing" dei pacchetti di livello applicativo.

2.1.2 Le classi viste rispetto ai moduli software

Premettendo che la classe **LocalVPN** è la main activity, e ha il compito di avviare il service dell'applicazione, nel nostro caso **LocalVPNService**, sostanzialmente possiamo ripartire a livello concettuale le classi in tre macroaree individuabili in tre sotto-moduli software:

1. gestione della comunicazione con l' interfaccia TUN
2. gestione della comunicazione con la rete
3. gestione dell'attività di monitoring

2.1.3 Richiami teorici

Le classi implementate utilizzano concetti propri del linguaggio Java e nei prossimi paragrafi sarà necessario richiamare in breve il funzionamento di:

- **VpnService**
- **Java NIO Selector**
- **Java NIO Buffer**
- **LinkedHashMap**
- **Java NIO Datagram channel**
- **Java NIO Socket channel**

2.2 Utilizzo di VpnService

VpnService si presenta sotto forma di una classe base Java, che le applicazioni possono estendere per costruire le loro soluzioni VPN in un dispositivo Android. Si tratta di un meccanismo ad alto livello: operazioni, come la creazione di una "virtual network interface" e la configurazione di indirizzi e regole di routing, sono effettuate dal sistema operativo. Come anticipato, Android crea una interfaccia di tipo TUN [1] per la VPNService, e mette a disposizione un set di API che possono essere usate dall'applicazione per interagire con la stessa interfaccia TUN virtuale.

Per implementare ed interagire con l'interfaccia creata con VPNservice è buona norma seguire un'insieme generale di passi:

1. utilizzare un Builder per ottenere una interfaccia (fornita per mezzo di FileDescriptor) per il TUN. L' indirizzo IP virtuale dell'interfaccia, il DNS da utilizzare, e la tabella di routing possono essere configurati attraverso il Builder. In particolare ci si deve assicurare che il metodo **addRoute()** sia chiamato propriamente, così che la tabella di routing determini quali pacchetti saranno "routati" verso la TUN;

[1] Nelle reti Informatiche, TUN e TAP sono driver che permettono la creazione di periferiche di rete virtuali. Rispetto alle comuni periferiche (ad es. eth0) che sono controllate direttamente dalle schede di rete, i pacchetti spediti da o verso dispositivi TUN/TAP sono spediti da o verso programmi software. TUN è in grado di simulare una periferica di rete di tipo punto-punto e lavora con pacchetti di tipo IP mentre TAP è in grado di simulare un dispositivo Ethernet e logicamente utilizza i frame Ethernet.

2. ottenere uno stream di input e di output per prelevare e fornire pacchetti all'interfaccia virtuale. Gli stream di input e di output sono utilizzati per leggere e scrivere pacchetti in modalità "a stream";

3. effettuare una connessione al VPN server. Uno sviluppatore può realizzare ogni tipo di logica per gestire il pacchetto, di solito utilizzando una connessione a un tunnel verso il VPN server remoto. Come detto, nel progetto in esame quello che si vuole **non è realizzare una soluzione VPN** ma esclusivamente **sfruttare VPNService per dirottare i pacchetti** senza dover eseguire la procedura di rooting del dispositivo;

4. proteggere le socket dal VPNService attraverso il metodo **protect()**. Per evitare un loop infinito dei pacchetti (il pacchetto di una socket è comunque un pacchetto di rete,vse la socket non è protetta esso verrà dirottato di nuovo verso l'interfaccia virtuale e non uscirà mai sulla sottostante rete Internet, causando un loop infinito), la connessione deve essere protetta dalla VPN service;

5. gestire il routing dei pacchetti.

La struttura logica di funzionamento è perciò la seguente:

```
Network Activity -> TUN -> file descriptor "in" -> socket -> Remote Server
Network Activity <- TUN <- file descriptor "out" <- socket <- Remote Server
```

dove con TUN si intende l'interfaccia virtuale messa su da VPNService a cui vengono dirottati tutti i pacchetti di rete, con i file descriptor "in" e "out" si intendono le specifiche interfacce per l'input e l'output da e verso l'interfaccia virtuale completa.
Ci sono due metodi di grande importanza nella classe VPNService:

- **prepare(Context)**, di solito invocata dopo un'azione da parte dell'utente (e.g. pressione di un Button) ,procede al "kill" di una vecchia interfaccia virtuale VPNService precedentemente creata da un'altra applicazione. L' **intent** restituito assicura che ci sia solo una connessione VPN attiva. Se c'è già una connessione VPN **prepared**, allora sarà restituito il valore **null**. In questa eventualità, si dovrà invocare e gestire il metodo **onActivityResult** manualmente;

- **establish()**, che crea l'interfaccia virtuale utilizzando i parametri forniti al **VpnService.Builder**

Di seguito sono riassunti i punti generali per la creazione, la gestione e la distruzione di una interfaccia virtuale con VPNService:

- quando l'utente preme un bottone per autorizzare la connessione, viene chiamata la **prepare(Context)** e lanciata l'activity associata all'**intent** restituito, se non nullo; qui di seguito è illustrato un esempio di codice utilizzato per gestire l'intent restituito da una invocazione di prepare():

Codice 2.1: Corretto utilizzo della prepare()

```
public void onClick(View v) {
  Intent intent = VpnService.prepare(getApplicationContext());
  if (intent != null) {
       startActivityForResult(intent, 0);
  } else {
       onActivityResult(0, RESULT_OK, null);
  }
}
```

- quando l'applicazione diventa preparata, viene eseguita l'operazione di start del service chiamando il metodo **startService**, passando ad esso un intent, associato alla classe VPNService estesa in modo custom, come parametro.
- vengono forniti dei parametri specifici ad un **VpnService.Builder** e creata quindi l'interfaccia virtuale invocando il metodo **establish()**;
- vengono processati e scambiati pacchetti tra le socket e il descrittore di file restituito come interfaccia verso l'interfaccia virtuale;
- quando il metodo **onRevoke( )** viene richiamato, esso chiude il descrittore di file e rimuove l'interfaccia virtuale, ripristinando così il funzionamento originale della rete.

Riassumendo, ogni lettura dal descrittore recupera un pacchetto in uscita dal dispositivo, che è stato routato verso l'interfaccia virtuale; ogni scrittura sul descrittore inietta nell'interfaccia virtuale un pacchetto in entrata al dispositivo; sarà l'interfaccia virtuale a recapitare i pacchetti alle applicazioni corrispondenti, le quali non noteranno alcun tipo di differenza rispetto al caso in cui sia in piedi una connessione di rete normale. L'interfaccia virtuale, come già spiegato, opera al livello 3 della pila OSI, di conseguenza i pacchetti iniziano sempre con headers di livello IP.

Per quanto concerne il progetto realizzato, si ricorda che lo scopo è analizzare i pacchetti e effettuare il logging delle informazioni notevoli di livello applicativo. Tuttavia la TUN lavora al livello 3 dello stack OSI, di conseguenza ciò che si desidera leggere e scrivere sono pacchetti IP. Per quel che concerne l'implementazione, è notevole osservare che Android/Java non supporta le raw socket, di conseguenza non è possibile scrivere direttamente i pacchetti provenienti dalla rete esterna nell'interfaccia virtuale e, nel tragitto inverso, i pacchetti provenienti dall'interfaccia virtuale nelle socket aperte verso la rete. Nasce dunque la necessità di dover modificare le intestazioni. Poiché è possibile proteggere le socket dal routing verso l'interfaccia virtuale, è possibile usare socket protette per inviare o ricevere pacchetti da e verso la rete esterna. Dato che le socket TCP/UDP lavorano al livello 4 OSI, si può dunque operare nel seguente modo:

- prelievo del pacchetto IP dal'interfaccia TUN;
- estrazione delle informazioni di livello 4 sul tipo di protocollo (e.g. TCP/UDP) e relativo payload. Si noti che siccome è presente una procedura di handshake in TCP, prima di iniziare a scambiare dati e prelevare i payload, è necessario scrivere indietro prima il pacchetto di handshake.
- scelta della socket tramite la quale inviare il payload sulla rete. A questo step si opera a livello 4, dunque è necessario salvare le socket e cercare di ricevere successivamente tramite esse i dati di ritorno provenienti dalla rete esterna. Se si riceve un qualche dato di ritorno, ci sarà la necessità di passare questi pacchetti all'interfaccia TUN in modo opportuno;
- prelievo del pacchetto dalla socket, e costruzione di un pacchetto a livello 3. In precedenza però, è necessario costruire un valido pacchetto a livello 4:
  - UDP è un po' più semplice in quanto i 4 byte di intestazione di UDP contengono solo l' indirizzo sorgente, porta sorgente, indirizzo di destinazione, porta di destinazione.

  - TCP è più complesso in quanto si tratta di un protocollo che mantiene informazioni di stato sulla connessione, pertanto i numeri di sequenza e gli ack devono essere memorizzati ed impostati ogni volta in modo corretto.

- invio del pacchetto IP, ricevuto dalla rete attraverso le socket protette e opportunamente rimodellato, all'indietro verso l'interfaccia TUN, la quale provvederà a recapitare i pacchetti al legittimo richiedente originale (e.g. una qualsiasi app che genera traffico di rete, ad esempio un browser).

Le classi che estendono e specializzano la classe VPNService necessitano di essere dichiarati con un permesso appropriato, il permesso **BIND_VPN_SERVICE**. Qui un esempio di codice da inserire nel file **AndroidManifest.xml**:

Codice 2.2: Permessi necessari al corretto funzionamento di VPNService

```
...
 <service
 android:name=".ExampleVpnService"
        android:permission="android.permission.BIND_VPN_SERVICE">
    <intent-filter>
        <action android:name="android.net.VpnService"/>
    </intent-filter>
 </service>
...
```

2.3 Uso di Java NIO Selector

Un **Selettore** è un componente di Java NIO che può esaminare uno o più **canali** in modo non bloccante, e determinare quali canali siano pronti, ad esempio, per l'operazione di lettura o la scrittura. In questo modo un singolo thread può gestire più canali e, di conseguenza, più connessioni di rete.

2.3.1 Perché conviene usare un selettore

Il vantaggio di usare un singolo thread per gestire più canali risiede nel fatto che effettuare uno switch di diversi thread in esecuzione risulta essere costoso per un sistema operativo, ed ogni thread richiede comunque delle risorse dedicate. Da tali punti di vista meno thread si hanno in gioco e migliore sarà la situazione, sottolineando inoltre il fatto che si sta lavorando su un device mobile, che presenta **risorse e capacità limitate**.

La figura seguente schematizza l'utilizzo di un Selector.

2.3.2 Creazione di un Selector

Per creare un selettore si ricorre al metodo **Selector.open()** in questo modo:

Codice 2.3: Creazione di un Selector

```
Selector selector = Selector.open();
```

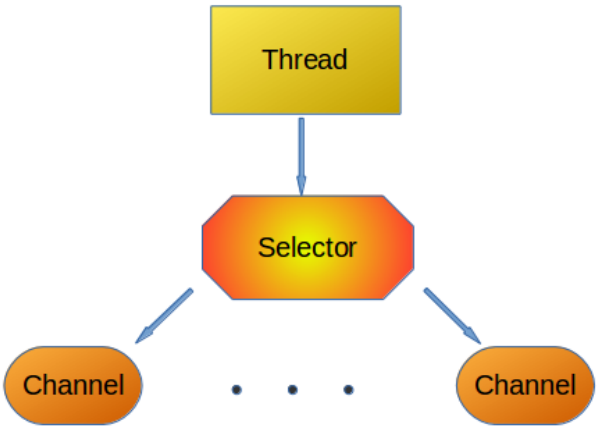


Figura 2.2: Schema concettuale di utilizzo di un Selector

2.3.3 Registrazione dei Channels con il Selector

Per poter monitorare un Channel mediante un Selector, lo stesso Channel va registrato. Questa operazione è eseguita attraverso il metodo **SelectableChannel.register()**, come segue:

Codice 2.4: Registrazione di un canale di comunicazione non bloccante presso un Selector, con interesse a monitorare il canale se pronto in lettura

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

Il Channel deve essere impostato in modalità **non-blocking** per poter essere utilizzato con un Selector, cioè in modalità non bloccante. Questo significa ad esempio che non si può monitorare un FileChannel con un Selector, dal momento che il FileChannel non può essere switchato nella modalità **non-blocking**, mentre i SocketChannel, interessanti per gli scopi di questo progetto,possono esserlo, motivo per cui sarà possibile registrarli e gestirli mediante un selettore. Il secondo parametro del metodo **register()** è un "interest set", che ha il significato relativo di "a quali eventi" si è interessati nell'ascolto di un dato Channel attraverso il Selector. Ci sono 4 differenti eventi di cui si può essere in ascolto in tal senso, ciascuno rappresentata da una macro **SelectionKey**:

- **Connect**, rappresentata da **SelectionKey.OP_CONNECT**;
- **Accept**, rappresentata da**SelectionKey.OP_ACCEPT**;
- **Read**, rappresentata da **SelectionKey.OP_READ**;
- **Write**, rappresentata da **SelectionKey.OP_WRITE**.

Se si è interessati a più di un evento basta ricorrere all'operazione logica di **OR** di tali costanti, ad esempio:

Codice 2.5: OR logico utilizzato nel caso in cui si è interessati a più di un evento

```
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

2.3.4 SelectionKey

Quando si registra un determinato Channel ad un Selector il metodo **register()** restituisce oggetti di tipo **SelectionKey**, che fornisce proprietà interessanti:

1. l' **interest set**: è il set di eventi a cui si è interessati nell'operazione di "selecting" di cui già si è discusso, può essere letto e scritto da **SelectionKey** ad esempio tramite i seguenti comandi:

Codice 2.6: I diversi tipi di operazione a cui si può essere interessati

```
int interestSet = selectionKey.interestOps();
boolean isInterestedInAccept = interestSet & SelectionKey.OP_ACCEPT;
boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT;
boolean isInterestedInRead = interestSet & SelectionKey.OP_READ;
boolean isInterestedInWrite = interestSet & SelectionKey.OP_WRITE;
```

Come si può notare, è possibile effettuare l'operazione logica di **AND** tra l' **interestSet** e la macro di **SelectionKey** per scoprire se un certo evento è contenuto nell'**interestSet** o meno.

2. il **ready set**: è il set di operazioni per cui un Channel è pronto. Si può accedere al **readySet** dal **SelectionKey** nel seguente modo:

Codice 2.7: Ottenere il set di operazioni per cui il canale è pronto

```
int readySet = selectionKey.readyOps();
```

E' possibile testare per quali operazioni o eventi il canale è pronto tramite le stesse modalità già viste per l'**interest set**, oppure utilizzando uno dei seguenti metodi, che restituiscono un valore booleano:

Codice 2.8: Testing dell'interest set per cui un canale è pronto

```
selectionKey.isAcceptable();
selectionKey.isConnectable();
selectionKey.isReadable();
selectionKey.isWritable();
```

3. **Channel**: è il canale associato al **SelectionKey**, accessibile come:

Codice 2.9: Ottenimento del canale associato a SelectionKey

```
Channel channel = selectionKey.channel();
```

4. **Selector**: è il selettore associato al **SelectionKey**, accessibile come:

Codice 2.10: Ottenimento del selettore associato a SelectionKey

```
Selector selector = selectionKey.selector();
```

5. **Oggetti associati**: E' possbile associare degli oggetti al **SelectionKey**, sia per riconoscere un canale sia per aggiungere ulteriori informazioni, ad esempio utilizzando il codice seguente:

Codice 2.11: Associazione di un oggetto al SelectionKey

```
selectionKey.attach(theObject);
```

ed è possibile ottenere tale oggetto associato al **selectionKey** nel modo seguente:

Codice 2.12: Ottenimento di un oggetto associato al SelectionKey

```
Object attachedObj = selectionKey.attachment();
```

E' possibile, inoltre, associare un oggetto già durante la registrazione del canale con il selettore, come parametro ulteriore del metodo **register**:

Codice 2.13: Associazione di un oggetto contestualmente alla registrazione

```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ,
    theObject);
```

2.3.5 Selezionare i Channels attraverso un Selector

Dopo aver registrato uno o più canali ad un **Selector**, si può ricorrere ai metodi **select()** per selezionarne uno specifico tra loro. Questi metodi restituiscono i canali che sono ready" per gli eventi a cui si è interessati: **connect, accept, read o write**:

- **int select()**: blocca finché almeno un canale è pronto per l'evento per cui è stato registrato;
- **int select(long timeout)**: stessa semantica della **select()**, con l'unica differenza che questa blocca per un massimo di **timeout** millisecondi.
- **int selectNow()**: è non bloccante, e restituisce immediatamente qualsiasi canali pronto al momento dell'invocazione.

Il valore **int** restituito dai metodi **select()** è indice del numero di canali pronti in accordo all'**interest set**.

2.3.6 selectedKeys()

Una volta che è stato utilizzato uno tra i metodi **select( )**, ed il suo valore di ritorno ha indicato che uno o più canali sono pronti, è possibile accedere ai canali pronti tramite il set di chiavi selezionate, invocando il metodo **selectedKeys()**. Inoltre è possibile iterare su questo set di chiavi attraverso un oggetto di tipo **Iterator<SelectionKey>**, e chiedersi per quale specificico evento sia pronto quel canale:

Codice 2.14: Controllo dell'evento per il quale un canale è pronto

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
while(keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.
    } else if (key.isConnectable()) {
        // a connection was established with a remote server.
    } else if (key.isReadable()) {
        // a channel is ready for reading
    } else if (key.isWritable()) {
        // a channel is ready for writing
    }
    keyIterator.remove();
}
```

E' da notare in questo snippet che la **keyIterator.remove()** è chiamata al termine di ogni iterazione. Il **Selector** non rimuove le istanze di **SelectionKey** dal set. Perciò questa operazione è da eseguire esplicitamente, nel momento in cui è stata soddisfatta la richiesta del canale. La prossima volta che quel canale sarà nuovamente pronto il **Selector** lo riaggiungerà nuovamente al set. Ovviamente il canale restituito dalla **SelectionKey.channel()** dovrà essere "castato" al tipo di canale con cui si intende lavorare (nel caso del progetto in esame si tratta di **SocketChannel** e **DatagramChannel**).

2.3.7 wakeUp()

Un thread bloccato su una invocazione di **select()**, può in qualche modo "lasciare" il metodo **select()** anche se nessun canale è dichiarato ready dal selector. Questa operazione è realizzata utilizzando un thread differente che invoca la **Selector.wakeup()** sullo stesso **Selector** su cui il primo thread ha precedentemente invocato una **select()**. A questo punto il thread in attesa bloccante sulla **select()** ritornerà immediatamente. Se un diverso thread invoca la **wakeup()** e nessun thread è bloccato sulla **select()**, il thread successivo che invocherà la **select()** "si sveglierà" immediatamente.

2.3.8 close()

Quando si è terminato il ciclo di utilizzo di un **Selector** è possibile invocare il metodo **close()**. Tale metodo chiude il **Selector** ed invalida tutte le istanze di **SelectionKey** registrate con il **Selector**. Si noti che i Channel registrati non vengono chiusi.

2.4 Uso di Java NIO Buffer

I Buffers sono utilizzati come appoggio nell'interazione con Channel di tipo NIO: i dati vengono letti dai Channel nei buffers, e vengono scritti dai buffers nei Channel .

2.4.1 Buffer

Un buffer è essenzialmente un **blocco di memoria**, nel quale è possibile scriverere dati, per poi poterli leggere in un secondo momento. Tale blocco di memoria è fornito da un oggetto di tipo NIO Buffer, che provvede inoltre a mettere a disposizione un set di metodi che rendono più semplice il lavoro con gli stessi blocchi di memoria.

2.4.2 Utilizzo generale di un Buffer

Utilizzando un buffer per leggere e scrivere dati tipicamente è preferibile seguire un processo costituito indicativamente da 4 step:

1. scrivere dei dati nell'oggetto buffer;
2. invocare la **buffer.flip()**;
3. leggere dei dati dal buffer;
4. invocare la **buffer.clear()** oppure la **buffer.compact()**;

Quando si scrivono i dati in un buffer, esso stesso tiene traccia di quanti dati vengono scritti. Nel momento in cui si vogliono leggere i dati, è necessario "flippare" il buffer dalla modalità di scrittura nella modalità di lettura attraverso il metodo **flip()**. Nella modalità di lettura il buffer permette di leggere tutti i dati scritti nel buffer.

Dopo aver letto tutti i dati, è necessario cancellare il buffer, in modo da renderlo pronto nuovamente per la scrittura. È possibile farlo attraverso due modalità: invocando **clear( )** oppure **compact()**. Il metodo **clear()** cancella l' intero buffer, mentre il metodo **compact()** rimuove dal buffer esclusivamente i dati letti. Ogni dato non letto viene spostato all'inizio del buffer e i dati saranno scritti nel buffer successivamente a quelli non letti.

2.4.3 Buffer Capacity, Position e Limit

Un buffer possiede tre proprietà con le quali è necessario avere dimestichezza, per comprendere la logica sottostante. Esse sono:

- **capacity**: essendo un blocco di memoria, un Buffer ha una certa dimensione fissata, detta "capacity". Quando esso diventerà pieno, sarà necessario svuotarlo, ad esempio facendo una **read** o una **clear** prima di poter scrivere ulteriormente su di esso;
- **position**:
  - nel caso in cui si effettui una **scrittura**, essa avverrà ad una specifica posizione. Inizialmente essa sarà pari a 0, tuttavia quando un byte sarà scritto nel Buffer, **position** viene avanzato per puntare alla cella successiva all'interno del buffer, e potrà al massimo raggiungere la sua **capacity** - 1.
  - nel caso in cui si effettui una **lettura**, essa potrà essere compiuta anche in una posizione specifica. Nel momento in cui viene operato un **flip()** ad un Buffer dalla modalità scrittura a quella di lettura, **position** è resettata a 0. Dopo aver letto il dato dal Buffer in una data posizione, **position** viene fatto avanzare automaticamente alla prossima posizione da leggere.
- **limit**:
  - in modalità di scrittura il **limit** di un Buffer è il limite della quantità di dati che si possono scrivere nel buffer. Perciò in modalità di scrittura limit sarà uguale alla **capacity** del Buffer.
  - quando si invoca un **flip()** dalla modalità di scrittura a quella di lettura, **limit** ha il significato di "quanti dati possono essere letti". E perciò, quando si fa un flipping in un Buffer in modalità lettura, **limit** è settato alla **position** della modalità scrittura. In altre parole, è possibile leggere tanti byte quanti sono stati scritti (il limite è impostato al numero di byte scritti, che è caratterizzato da **position**).

Il significato di **position** e **limit**, quindi, è diverso nei due casi in cui il buffer si trovi in modalità di lettura o scrittura, mentre per la **capacity** si ha sempre la stessa semantica.

2.4.4 Buffer types

Il Java NIO fornisce i seguenti tipi di buffer:

- ByteBuffer

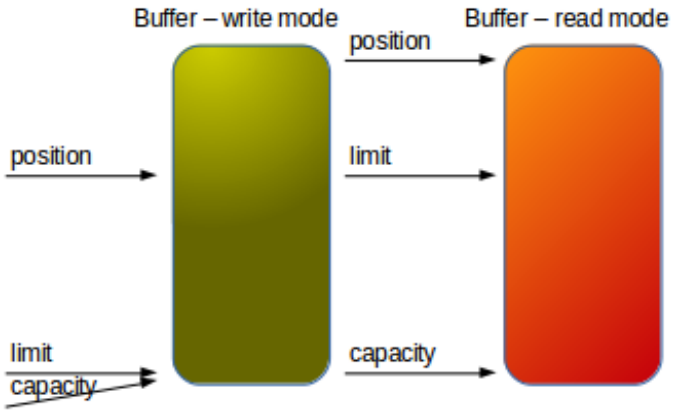


Figura 2.3: Schema riassuntivo dei marker di un buffer

- MappedByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

Queste tipologie differenti di Buffer rappresentano diversi tipi di dati, ma nel progetto svolto viene utilizzato unicamente il **ByteBuffer**, perciò ci si concentrerà unicamente su questo.

2.4.5 Allocare un Buffer

Per ottenere un oggetto Buffer si deve innanzitutto allocarlo ed ogni classe Buffer ha un metodo **allocate()** per eseguire tale operazione. Un esempio è il seguente:

Codice 2.15: Allocazione di un Buffer

```
ByteBuffer buf = ByteBuffer.allocate(48);
```

2.4.6 Scrivere dati su un Buffer

E' possibile scrivere dati in un Buffer in due modalità differenti:

- scrivendo dati da un Channel in un Buffer;
- scrivendo il dato in Buffer, attraverso il metodo **put()** di Buffer.

2.4.7 Metodo flip()

Il metodo **flip()**,come anticipato, fa da switch (per un Buffer) dalla modalità scrittura alla modalità lettura. Invocando **flip()** viene settata la **position** a 0, e viene settato inoltre il limite fissato in precedenza da **position**. In altre parole, **position** ora indica la posizione di lettura e **limit** segnala quanti byte sono stati scritti nel buffer.

2.4.8 Leggere dati da un buffer

Ci sono due modi per leggere dati da un buffer:

- leggere dati dal buffer nel Channel;
- leggere i dati dal buffer stesso, usando uno dei metodi **get()** forniti dal buffer.

Esistono molte versioni del metodo **get( )**, il quale consente di leggere i dati dal buffer in molti modi differenti.

2.4.9 Metodo rewind()

Il metodo **rewind()** setta semplicemente la **position** indietro a 0, in modo da poter rileggere tutti i dati nel buffer. Il **limit** rimane intatto, quindi segnala, ancora una volta, quanti elementi (byte) possono essere letti dal buffer.

2.4.10 I metodi clear() e compact()

Dopo aver letto i dati dal buffer, si deve rendere pronto nuovamente il buffer per la scrittura. È possibile farlo invocando i metodi:

- **clear()**: in questo caso **position** viene impostata a 0 e **limit** è impostata al valore di **capacity**. In altre parole, il Buffer viene cancellato. I dati nel buffer in realtà non vengono cancellati, ma i marcatori che indicano dove è possibile scrivere i dati nel buffer sono resettati. Se ci sono dati non letti nel buffer quando si invoca **clear()**, i dati saranno "dimenticati", il che significa che non esisteranno più eventuali marcatori che dicono quali dati sono stati letti, o ciò che non è stato letto.
- **compact ()**: se dovessero essere presenti dati ancora non letti dal buffer, e si desidera leggerli in un secondo momento (magari è necessario effettuare una operazione di scrittura urgentemente) si può invocare tale metodo che copia tutti i dati non letti fino all'inizio del buffer. Inoltre esso imposta **position** appena dopo l'ultimo elemento non letto. La **limit** è ancora impostata a **capaciy**, analogamente a **clear()**. Da questo momento il buffer è pronto per la scrittura, ma non sovrascrive i dati non letti.

2.5 La classe ByteBufferPool

Per favorire una migliore gestione dei ByteBuffer, favorendo un riutilizzo dei buffer allocati (per una migliore gestione della memoria, attraverso un suo risparmio e riutilizzo), dei quali si è fissata la **BUFFER_SIZE** a 65535, senza doverli necessariamente allocare in continuazione, quello che si fa è definire una classe che faccia da pool per i ByteBuffer. Saranno presenti, in questa classe, un attributo statico di tipo **ConcurrentLinkedQueue<ByteBuffer>**, e tre metodi:
Il metodo **acquire()**,

Codice 2.16: Il metodo acquire()

```
...
public static ByteBuffer acquire()
{
       ByteBuffer buffer = pool.poll();
       if (buffer == null)
```

```
            buffer = ByteBuffer.allocateDirect(BUFFER_SIZE+20);
        return buffer;
}
...
```

in cui viene fatta una **poll()** dalla coda **pool**, ovvero viene preso il primo elemento in testa. Qualora dovesse essere vuota la coda, allora viene allocato il buffer di dimensione **BUFFER_SIZE**.
Il metodo **release()**,

Codice 2.17: Il metodo release()

```
...
public static void release(ByteBuffer buffer)
{
        buffer.clear();
        pool.offer(buffer);
}
...
```

in cui viene fatta una **clear()** sul buffer passato come argomento, per poi essere inserito nel **pool** come ultimo elemento in coda, attraverso il metodo **offer()**.
Il metodo **clear()**,

Codice 2.18: Il metodo clear()

```
...
public static void clear()
{
        pool.clear();
}
...
```

che esegue la **clear()** su tutta la coda **pool**. Nella figura seguente è illustrato uno schema di quanto discusso.

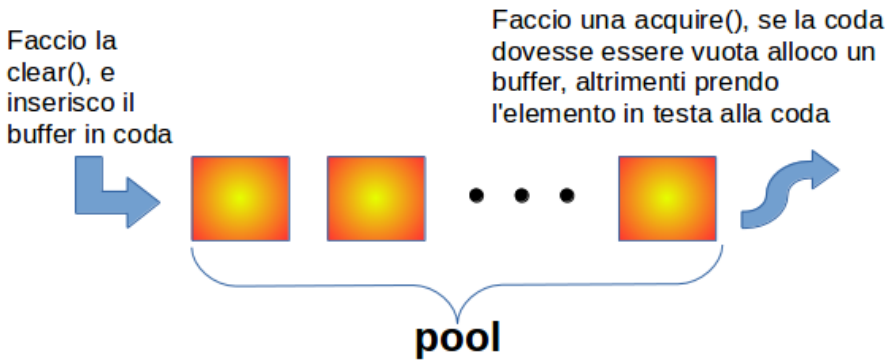


Figura 2.4: Schema riassuntivo del funzionamento del pool

2.6 L'Activity Principale: LocalVPN

LocalVPN è l'activity principale del modulo software proxy, realizzato su piattaforma Android. Essa si occupa di gestire le inizializzazioni del Database e del PreferencesMa-

nager, che successivamente verranno utilizzati, rispettivamente, per tenere traccia dei pacchetti intercettati durante le sessioni di monitoring e per mantenere in memoria la lista delle applicazioni che l'utente ha scelto di escludere dall'attività di monitoring tramite il modulo realizzato.
Si occupa inoltre di visualizzare a schermo la lista delle sessioni avviate dall'utente, in modo che quest'ultimo potrà scegliere una determinata sessione e visualizzarne i pacchetti intercettati durante l'attività di monitoring.
L'activity LocalVPN, infine, è fornita di tutta una serie di operazioni che le permettono di assolvere alla funzione di avviare correttamente il service necessario a metter su l'interfaccia virtuale, base su cui si poggia tutta l'attività di monitoring e di proxy richieste dalla specifica di progetto.

Codice 2.19: In questo snippet di codice è possibile notare il set di operazioni che LocalVPN esegue per assolvere correttamente al compito di avviare il service VPNService

```
...
private void startVPN() {
       Log.i(TAG, "startVPN");
       Utils.getLbqS().clear();
       Intent vpnIntent = VpnService.prepare(this);
       String time = Utils.DataAndTime();
       Utils.setTimeSession(time);
       sstart = new Session(lastSessionid+1,Utils.DataAndTime(),"me");
       arrayOfSession.add(sstart);
       adapter.notifyDataSetChanged();
       if (vpnIntent != null)
           startActivityForResult(vpnIntent, VPN_REQUEST_CODE);
       else {
           onActivityResult(VPN_REQUEST_CODE, RESULT_OK, null);
       }

   }
...
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
       Log.i(TAG, "OnActivityResult");
       super.onActivityResult(requestCode, resultCode, data);
       if (requestCode == VPN_REQUEST_CODE && resultCode == RESULT_OK) {
           waitingForVPNStart = true;
           localServiceIntent = new Intent(this, LocalVPNService.class);
           startService(localServiceIntent);

       }
   }
...
```

2.7 La classe VPNLocalService

Nell'applicazione sviluppata, questa è la classe che estende e specializza **VpnService**, creando così la già menzionata interfaccia virtuale TUN. L'indirizzo IP virtuale è stato settato ad un indirizzo casualmente scelto **VPN_ADDRESS =“10.0.0.2”**; assegnare un IP virtuale in questo modo implica il supporto univoco al protocollo IPv4. Poichè si vuole dirottare qualsiasi pacchetto verso l'interfaccia virtuale, bisogna settare una **VPN_ROUTE**

= "0.0.0.0" (con la semantica di "**tutte le interfacce** vengono intercettate e dirottate verso la TUN"), parametro da fornire al metodo già citato **addRoute()**. In questo service vengono definiti, tra i vari attributi, i due selettori **udpSelector** e **tcpSelector** e tre code, di tipo **ConcurrentLinkedQueue<Packet>**, che saranno impiegate per far interagire tra loro i vari thread in esecuzione. Nella **onCreate()** del service viene inizializzata la struttura della VPN, nonché i valori dei parametri chiave di cui già si è discusso. La **setupVPN()** viene invocata per inizializzare l'interfaccia TUN, utilizzando un **Builder()**. Per completezza si riporta il codice:

Codice 2.20: La setupVPN(), in cui si impostano i parametri e si mette su l'interfaccia virtuale

```
...
private void setupVPN() {
    if (vpnInterface == null) {
        Intent statusActivityIntent = new Intent(this, LocalVPN.class);
        pendingIntent = PendingIntent.getActivity(this, 0, statusActivityIntent,
            0);
        Builder builder = new Builder();
        builder.addAddress(VPN_ADDRESS, 32);
        builder.addRoute(VPN_ROUTE, 0);
        //builder.addDnsServer("8.8.8.8");
         ...
        vpnInterface = builder.setSession(getString(R.string.app_name))
         .setConfigureIntent(pendingIntent).establish();
    }
}
...
```

Si noti il comando commentato: esso è utilizzabile qualora si voglia impostare esplicitamente l'indirizzo IP di un DNS server per l'interfaccia virtuale. Ciò implicitamente suggerisce che l'interfaccia TUN messa su da VPNService gestisca da sè i pacchetti appartenenti al protocollo DNS. Questo è causa di alcuni problemi di funzionamento in specifiche versioni di Android, come si discuterà in seguito. Si precisa, infine, che nel caso in cui non venga invocato il metodo **addDnsServer()**, in automatico verrà utilizzato il DNS server impostato nelle configurazioni di rete del dispositivo Android (tipicamente 8.8.8.8, Google). Il metodo **setSession()** imposta un nome per la sessione, il quale verrà mostrato nella finestra di dialogo con l'utente. Il metodo **establish()**, infine, crea e restituisce il fileDescriptor utilizzabile come interfaccia verso la TUN appena inizializzata. Il fileDescriptor è impostato di default in modalità non bloccante, in modo da evitare blocchi causati da thread differenti.
Sempre nella **onCreate()** vengono avviati sei thread:

Codice 2.21: Nella sezione successiva è spiegato l'utilizzo dei sei thread

```
...
executorService.submit(new UDPInput(networkToDeviceQueue, udpSelector));
executorService.submit(new UDPOutput(deviceToNetworkUDPQueue, udpSelector,
    this));
executorService.submit(new TCPInput(networkToDeviceQueue, tcpSelector));
executorService.submit(new TCPOutput(deviceToNetworkTCPQueue,
    networkToDeviceQueue, tcpSelector, this));
executorService.submit(new VPNRunnable(vpnInterface.getFileDescriptor(),
    deviceToNetworkUDPQueue, deviceToNetworkTCPQueue, networkToDeviceQueue));
```

```
executorService.submit(new VPNOutput(vpnInterface.getFileDescriptor(),
    networkToDeviceQueue));
...
```

Un'ultima peculiarità da discutere è quella riguardante la lista delle applicazioni che l' utente ha scelto di escludere dall' attività di monitoring. L'interfaccia VPNService è in grado di distinguere due set di applicazioni: il set delle applicazioni "allowed" ed il set delle applicazioni "disallowed". Per applicazioni "allowed" si intende l'insieme di applicazioni a cui è permesso (anzi è obbligato) l'utilizzo dell'interfaccia virtuale; il set delle applicazioni "disallowed" è, al contrario, quello al quale non è permesso l'utilizzo dell'interfaccia virtuale (le quali quindi continueranno ad utilizzare la rete normalmente). Si precisa inoltre che quanto affermato **non significa che le applicazioni rilevino in qualche modo la presenza dell'interfaccia**, ma solo che **siano vincolate al passaggio tramite essa** nelle comunicazioni di rete. Avendo definito i due differenti set che l'interfaccia VPNService è in grado di distinguere, è opportuno menzionare due metodi che VPNService stesso mette a disposizione: **addAllowedApplication()** e **addDisallowedApplication()**. Questi due metodi si occupano di settare i due insiemi precedentemente descritti. E' opportuno precisare che basta effettuare una singola invocazione ad uno dei due metodi, in modo da definire uno solo dei due set menzionati; il set restante sarà riconoscibile semplicemente come il duale. Si noti infine che i due metodi di cui si è discusso accettano come parametro in ingresso il nome del package dell'applicazione da includere nel relativo set. E' necessario invocare quindi lo specifico metodo adatto per specificare uno dei due set una volta per ogni applicazione che si vuole inserire in quell'insieme.

2.7.1 Utilizzo di 6 thread

Si è scelto di utilizzare 6 diversi thread, in modo da massimizzare il parallelismo delle operazioni da svolgere per assolvere correttamente a tutte le funzionalità che si son volute offrire. Come detto precedentemente, si è scelto di supportare i protocolli a livello di trasporto **TCP** e **UDP**. I primi 4 dei suddetti 6 thread utilizzati si occupano pertanto, ciascuno rispettivamente, di gestire: l'input UDP (cioè i pacchetti UDP in ingresso dalla rete al dispositivo), l'output UDP (cioè i pacchetti UDP in uscita dal dispositivo,dirottati verso l'interfaccia virtuale e letti tramite il fileDescriptor associato), l'input TCP e l'output TCP, per cui vale un discorso analogo ai gestori di UDP. Gli ultimi due thread,infine, si occupano di gestire il prelievo ed il push di pacchetti opportuni da e verso l'interfaccia virtuale costruita mediante l'utilizzo di VPNService. Nel precedente snippet di codice, si è notato come, a livello puramente implementativo, vengano fornite delle code come parametri ad una serie di costruttori. Si è scelto infatti di realizzare i vari thread mediante un set di classi, ciascuna contenente un proprio metodo run, che l'executorService manderà in esecuzione. Le code fornite ai vari costruttori sono delle strutture dati utilizzate per mantenere traccia dei pacchetti in ingresso e in uscita, differenziando, anche qui, code utilizzabili per gestire UDP e code per gestire TCP. Si può notare un ulteriore parametro fornito ai vari costruttori, un Selector (ancora una volta differenti in base al protocollo di livello di trasporto di cui si occupano). Tali Selector si occupano di monitorare le socket aperte (e protette dall'interfaccia virtuale per evitare loop,come già spiegato) verso la rete esterna, in modo da poter leggere i pacchetti di risposta ove presenti, in modo non bloccante, e da poter scrivere i pacchetti ricevuti nelle code opportune, per poter essere successivamente gestite dagli altri thread in esecuzione.

2.7.2 VPNRunnable e VPNOutput

La classe VPNRunnable, la quale implementa Runnable e possiede un proprio metodo run(),ha il compito di controllare l'eventualità in cui siano presenti dei pacchetti di rete in uscita dal dispositivo verso la rete. Tali pacchetti, data la presenza dell'interfaccia virtuale,sono dirottati sempre verso di essa. La classe VPNRunnable ha pertanto il compito di recuperare i pacchetti dirottati verso l'interfaccia virtuale, riconoscerne la "natura di livello 4" (i.e. TCP o UDP), ed in base a tale natura fornire il pacchetto alla coda dedicata, che viene gestita in parallelo da un thread differente. Il costruttore di VPNRunnable prevede come parametri le tre code utilizzate per gestire i pacchetti in ingresso e in uscita dal dispositivo, ed un **FileDescriptor**, che rappresenta l'interfaccia verso la TUN. Si noti che all'interno di VPNRunnable viene aperto il **FileChannel** associato al **FileDescriptor** in lettura:

Codice 2.22: Ottenimento dell'interfaccia virtuale VPNService in lettura

```
...
FileChannel vpnInput = new FileInputStream(vpnFileDescriptor).getChannel();
...
```

Se viene letto un pacchetto dal **FileChannel**, allora viene creato un nuovo oggetto di tipo Packet in cui è inserito il pacchetto letto dall'interfaccia. A questo punto si effettua una discriminazione tra i due protocolli a livello di trasporto TCP e UDP, per l'inserimento sulla coda opportuna in uscita.

Codice 2.23: Discriminazione tra traffico TCP e UDP

```
...
if (readBytes > 0) {
dataSent = true;
bufferToNetwork.flip();
Packet packet = new Packet(bufferToNetwork);

      if (packet.isUDP()) {
          deviceToNetworkUDPQueue.offer(packet);
      } else if (packet.isTCP()) {
          deviceToNetworkTCPQueue.offer(packet);
      } else {
          KLog.w(TAG, "Unknown packet = " + packet.ip4Header.toString());
          dataSent = false;
      }
} else {
     dataSent = false;
}
...
```

In modo duale, la classe VPNOutput, la quale anch' essa implementa Runnable e possiede un proprio metodo run(), ha invece il compito di controllare l'eventualità in cui siano presenti dei pacchetti di rete in ingresso dalla rete al dispositivo. Si noti che all'interno di VPNRunnable viene aperto il **FileChannel** associato al **FileDescriptor** in scrittura:

Codice 2.24: Ottenimento dell'interfaccia virtuale VPNService in scrittura

```
...
```

```
FileChannel vpnOutput = new FileOutputStream(vpnFileDescriptor).getChannel();
...
```

La classe VPNOutput, dunque, ha il compito di prelevare pacchetti, ove presenti, dalla coda che gestisce i pacchetti in arrivo dalla rete verso il dispositivo; dopo aver prelevato con successo tali pacchetti, VPNOutput provvederà a scriverli sull'interfaccia virtuale, la quale consegnerà tali pacchetti alle applicazioni originariamente dirette destinatarie di essi. Si noti che, al momento del prelievo dalla coda **networkToDeviceQueue**, i pacchetti sono già stati opportunamente "modellati" dalle altre classi (thread) che si occupano di gestire i pacchetti proveniente dalla rete, motivo per cui può essere eseguita l'operazione di **write()** sull'interfaccia senza dover operare nuovamente su questi dati. In una situazione differente, si ricorda che sarebbe stato necessario operare sui pacchetti per due motivi:

- l'interfaccia virtuale opera al livello 3 della pila OSI, motivo per cui essa si aspetta di ricevere pacchetti di livello IP
- le socket aperte verso la rete esterna **non sono le stesse socket** che hanno aperto le applicazioni da cui hanno origine i pacchetti dirottati verso l'interfaccia. Per questo motivo i pacchetti di risposta che l'interfaccia deve inoltrare indietro alle applicazioni devono essere coerenti con i parametri delle socket possedute dalle applicazioni, in modo tale che ad esse sia **totalmente trasparente** la presenza di tali meccanismi.

Di seguito si riporta "il cuore" di VPNOutput, che implementa semplicemente quanto descritto in precedenza.

Codice 2.25: Scrittura sull'interfaccia virtuale dei pacchetti in ingresso dalla rete

```
...
bufferFromNetwork = networkToDeviceQueue.poll();
if (bufferFromNetwork != null) {
     bufferFromNetwork.flip();
    ...
     while (bufferFromNetwork.hasRemaining()) {
         vpnOutput.write(bufferFromNetwork);
     }
     dataReceived = true;
     ByteBufferPool.release(bufferFromNetwork);
} else {
     dataReceived = false;
}
...
```

2.8 La classe Packet

La classe Packet è una classe ad-hoc realizzata per rappresentare un pacchetto di livello IP e fornire un set di metodi per automatizzarne la gestione. Come infatti ampliamente già discusso, l'interfaccia virtuale messa su tramite VPNService opera a livello IP, perciò ci costringe a manipolare e a ricostruire i pacchetti appositi da poter recapitare a tale interfaccia. Ogni campo del pacchetto IP (Figura 2.5) assume pertanto una precisa importanza: ad esempio è proprio attraverso uno dei suoi campi che si discrimina sul protocollo di trasporto tra UDP, TCP, o qualunque altro protocollo (non supportato però da Proxy In The Middle, come illustrato in 4.6 dove si fa un ping, che usa ICMP). Risulta pertanto

di primaria importanza sia conservare le informazioni relative all'header IP, sia quelle relative all'header TCP e UDP. Inoltre è necessario avere a disposizione dei metodi per il corretto calcolo dei checksum sia a livello 3 che a livello 4 (in Figura 2.6 sono mostrate le strutture dei pacchetti TCP e UDP, dove è possibile notare come ognuno di essi presenti un campo "checksum").

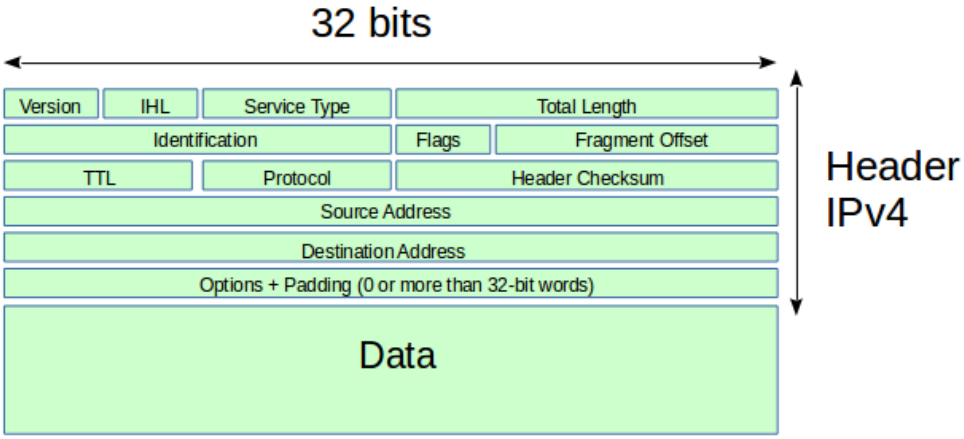
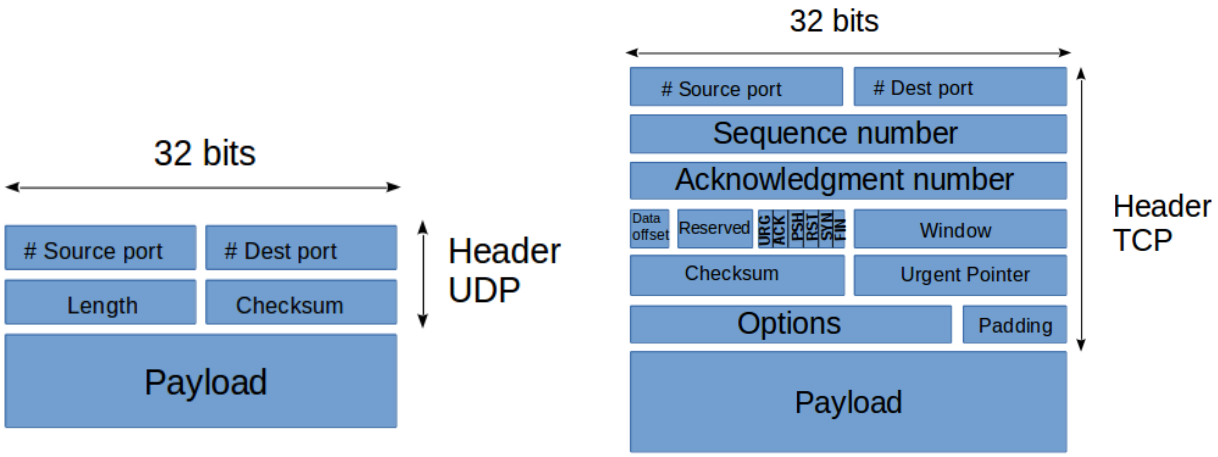


Figura 2.5: Datagramma IPv4



(a) *Segmento UDP*.

(b) *Segmento TCP*.

Figura 2.6: Struttura dei pacchetti UDP e TCP, di livello di trasporto

2.9 Il tracciamento delle connessioni

Dopo aver correttamente analizzato i pacchetti uscenti dal dispositivo e dirottati verso l'interfaccia, è stato necessario dunque aprire delle socket (protette) verso la rete esterna, per poter dare luogo alla comunicazione effettiva con il server (o peer eventualmente) legittimo destinatario del pacchetto originario e poter ricevere da essi delle risposte da inoltrare indietro alle applicazioni richiedenti un dato servizio. Per ottimizzare le prestazioni si è resa necessaria l'implementazione di una politica di **caching delle connessioni** (e quindi delle socket), in modo da poter avere a disposizione in breve tempo le socket attive e poter interagire con esse. Si è scelto di implementare un caching basato sulla politica Least Recently Used, con la quale si assume che le socket più recentemente utilizzate siano attive con probabilità più elevata negli intervalli temporali successivi.

2.9.1 La classe LinkedHashMap

La classe **LinkedHashMap** è una classe che estende **HashMap** e implementa l'interfaccia **Map**. Questa implementazione differisce dalla classe **HashMap** per il fatto che essa mantiene una lista doppiamente collegata. Tale lista definisce l'ordine di iterazione, che di solito è l'ordine nelle quale le chiavi sono state inserite nella mappa (insertion-order). E' bene notare che l'ordine di inserzione non cambia se una chiave è reinserita nella mappa (Una chiave k è reinserita nella mappa *m* se ,ad esempio, **m.put(k, v)** è invocato quando **m.containsKey(k)** restituisce **true**). Questa implementazione in un certo senso "salva" l'utilizzatore dall'ordine caotico a cui è sottoposto **HashMap**, e dall'aumento dei costi di gestione associato a **TreeMap**. E' anche previsto uno speciale costruttore per creare una **LinkedHashMap** in cui l'ordine di iterazione corrisponda all'ordine in cui le sue voci **sono state accedute**, dalla meno recente alla più recente (access-order), che è la feature interessante per l'implementazione utilizzata in questo progetto per la creazione della classe LRUCache. Detto costruttore ha la seguente segnatura:

Codice 2.26: Costruttore della classe LinkedHashMap

```
public LinkedHashMap (int initialCapacity, float loadFactor, boolean accessOrder)
```

in cui i parametri rappresentano, rispettivamente:

- initialCapacity, la capacità iniziale della hash map;
- loadFactor, il fattore di carico iniziale;
- accessOrder:
  - **true**, se l'ordine deve essere scelto in base all'ultimo accesso (dalla entry usata meno di recente a quella usata da più di recente);
  - **false**, se l'ordine dovrebbe essere quello in qui le "entries" sono state inserite.

2.9.2 La classe LRUCache

E' la classe implementata che realizza la Cache secondo l'algoritmo LRU[2], estendendo **LinkedHashMap**. Questa classe presenta due attributi importanti, **maxsize** e **callback**, ed è fornita di un costruttore:

Codice 2.27: Costruttore della classe LRUCache

```
...
public LRUCache(int maxSize, CleanupCallback callback)
{
    super(maxSize + 1, 1, true);
    this.maxSize = maxSize;
    this.callback = callback;
}
...
```

[2]Least Recently Used (LRU) è un algoritmo che nella cache scarta prima gli elementi meno utilizzati di recente. Questo algoritmo richiede tenere traccia di quando qualcosa è stata utilizzata , che è una operazione costosa se ci si vuole assicurare che l'algoritmo scarti sempre l' elemento meno utilizzato di recente. Implementazioni generali di questa tecnica richiedono il mantenenimento di "bit di età" per le linee della cache e regolarsi per applicare LRU in base a tale valore. In tale implementazione, ogni volta che la riga di una cache viene utilizzata, l'età di tutte le altre righe della cache cambia.

Si noti che nel costruttore e è richiamato il costruttore della classe estesa, **LinkedHashMap**, e come discusso il valore true passato come terzo parametro consente di settare la politica LRU voluta. Il metodo:

Codice 2.28: Metodo garbage collector della Cache LRU

```
...
    @Override
    protected boolean removeEldestEntry(Entry<K, V> eldest)
    {
        if (callback.canCleanup(eldest)) {

            callback.cleanup(eldest);
            return true;
        }

        return false;
    }
...
```

è utilizzato per rimuovere le entry più anziane (i.e. meno recentemente utilizzate) dalla Cache stessa.

2.10 Gestione del protocollo UDP

Nelle prossime sezioni si procederà a spiegare in che modo si è gestito ogni specifico protocollo di livello di trasporto. Sarà possibile notare, come anticipato, che la gestione del protocollo TCP è molto più complessa rispetto a quella di UDP, poichè quest'ultimo non mantiene informazioni sullo stato della connessione. Il protocollo UDP prevede un formato dei pacchetti molto ridotto rispetto al corrispettivo formato dei pacchetti TCP. Per gestire le socket in modo non bloccante, in Java (–>Android) è necessario l'utilizzo dei cosiddetti Channel. In particolare esistono due particolari versioni:

- **DatagramChannel**, utilizzata per avere a disposizione un canale specifico per UDP;
- **SocketChannel**, utilizzato per avere a disposizione un canale per TCP.

2.10.1 Utilizzo di Java NIO DatagramChannel

DatagramChannel è un canale che può inviare e ricevere segmenti UDP. Dal momento che UDP è privo di connessione, non è possibile utilizzare di default le modalità con cui normalmente si legge e si scrive sugli altri canali. Quello che si fa è, quindi, inviare e ricevere pacchetti di dati. E' possibile aprire un **DatagramChannel**, ed effetuarne il bind su una specifica porta in questo modo:

Codice 2.29: Apertura e bind di un DatagramChannel

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(NUMBER_PORT_S));
```

dove **NUMBER_PORT_S** è il numero di porta sorgente specifica.
Per ricevere dati attraverso un DatagramChannel è necessario invocare la **receive()**, allocando preventivamente un buffer di memoria, e invocando una **clear()** su di esso per resettarne i marker:

Codice 2.30: Ricezione dati su un DatagramChannel

```
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
channel.receive(buf);
```

Il metodo **receive()** effettua una copia del contenuto del pacchetto di dati ricevuto nel buffer passato come parametro. Se il pacchetto contenuto contiene più dati di quelli che il buffer può contenere, allora tali dati saranno scartati.
L'invio di dati attraverso un DatagramChannel avviene invocando il metodo **sent()**, nel modo seguente:

Codice 2.31: Invio dati su un DatagramChannel

```
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();
int bytesSent = channel.send(buf, new InetSocketAddress(URL_D, NUMBER_PORT_D));
```

Non ci si aspetta nessuna notifica di ricezione (i.e. ACK), dal momento che UDP è un protocollo non orientato alla connessione.
E' possibile tuttavia effettuare non solo l'invio, ma anche una vera e propria connessione ad uno specifico indirizzo sulla rete. Dal momento che però UDP non è orientato alla connessione, non viene creata una reale connessione, come accade con un canale TCP. Ciò che accade, piuttosto, è che il **DatagramChannel** è bloccato in modo che l'effetto finale è che si può solo inviare e ricevere pacchetti di dati da un indirizzo specifico.

Codice 2.32: Connessione ad un host remoto

```
channel.connect(new InetSocketAddress(URL_D, NUMBER_PORT_D));
```

Una volta connessi è possibile ricorrere ai metodi **read()** e **write**, come si fa per i canali tradizionali. Si ricorda che con UDP non c'è alcuna garanzia di consegna dei dati.

Codice 2.33: Scrittura e lettura sul DatagramChannel

```
int bytesRead = channel.read(buf);
int bytesWritten = channel.write(buf);
```

2.10.2 La classe UDB

La classe UDB è una classe che estende il concetto di TCB [3] (Transmission Control Block) al caso UDP, per gestire con maggiore efficienza questo caso. L'idea è quella di costruire una tabella di associazioni (Cache di tipo LRU) in cui ogni entry della tabella UDB sia associata ad un singolo canale di tipo DatagramChannel che si andrà ad aprire verso una certa destinazione. La classe, per comodità, presenta un set di attributi tutti aventi il modificatore **public**.

La cache **udbCache**, **statica** e di tipo **LRUCache()**, avrà due parametri:

- **MAX_CACHE_SIZE**, che rappresenta la grandezza della cache;

[3] TCP adotta una struttura complessa di dati nota come Transmission Control Block (TCB). Essa mantiene le informazioni sui numeri di socket locali e remoti, l' invio e la ricezione dei valori buffer, di sicurezza e di priorità, e il segmento corrente nella coda. Il Transmission Control Block (TCB) gestisce anche l'invio e la ricezione dei numeri di sequenza.

- **new LRUCache.CleanupCallback<String, UDB>()** che rappresenta il **callback**.

Poiché quest'ultimo parametro è una interfaccia, i cui metodi, **canCleanup()** e **cleanup()**, sono richiamati nel metodo **removeEldestEntry()** di **LRUCache()**, allora è possibile implementarli e definire nella classe **UDB**:

- **MAX_WAIT_DATA_TIME** con cui è definito il discostamento temporale massimo per poter dichiarare se una entry è cancellabile o meno;
- la chiusura del Channel associato alla entry stessa, in **cleanup()** (ovvero l'attributo **channel** di tipo **DatagramChannel** della entry **eldest**).

Una LRUCache è una mappa dove ogni entry presenta una chiave K e un valore V, ed in questo caso specifico:

- K è il parametro di tipo **String** passato, ovvero l'attributo **ipAndPort** di **UDB**,che memorizza l'indirizzo e la porta UDP dell'interlocutore con cui sta comunicando un certo DatagramChannel;
- V è una entry di **UDB**.

Sono presenti poi i metodi "di classe" (statici) di UDB:

- **getUDB()**, per ottenere la entry **UDB** di indice **ipAndPort**;
- **putUDB()**, per inserire l'oggetto **udb** di tipo **UDB** di indice **ipAndPort**.

Ogni entry UDB possiede anche un attributo, **lastDataExTime**, che inizialmente rappresenta l'istante della sua creazione, e che successivamente, qualora venisse acceduto, viene aggiornato tramite la **refreshDataEXTime()**, ulteriore metodo di cui è fornita la classe **UDB**.

Gli attributi **referencePacket**, **readlen** e **writelen** sono utili per l'uso che si farà di **UDB** in **UDPInput** e **UDPOutput**.

Il metodo **closeUDB(UDB udb)** chiude il **DatagramChannel** associato alla entry **udb** e rimuove **udb** dalla Cache **udbCache**.

Il metodo **closeAll()** chiude tutti i canali e rimuove tutte le entries dalla tabella.

Riassumendo, la tabella UDB utilizzata per i canali di comunicazione UDP è implementata unicamente con lo scopo di mantenere in una Cache i canali più recentemente utilizzati. Nel caso di TCP, invece, una tabella analoga sarà imprescindibile, poichè è necessario memorizzare i dati relativi ai numeri di riscontro e di sequenza che tale protocollo a livello di trasporto prevede per mantenere una connessione "stateful".

2.10.3 La classe UDPoutput

E' la classe (il thread) che si occupa di gestire il traffico UDP in uscita dal dispositivo verso la rete esterna. Il costruttore di tale classe accetta come parametri di ingresso il selettore udp, la coda contenente i pacchetti UDP uscenti dal dispositivo verso la rete e un riferimento alla VPNService, il quale servirà per proteggere le socket aperte verso il mondo esterno dal dirottamento verso l'interfaccia virtuale (con conseguente effetto loop già discusso). Dopo aver prelevato un pacchetto UDP, in uscita dal dispositivo, dalla coda apposita tramite l'istruzione :

Codice 2.34: Prelievo di un pacchetto UDP in uscita dal dispositivo

```
...
Packet currentPacket;
...
currentPacket = inputQueue.poll();
...
```

si procede a prelevare il payload dal pacchetto:

Codice 2.35: Prelievo del payload di un pacchetto UDP in uscita dal dispositivo

```
...
ByteBuffer payloadBuffer = currentPacket.backingBuffer;
...
```

Dall'header IP del pacchetto vengono prelevate e salvate le informazioni relative all'indirizzo di destinazione e alle porte sorgente e destinazione:

Codice 2.36: Salvataggio delle informazioni

```
...
InetAddress destinationAddress = currentPacket.ip4Header.destinationAddress;
int destinationPort = currentPacket.udpHeader.destinationPort;
int sourcePort = currentPacket.udpHeader.sourcePort;
String ipAndPort = destinationAddress.getHostAddress() +
":" + destinationPort + ":" + sourcePort;
...
```

Si è gia discusso il fatto che per tenere traccia del traffico di ritorno dalla rete al dispositivo, è necessario salvare in una Cache (UDB) i canali aperti. Poiché si è scelto di differenziare i canali mediante una stringa che mantenga i valori di indirizzo IP destinazione e delle porte sorgente e destinazione, per ogni pacchetto in uscita dal dispositivo si verifica se esiste un canale già aperto verso stessa destinazione e porta UDP. Se esiste si prosegue ad inviare il pacchetto sullo stesso canale, altrimenti viene aperto un nuovo canale,proteggendolo opportunamente dall'interfaccia virtuale e registrandolo sul selettore udp, in attesa di una risposta dal server remoto (o da un peer):

Codice 2.37: Invio dei dati UDP attraverso un DatagramChannel

```
...
udb = UDB.getUDB(ipAndPort);
if (udb == null) {
     outputChannel = DatagramChannel.open();
     vpnService.protect(outputChannel.socket());

     try
     {
        outputChannel.socket().setReceiveBufferSize(65535);
        outputChannel.socket().setSendBufferSize(65535);
        outputChannel.connect(new InetSocketAddress(destinationAddress,
            destinationPort));
     }
     catch (IOException e)
     {
        e.printStackTrace();
```

```
            closeChannel(outputChannel);
            ByteBufferPool.release(payloadBuffer);
            continue;
        }

        if (!outputChannel.isConnected()){
            Utils.addElementtoLbqTL("-OUT: " + ipAndPort + " isConnected fail!!!");
            closeChannel(outputChannel);
            ByteBufferPool.release(payloadBuffer);
            continue;
        }

        currentPacket.swapSourceAndDestination();
        udb = new UDB(ipAndPort, outputChannel, currentPacket);
        UDB.putUDB(ipAndPort, udb);
        outputChannel.configureBlocking(false);
        selector.wakeup();
         outputChannel.register(selector, SelectionKey.OP_READ, udb);
}
...
```

A questo punto si procede ad inviare i pacchetti sul canale di comunicazione aperto:

Codice 2.38: Invio dei dati UDP attraverso un DatagramChannel

```
...
outputChannel = udb.channel;
Utils.parseUDPOutputApplicationProtocol(payloadBuffer, udb.referencePacket,
    context);
try
{
    while (payloadBuffer.hasRemaining())
           outputChannel.write(payloadBuffer)
}
...
```

Si noti che è invocata l'operazione di **write()** fornendo come parametro un pacchetto di livello applicativo: **payloadBuffer** infatti rappresenta il payload del pacchetto di livello 3 prelevato inizialmente dalla coda. Tutto ciò è corretto, in quanto una socket (TCP o UDP che sia) si aspetta di ricevere dati di livello applicativo da inviare ad un computer system remoto, sollevando lo sviluppatore dall'onere di dover calcolare ogni volta il checksum e gli altri parametri dell'header di livello 4.

2.10.4 La classe UDPinput

E' la classe (il thread) che si occupa di gestire il traffico UDP in ingresso dalla rete esterna verso il dispositivo. Viene avviato da **LocalVpnService**, e prende come parametri il selettore **udpSelector** e la coda **networkToDeviceQueue**. Il ciclo eseguito dal thread verifica la presenza di canali pronti per l'operazione di read. Se il numero di canali pronti è 0 viene addormentato il thread per un certo tempo, altrimenti vengono impostati:

```
...
 Set<SelectionKey> keys = selector.selectedKeys();
 Iterator<SelectionKey> keyIterator = keys.iterator();
```

...

dove **keys** è la collezione di chiavi pronte. Esistono infatti due set associati ad un oggetto di tipo Selector, uno di registrazione, che contiene tutti i channel registrati, e uno di chiavi selezionate, e su questo viene definito un iteratore per poterlo scorrere. Fino a quando il thread non viene interrotto e **keyIterator.hasNext()** è **true**, ovvero c'è almeno un elemento nel set di chiavi selezionate, soddisfatta l'elaborazione associata ad una chiave pronta, questa sarà rimossa dal set di chiavi selezionate. Successivamente, dovesse tornare ad essere pronta, sarà nuovamente inserita in tale set e nuovamente elaborata. L'elaborazione avviene dopo un controllo **key.isValid() && key.isReadable()**, e consiste in una lettura dall' **inputChannel** (che è di tipo **DatagramChannel** ed è il canale associato a **key**). Si acquisisce anche la **udb** di tipo **UDB** associato a **key**, e da quel canale di input, sincronizzandosi sull'accesso ad **udb**, si effettua un'operazione di lettura, salvando i byte letti in **receiveBuffer**, ovvero un buffer precedentemente predisposto a tale scopo. Infine si invoca **udb.refreshDataEXTime()**, necessaria per realizzare e mantenere l'algoritmo LRU scelto per la cache. Dalla **udb** si legge il vecchio **referencePacket** di tipo **Packet**, e si fa l' **updateUDPBuffer** per aggiornare quel pacchetto; passando il numero **readBytes** come parametro, si imposta la position del buffer ad **HEADER_SIZE + readBytes** (HEADER_SIZE è pari alla lunghezza tipica di un header UDP sommata alla lunghezza tipica di un header IP). Successivamente verrà eseguita una **outputQueue.offer(receiveBuffer)**, sulla coda **networkToDeviceQueue** , in modo da fornire a tale coda un pacchetto opportunamente costruito, che tramite la classe VPNOutput verrà fornito all'interfaccia virtuale. L'interfaccia virtuale a sua volta fornirà indietro alle applicazioni, legittime destinatarie, il pacchetto opportunamente forgiato.
Le applicazioni che generano e ricevono traffico di rete UDP pertanto, **vedono la rete come se non ci fosse alcuna interfaccia virtuale ed alcun meccanismo di sniffing e monitoring dei pacchetti attivo.**

2.11 Gestione del protocollo TCP

La gestione del protocollo TCP è molto più complessa rispetto a quella di UDP, poichè è necessario mantenere informazioni sullo stato della connessione. Inoltre è necessario utilizzare canali di comunicazione non bloccante di tipo SocketChannel,che rappresentano un'astrazione di un canale di comunicazione che sfrutta il protocollo TCP a livello di trasporto. Nelle sezioni seguenti si riassumeranno i concetti base del protocollo TCP ed i concetti base dei SocketChannel.

2.11.1 Utilizzo di Java NIO SocketChannel

Un SocketChannel è un canale che è connesso ad una socket TCP. Esistono due differenti situazioni per cui si rende necessario creare un **SocketChannel**:

- utilizzare **SocketChannel** per una connessione remota ad un server che supporta TCP (es. un server Web);
- una **SocketChannel** può essere creata nel momento in cui arriva una richiesta TCP ad una **ServerSocketChannel**[4]

In particolare per aprire una connessione è necessaria la seguente operazione:

[4]Una Java NIO **ServerSocketChannel** è un canale che è in ascolto per connessioni TCP in entrata.

Codice 2.39: Apertura e connessione di un SocketChannel ad un host remoto

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(new InetSocketAddress(URL_D, NUMBER_PORT_D));
```

Per chiudere un **SocketChannel**,invece:

Codice 2.40: Chiusura di un SocketChannel

```
socketChannel.close();
```

Per leggere da una **SocketChannel** quello che si farà è ricorrere all'operazione di **read()**:

Codice 2.41: Lettura da un SocketChannel

```
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = socketChannel.read(buf);
```

Per effettuare correttamente questa operazione viene allocato un buffer ed il dato letto dalla **SocketChannel** è scritto all'interno di tale **buffer**. Il numero intero restituito dall'operazione di lettura è pari al numero dei byte effettivamente letti, oppure -1 se è stata raggiunta la fine dello stream (ovvero la connessione è stata chiusa). Per scrivere su un **SocketChannel** è possibile eseguire le seguenti operazioni:

Codice 2.42: Scrittura su un SocketChannel

```
String newData = "string";
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();
while(buf.hasRemaining()) {
    channel.write(buf);
}
```

In questo snippet di esempio viene allocato un buffer; il buffer viene scritto, in particolare vi si scrivono i byte della stringa precedentemente dichiarata. Poiché la scrittura sul canale avviene leggendo il buffer, viene innanzitutto eseguito un **flip()** (per switchare da lettura a scrittura e per reimpostare la position), e viene dunque passato il buffer alla **write()**. Si noti che la **SocketChannel.write()** è invocata all'interno di un ciclo **while**. Esso ripeterà la **write()** fino a quando non si disponga più di byte del buffer da scrivere.

Come già detto in precedenza, può essere settata la **SocketChannel** in una modalità non-blocking, e questa è una feature offerta molto importante. Dal momento in cui ciò viene fatto, la **connect()** ,la **read()** e la **write()** saranno in modalità asincrona.

- **connect()**: in modalità asincrona potrebbe ritornare prima che una connessione è stabilita. Per determinare se la connessione è stata stabilita, è possibile chiamare **finishConnect()**, in questo modo:

Codice 2.43: Controllo di avvenuta connessione

```
socketChannel.configureBlocking(false);
socketChannel.connect(new InetSocketAddress(URL_D, NUMBER_PORT_D));
while(! socketChannel.finishConnect() ){
    //wait, or do something else...
}
```

- **write()**: in modalità asincrona la **write()** potrebbe ritornare senza aver finito di scrivere. Perciò è necessario richiamare la **write()** in un loop.

- **read()**: nella modalità asincrona potrebbe ritornare senza aver finito di leggere tutto. Perciò si dovrà fare molta attenzione al valore intero restituito, che ci dice proprio quanti bytes sono stati letti.

La modalità non-blocking di SocketChannel si accoppia molto bene con l'utilizzo di un **Selector**. Registrando uno o più SocketChannel ad un Selector, si sono già discussi i vantaggi sia in fase di lettura che in fase di scrittura.

2.11.2 Gestione della connessione TCP

Il protocollo TCP, a differenza di UDP, mantiene una connessione vera e propria, ed uno stato associato alla connessione. Nasce dunque la necessità di aprire e chiudere connessioni TCP e gestire gli stati associati alle varie connessioni.
Di seguito è presentato un esempio di instaurazione di una connessione TCP, per meglio fissare le idee su cosa sia essenziale tenere in considerazione per l'implementazione progettuale.
Se un processo in esecuzione su un host (client) vuole stabilire una connessione TCP con un processo in esecuzione su un altro host (server) i passi seguiti sono i seguenti:

- **passo 1:** *TCP lato client invia uno speciale segmento al TCP lato server. Questo segmento speciale non contiene dati a livello applicativo, ma uno dei bit dell'intestazione, il bit SYN, è posto ad 1. Per questo motivo tale segmento viene detto segmento SYN. Inoltre il client sceglie a caso un numero di sequenza iniziale (**client_isn**) e lo pone nel campo numero di sequenza del segmento SYN iniziale.*

- **passo 2:** *Quando il datagramma IP contenente il segmento TCP SYN arriva all'host server, il server estrae il segmento dal datagrammma, alloca i buffer e le variabili TCP necessarie alla connessione, e invia un segmento di connessione approvata al client TCP. Anche questo segmento non contiene dati a livello applicativo, ma nella sua intestazione vi sono tre informazioni importanti. Innanzitutto il bit SYN è posto ad 1. In secondo luogo, il campo ACK assume il valore **client_isn+1**. Infine, il server sceglie il proprio numero di sequenza iniziale (**server_isn**) e lo pone nel campo del numero di sequenza. Tale segmento di connessione approvata è detto talvolta **segmento SYNACK**.*

- **passo 3:** *Alla ricezione del segmento SYNACK, anche il client alloca variabili e buffer per la connessione. L'host client invia al server un altro segmento in risposta al segmento di connessione approvata al server. Tale operazione viene svolta dal client ponendo il valore **server_isn+1** nel campo ACK dell'intestazione del segmento TCP. Il bit SYN è posto a 0, dato che la connessione è stata stabilita. In questo terzo passo dell'handshake a tre vie il campo dati del segmento può contenerer informazioni che vanno dal client verso il server.*

Una volta completati questi tre passi, gli host client e server possono scambiarsi segmenti contenenti dati. In ciascuno di questi futuri segmenti, il bit SYN sarà posto a 0. Chiunque dei due partecipanti alla connessione può terminarla, e, in tal caso, le risorse negli host (ossia buffer e variabili), vengono deallocate. Un comando di chiusura forza l'invio di un segmento TCP speciale, che ha nell'intestazione il bit FIN con valore 1. Chi riceve questo segmento, invia un segmento di shutdown, sempre con il bit FIN uguale ad 1. A questo punto tutte le risorse risulteranno deallocate.

2.11.3 Gli stati TCP

Durante una connessione TCP, il protocollo in esecuzione sugli host passa attraverso vari "stati TCP".

Lato client

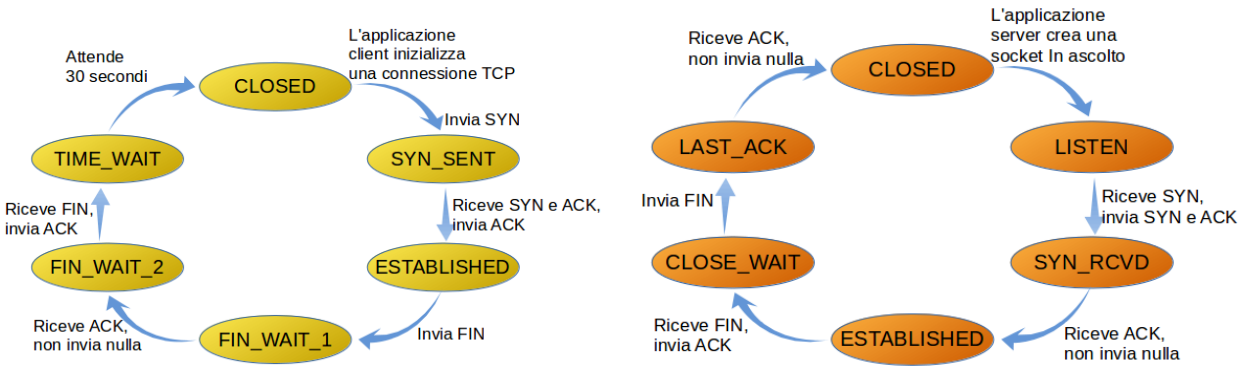
Il client TCP normalmente parte dallo stato **CLOSED**. Quando vuole inizializzare una connessione, il TCP client invia un messaggio SYN e lo stesso client entra nello stato **SYN_SENT**, durante il quale attende dal server TCP un segmento con un acknowledgment per un precedente segmento del client e che abbia il bit di SYN posto ad 1. Una volta ricevuto tale segmento, il client TCP entra nello stato **ESTABLISHED**, durante il quale può inviare e ricevere segmenti che contengono dati utili (ossia generati dall'applicazione). Supponendo che l'applicazione client decida di voler chiudere la connessione (anche il server potrebbe scegliere di terminarla), essa deve inviare un segmento TCP con il bit FIN impostato a 1 ed entrare nello stato **FIN_WAIT_1**. Quando si trova in questo stato, il client TCP attende dal server un segmento TCP contenente un acknowledgment e, nel momento in cui lo riceve, entra nello stato **FIN_WAIT_2**, in cui il client attende un altro segmento dal server con il bit impostato ad 1. Dopo aver ricevuto questo segmento, il client TCP manda un acknowledgment ed entra nello stato **TIME_WAIT**, che consente al client TCP di poter inviare nuovamente l'ultimo acknowledgment nel caso in cui l'ACK vada perduto. Il tempo trascorso nello stato **TIME_WAIT** normalmente dipende dall'implementazione, ma valori tipici sono normalmente 30 secondi, 1 o 2 minuti. Conclusa questa fase, la connessione viene completamente chiusa, e le risorse vengono completamente rilasciate, compresi i numeri di porta.

Lato server

Per il lato server, nel caso in cui il client chiuda la connessione, la situazione è speculare. L'applicazione server parte dallo stato di **CLOSED**, e necessita di mettersi in ascolto, creando una socket su una certa porta, ed entrando così nello stato di **LISTEN**. Quando riceve un segmento di SYN, invia il suo messaggio di SYN più l'ACK, per comunicare l'avvenuta ricezione e la comprensione della volontà del client di voler effettuare una connessione, entrando così nello stato **SYN_RCVD**. Ricevuto l'ACK dal client, inviato normalmente insieme ai dati, non invia nulla, ma passa allo stato **ESTABLISHED** (e qui termina la fase di handshake a tre vie). Quando poi il client comunica di voler terminare la connessione, il server riceve il messaggio di FIN e invia l'ACK. Allora il server passa nello stato di **CLOSE_WAIT**. Giunto in questo stato invia anche il secondo messaggio di FIN e passa allo stato di **LAST_ACK**. Infine, ricevuto l'ACK, non invia più nulla e torna allo stato di **CLOSED**.

Il segmento di reset

Quando un host TCP riceve richieste i cui numeri di porta o il cui indirizzo IP di origine non corrispondono a nessuna socket attiva sull'host , allora invierà al mittente un segmento speciale di reset, impostando il bit RST a 1, con lo scopo di comunicare alla sorgente: "Non ho una socket per quel segmento. Per favore non rimandarlo".



(a) *Stati lato client*. (b) *Stati lato server*.

Figura 2.7: Gli stati TCP lato client e lato server.

2.11.4 La classe TCB

La TCB (Transmission Control Block) è la struttura dati che si prefigge il compito di tenere traccia delle connessioni TCP. La classe, per comodità, presenta tutti attributi di tipo **public** (come **UDB**), nonché i metodi statici:

- **closeTCB(TCB tcb)**, che chiude il **DatagramChannel** associato a **tcb** e rimuove **tcb** dalla **tcbCache**.
- **closeAll()**,che chiude tutti i canali e rimuove tutte le entries nella tabella TCB.
- **getTCB()**, per prendere la entry **tcb** di indice **ipAndPort**;
- **putTCB()**, per inserire la entry **tcb** di tipo **TCB** di indice **ipAndPort**.

Ogni entry della tabella TCB possiede l' attributo **lastDataExTime**, che all'inizio presenta un tempo che indica l'istante temporale della sua creazione, e che successivamente,qualora venisse acceduto, viene aggiornato tramite la **refreshDataEXTime()**. Ogni entry possiede inoltre gli attributi **referencePacket**, **readlen** e **writelen**, utili per l'uso che si farà di **TCB** in **TCPInput** e **TCPOutput**.
TCB deve tenere conto dello stato della connessione tcp, e perciò, diversamente da **UDB**, si rende necessaria l'aggiunta di altri attributi:

- due attributi long **mySequenceNum**, **theirSequenceNum**, per avere una informazione precisa sui numeri di sequenza di entrambe le entità interagenti nello scambio dati TCP;
- due attributi long **myAcknowledgementNum**, **theirAcknowledgementNum**, per avere informazione relativa agli ACK;
- un attributo pubblico di tipo TCBStatus **status**, per tenere traccia dello stato della entry TCB, dove **TCBStatus** può essere:

Codice 2.44: Possibili stati di un host che esegue TCP

```
...
public enum TCBStatus {
        SYN_SENT,
        SYN_RECEIVED,
        ESTABLISHED,
        CLOSE_WAIT,
```

```
        LAST_ACK,
    }
...
```

Come per **udbCache** la **tcbCache**, **static** e di tipo **LRUCache()**, avrà due campi: **MAX_CACHE_SIZE**, e **new LRUCache.CleanupCallback<String, TCB>()** come **callback**. Poiché quest'ultima è una interfaccia, i cui metodi **canCleanup()** e **cleanup()**, sono richiamati nel metodo **removeEldestEntry()** di **LRUCache()**, è possibile ridefinire nella classe **TCB** i parametri con cui lavora **LRUCache()**, in particolar modo:

- il valore di **MAX_WAIT_ACK_TIME**, con cui è determinato il discostamento temporale necessario per poter dire se una entry è cancellabile o meno in **canCleanup()**, che però, a differenza di quanto visto in UDB, tiene conto anche di **TCBStatus**; infatti si dà la priorità nella cancellazione a quegli stati che ormai non hanno più nulla da dire alla connessione.

Codice 2.45: La canCleanup()

```
    ...
    public boolean canCleanup(Map.Entry<String, TCB> eldest) {
        boolean ret = false;
        TCBStatus status = eldest.getValue().status;
        if (status == TCBStatus.CLOSE_WAIT || status == TCBStatus.LAST_ACK)
            {
            ret = true;
            KLog.d(eldest.getKey() + " canCleanup st = " + status);
        } else if (System.currentTimeMillis() -
            eldest.getValue().lastDataExTime > MAX_WAIT_ACK_TIME) {
            ret = true;
            KLog.d(eldest.getKey() + " canCleanup lastDataExTime > " +
                MAX_WAIT_ACK_TIME);
        }

        return ret;
    }
    ...
```

TCB è perciò una LRUCache, cioè una "mappa" in cui ogni entry presenta una chiave K e un valore V, in particolare:

- K è il parametro di tipo **String** passato, ovvero l'attributo **ipAndPort** di **TCB**;
- V è l'oggetto **TCB**.

2.11.5 La classe TCPoutput

E' la classe (il thread) che si occupa di gestire il traffico TCP in uscita dal dispositivo verso la rete esterna. Il costruttore di tale classe accetta come parametri in ingresso il selettore tcp, la coda contenente i pacchetti TCP uscenti dal dispositivo verso la rete e un riferimento alla VPNService, il quale servirà per proteggere le socket aperte verso il mondo esterno dal dirottamento verso l'interfaccia virtuale (con conseguente effetto loop già discusso). Viene eseguito un loop, controllando se ci sono pacchetti da inviare verso la rete esterna:

Codice 2.46: Controllo e prelievo dei pacchetti TCP in uscita dal dispositivo

```
...
 do {
        currentPacket = inputQueue.poll();
        if (currentPacket != null)
                break;
        Thread.sleep(10);
      } while (!currentThread.isInterrupted());
...
```

Nel caso ci fosse un qualche pacchetto da inviare si esce dal **do/while**, si leggono i pacchetti IP dalla coda in uscita, e quello che si trasmette è ancora una volta il payload, mentre si usano le informazioni contenute nell'header per aprire e gestire i canali.

Codice 2.47: Prelievo del payload

```
...
ByteBuffer payloadBuffer = currentPacket.backingBuffer;
currentPacket.backingBuffer = null;
ByteBuffer responseBuffer = ByteBufferPool.acquire();
...
```

Dall'header di **currentPacket** vengono estratti l'indirizzo IP di destinazione e l'header tcp, da cui a sua volta vengono estratti porta di origine e di destinazione. Con il metodo **destinationAddress.getHostAddress()** si ricava l'indirizzo IP sorgente sotto forma di stringa, e si compone così la stringa **ipAndPort**, utilizzata come indice per la tabella TCB.

Codice 2.48: Costruzione della stringa ipAndPort

```
...
InetAddress destinationAddress = currentPacket.ip4Header.destinationAddress;
TCPHeader tcpHeader = currentPacket.tcpHeader;
int destinationPort = tcpHeader.destinationPort;
int sourcePort = tcpHeader.sourcePort;
String ipAndPort = destinationAddress.getHostAddress() + ":" +
      destinationPort + ":" + sourcePort;
...
```

In base a tale stringa, indice all'interno della cache TCB, ci si chiede se si sta accedendo a una entry che già esiste, o a qualcosa di nuovo. Nel primo caso si effettua un "refresh" del valore di ultimo accesso, e si guarda lo status per decidere le azioni da intraprendere. Nel secondo caso si invocherà una **initializeConnection()**.

Codice 2.49: Processing di un pacchetto TCP

```
...
TCB tcb = TCB.getTCB(ipAndPort);

if (tcb != null) {
    tcb.refreshDataEXTime();
}
if (tcb == null) {
    initializeConnection(ipAndPort, destinationAddress, destinationPort,
   currentPacket, tcpHeader, responseBuffer);
} else if (tcpHeader.isSYN()) {
```

```
    processDuplicateSYN(tcb, tcpHeader, responseBuffer);
} else if (tcpHeader.isRST()) {
    Utils.addElementtoLbqTL(tcb.ipAndPort + " isRST" + " st = " + tcb.status + "
        readLen = " + tcb.readlen);
    TCB.closeTCB(tcb);
} else if (tcpHeader.isFIN()) {
    processFIN(tcb, tcpHeader, responseBuffer);
} else if (tcpHeader.isACK()) {
    processACK(tcb, tcpHeader, payloadBuffer, responseBuffer);
} else {
    KLog.w("", "ipAndPort = " + ipAndPort + "->unknow type!!!");
}
...
```

Una volta che il payload è stato trasmesso, non ha più senso mantenerlo, e quindi si fa una **ByteBufferPool.release(payloadBuffer)**.

A questo punto si vuol spostare il focus sui quattro metodi che sono il cuore di TCPOutput: initializeConnection(), processDuplicateSYN(), processFIN() e processACK().

initializeConnection()

Questo metodo prende in ingresso i parametri:

- String **ipAndPort**;
- InetAddress **destinationAddress**;
- int **destinationPort**;
- Packet **currentPacket**;
- TCPHeader **tcpHeader**;
- ByteBuffer **responseBuffer**.

Si ricorda che **currentPacket** è il pacchetto letto dalla coda dei pacchetti TCP in uscita dal dispositivo verso la rete. Di questo pacchetto viene invertito l'indirizzo sorgente con quello di destinazione attraverso la **swapSourceAndDestination()**, metodo della classe **Packet**. Poi si controlla l'header tcp: se **tcpHeader.isSYN()** è **true**, allora vengono eseguite le restanti operazioni del metodo, altrimenti si esce. Nel caso in cui esso sia **true** dunque, è necessario aprire un canale di tipo**SocketChannel**, chiamato **outputChannel** e protetto dalla VPNService, e ovviamente creata una nuova **tcb** di tipo **TCB** per poter registrare e gestire la connessione tramite essa; tale entry viene poi aggiunta alla tcbCache.

Codice 2.50: Apertura di un SocketChannel e protezione di esso dall'interfaccia virtuale VPNService

```
...
SocketChannel outputChannel = SocketChannel.open();
vpnService.protect(outputChannel.socket());
outputChannel.configureBlocking(false);
```

Tramite il metodo **socket()** della classe **SocketChannel**, si ottiene l'oggetto ti tipo**Socket** relativo ad **outputChannel**, e vengono impostate la dimensione del buffer di invio **SO_RCVBUF**[5], e di ricezione **SO_SNDBUF** [6]:

Codice 2.51: Impostazione dei buffer

```
...
outputChannel.socket().setReceiveBufferSize(65535);
outputChannel.socket().setSendBufferSize(65535);
...
```

A questo punto viene effettuata la connessione della **SocketChannel** attraverso la **connect()** e viene impostato lo status di **tcb** a **SYN_SENT**. Siccome si vuole effettuare una registrazione del canale al **Selector**, ed esso potrebbe però essere bloccato sulla richiesta di una **select()** del thread **TCPInput**, per poter effettuare la registrazione (la cui operazione di interesse è **OP_CONNECT**, poichè si è interessati a stabilire la connessione), si ricorre all'invocazione del metodo **wakeup()** sul selettore.

Codice 2.52: Utilizzo della wakeup()

```
...
outputChannel.connect(new InetSocketAddress(destinationAddress,
    destinationPort));
tcb.status = TCBStatus.SYN_SENT;
selector.wakeup();
tcb.selectionKey = outputChannel.register(selector, SelectionKey.OP_CONNECT,
    tcb);
...
```

processDuplicateSYN()

Questo metodo viene eseguito se il **tcpHeader** contiene un bit **SYN** messo a 1. Nel caso lo status di **tcb** fosse **SYN_SENT** (ciò vuol dire che già un SYN è stato inviato), allora nel **tcb** viene unicamente aggiornato l'attributo **myAcknowledgementNum**, altrimenti viene invocato il metodo **sendRST()** che sostanzialmente segnala l'errore e chiude la connessione, inviando il segmento RST.

Codice 2.53: Operazioni eseguite nella processDuplicateSYN()

```
...
    if (tcb.status == TCBStatus.SYN_SENT) {
                tcb.myAcknowledgementNum = tcpHeader.sequenceNumber + 1;
                return;
    }
}
sendRST(tcb, 1, responseBuffer);
...
```

[5]La dimensione in bytes del buffer in ricezione della socket. Questo deve essere un intero più grande di 0. Questo è un suggerimento per il kernel ; il kernel può utilizzare un buffer più grande. Per le datagram sockets, i pacchetti più grandi di questo valore saranno scartati.

[6]La dimensione in bytes del buffer in invio della socket. Questo deve essere un intero più grande di 0. Questo è un suggerimento per il kernel; il kernel può utilizzare un buffer più grande. Per le datagram sockets, è definito dall'implementazione se i pacchetti più grandi di queste dimensioni possono essere inviati .

processFIN()

In questo metodo vengono aggiornati gli attibuti di tcb **myAcknowledgementNum**, che corrisponde all'ACK che ci si aspetta di riceve, e che è quello che si deve inoltrare verso il mittente, e **theirAcknowledgementNum**, che è il numero di ACK del segmento vero e proprio. Si imposta lo status di **tcb** a **LAST_ACK**, si setta il **responseBuffer** con i giusti campi, si aggiorna il numero di sequenza di **tcb**, e si invia al mittente tale buffer.

Codice 2.54: Operazioni rilevanti eseguite nella processDuplicateSYN()

```
...
Packet referencePacket = tcb.referencePacket;
tcb.myAcknowledgementNum = tcpHeader.sequenceNumber + 1;
tcb.theirAcknowledgementNum = tcpHeader.acknowledgementNumber;
if (true) {
    tcb.status = TCBStatus.LAST_ACK;
    referencePacket.updateTCPBuffer(responseBuffer, (byte) (TCPHeader.FIN |
        TCPHeader.ACK),
         tcb.mySequenceNum, tcb.myAcknowledgementNum, 0);
    tcb.mySequenceNum++;
    Utils.addElementtoLbqTL("-OUT: "+tcb.ipAndPort + " FIN netToDevice FIN|ACK");
    outputQueue.offer(responseBuffer);
    return;
   }
}
...
```

processACK

In questo metodo si preleva il canale associato a **tcb**, e si passa attraverso uno switch case che discrimina diversi stati. Se infatti dovesse arrivare dal mittente un segmento da inoltrare, che ha un header di tipo ACK, e lo status è:

- **SIN_SENT**: c'è un errore;
- **SIN_RECEIVED**: allora lo status è ora **ESTABLISHED**, perché si è ricevuto il SYN e l'ACK, e si sta inviando un ACK verso il channel;
- **LAST_ACK**, stato in cui si è giunti con il metodo **processFIN()**. In questo caso, si deve unicamente chiudere il canale.

Quando si è nello stato **ESTABLISHED** si controlla la variabile **payloadSize**; se risulta essere maggiore di zero si procede. Se **waitingForNetworkData** è **false** lo si imposta a **true** registrando presso il selettore il canale con operazione di interesse **OP_READ**. A quel punto viene invocata la **wakeup()** sul selettore, dopo di che si scrive il **payloadBuffer** all'interno del ciclo **while**. A questo punto, si aggiornano gli attributi del **tcb**:

- **myAcknowledgementNum** è il numero di ACK che ricevo in risposta, quindi il numero di sequenza più la lunghezza del payload trasmesso;
- **theirAcknowledgementNum** è il numero di ACK che viene inviato al server, perciò è già scritto nell'intestazione.

Il **referencePacket**, attributo di **tcb**, viene ora ricostruito e viene ricostruito il **responseBuffer** , pacchetto di risposta del server da inviare in risposta al client.

Codice 2.55: Ricostruzione di un pacchetto

```
...
tcb.myAcknowledgementNum = tcpHeader.sequenceNumber + payloadSize;
tcb.theirAcknowledgementNum = tcpHeader.acknowledgementNumber;
Packet referencePacket = tcb.referencePacket;
referencePacket.updateTCPBuffer(responseBuffer, (byte) TCPHeader.ACK,
tcb.mySequenceNum, tcb.myAcknowledgementNum, 0);
Utils.addElementtoLbqTL("-OUT: "+tcb.ipAndPort + " ACK netToDevice ACK st = " +
    tcb.status);
}
outputQueue.offer(responseBuffer);
}
...
```

2.11.6 La classe TCPinput

E' la classe (il thread) che si occupa di gestire il traffico TCP in ingresso dalla rete esterna verso il dispositivo. Viene avviato da **LocalVpnService**, e prende come parametri il selettore tcp e la coda **networkToDeviceQueue**. Anche qui, come per **UDPInput**, viene eseguita una **select()** sul selector per esaminare quali sono i canali pronti. A questo punto, se ce ne sono, viene preso il set di chiavi pronte e con un iteratore lo si scorre. Se la chiave è valida, si discriminano i due casi: il caso in cui si va in lettura, perché era quella l'operazione di interesse, e il caso in cui invece si è interessati alla connessione.

Si avviano dunque, nei due casi, due metodi differenti per gestire tali eventi.

processConnect()

In questo metodo viene acquisito il **referencePacket** dal tcb.
Nel caso **tcb.channel.finishConnect()** sia true, allora viene fatto il refresh del valore di accesso del **tcb**, e lo status viene posto a **SYN_RECEIVED**. Viene poi acquisito il buffer di risposta, aggiornato ai valori corretti, da rispedire al mittente secondo il passo 2 di handshake a tre vie; viene poi inviato al mittente e viene quindi aggiornato **mySequenceNum**, e il canale è ora settato con operazione di interesse **OP_READ**.

Codice 2.56: Cuore della processConnect()

```
...
Packet referencePacket = tcb.referencePacket;
  try {
      if (tcb.channel.finishConnect()) {

      tcb.refreshDataEXTime();

      keyIterator.remove();
      tcb.status = TCBStatus.SYN_RECEIVED;

      ByteBuffer responseBuffer = ByteBufferPool.acquire();
      referencePacket.updateTCPBuffer(responseBuffer, (byte)
          (Packet.TCPHeader.SYN | Packet.TCPHeader.ACK),
        tcb.mySequenceNum, tcb.myAcknowledgementNum, 0);
```

```
    Utils.addElementtoLbqTL("-IN: " + tcb.ipAndPort + " TCP netToDevice
        SYN|ACK");

    outputQueue.offer(responseBuffer);

    tcb.mySequenceNum++;
    key.interestOps(SelectionKey.OP_READ);
    } else {
       Utils.addElementtoLbqTL("-IN: "+tcb.ipAndPort + " not finishConnect!!!");
    }
  } catch (IOException e) {
    KLog.e(TAG, tcb.ipAndPort + " Connection error: " + e.toString());
    ByteBuffer responseBuffer = ByteBufferPool.acquire();
    referencePacket.updateTCPBuffer(responseBuffer, (byte) Packet.TCPHeader.RST,
        0, tcb.myAcknowledgementNum, 0);

    Utils.addElementtoLbqTL("-IN: "+tcb.ipAndPort + " TCP netToDevice RST");

    outputQueue.offer(responseBuffer);
    TCB.closeTCB(tcb);
  }
}
...
```

processInput()

In questo metodo viene rimossa la chiave dal set poichè sta per essere processata la richiesta. Viene allocato un buffer, in cui ovviamente viene lasciato lo spazio per l'header (perché la TUN è a livello IP) e alla fine si aspetta di leggere quel tipo di pacchetto. Anche qui viene rinfrescato il tempo di accesso alla **tcb**, viene letto un numero **readBytes** dalla **inputChannel** e viene salvato il dato corrispondente nella **receiveBuffer**. A seconda dello status in cui ci si trova ci saranno delle azioni da intraprendere. Se ho raggiunto la fine dello stream, allora è da controllare il caso in cui ci si trovi in **CLOSE_WAIT**, in cui si attende lo scadere del timeout e si effettua la release della tcb. Inoltre nel caso di fine stream e status **CLOSE_WAIT**, lo status diventa **LAST_ACK**, e viene creato un paccheto con header TCP di tipo FIN da inviare al client. Altrimenti, se non ci si trova alla fine dello stream, vengono aggiornati **readDataTime** e **readlen**. In questo secondo caso si ha a disposizione un dato da trasmettere al client, quindi si ricostruisce un pacchetto corretto da inviare indietro al client, si aggiornano tutti i valori del tcb e si fornisce il pacchetto sulla **NetworkToDeviceQueue**, pronto per essere elaborato dal thread VPNOutput.

Capitolo 3

L'attività di Monitoring

Come si è già discusso, si è scelto di monitorare ed effettuare il log sia delle informazioni relative alle connessioni TCP, sia delle informazioni relative allo scambio dati di livello applicativo. Le informazioni TCP sono scritte su di un file di Log inserito nella cartella "Documents" del dispositivo, mentre tutte le informazioni di livello applicativo sono raccolte in un Database che tiene traccia dei log di ogni sessione.
Si è scelto di effettuare il parsing ed il logging delle informazioni di livello applicativo dei seguenti protocolli: **HTTP, HTTPS, DNS, SMTP, POP3, IMAP**.

3.1 Riconoscimento dei protocolli basato sulle porte

Per effettuare il riconoscimento dei protocolli di livello applicativo si è ritenuto opportuno implementare un meccanismo di riconoscimento basato sull'identificazione del numero di porta destinazione dei segmenti inviati dal dispositivo verso la rete esterna, e della porta sorgente dei segmenti che dalla rete vengono inviati verso il dispositivo. Scegliendo una implementazione basata su tale meccanismo, discriminando quindi i diversi protocolli in base al differente numero di porta di default utilizzato, si è quindi posto un limite implicito al modulo software proxy realizzato: i protocolli che utilizzano porte di livello 4 differenti da quelle di default del limitato set che si è scelto di riconoscere (si è già discusso come tale set comprenda HTTP, HTTPS, DNS, SMTP, POP3, IMAP) non vengono riconosciuti; nel log pertanto, si registrano tali protocolli come "unknown" .

3.2 Un esempio: il logging di HTTP

Vediamo, ad esempio, come avviene il logging delle informazioni relative al protocollo HTTP, poichè i restanti protocolli vengono trattati analogamente (eccezion fatta ovviamente per le modalità con le quali viene parsato il messaggio: ad esempio HTTP usa **UTF_8** come codifica, mentre il DNS presenta un formato un po' più complesso).

Il protocollo HTTP è un protocollo di default attivo sulla porta TCP 80. Nell'architettura progettata dunque è necessario "iniettarsi" appena dopo l'intercettazione dei pacchetti TCP.

Codice 3.1: Esempio di injection nell'implementazione allo scopo di analizzare un segmento TCP

```
...
currentPacket = inputQueue.poll();
...
```

```
ByteBuffer payloadBuffer = currentPacket.backingBuffer;
currentPacket.backingBuffer = null;
ByteBuffer responseBuffer = ByteBufferPool.acquire();

InetAddress destinationAddress = currentPacket.ip4Header.destinationAddress;

TCPHeader tcpHeader = currentPacket.tcpHeader;
int destinationPort = tcpHeader.destinationPort;
int sourcePort = tcpHeader.sourcePort;

String ipAndPort = destinationAddress.getHostAddress() + ":" +
                    destinationPort + ":" + sourcePort;

TCB tcb = TCB.getTCB(ipAndPort);
...
Utils.parseTCPOutputApplicationProtocol(payloadBuffer, tcb.referencePacket,
    context);
...
```

Il metodo preposto al parsing del messaggio si occupa innanzitutto di effettuare una discriminazione dei protocolli in base alle porte di destinazione (come si è detto, per HTTP sarà la TCP 80); dopo aver quindi riconosciuto il protocollo di livello applicativo in base alla porta di destinazione si utilizzerà un parser ad-hoc per il determinato protocollo.

Codice 3.2: Discriminazione su una porta specifica

```
...
 if(referencePacket.tcpHeader.sourcePort == 80) {
          Utils.parseHttpResponse(receiveBuffer, referencePacket.tcpHeader,
              context);
      }
...
```

Dopo aver intercettato correttamente un pacchetto TCP è dunque necessario estrarre il payload di livello applicativo ed effettuare il parsing dei dati HTTP. In particolare si cercheranno gli header più interessanti del protocollo e se ne riporteranno i valori sul log.

Codice 3.3: Esempio di parsing dell'header dei pacchetti di richiesta HTTP

```
 ...
 String[] lines = s.split("\r\n|\r|\n");
          for (String line : lines) {
              if (line.contains("GET")) {
              finalString = finalString.concat("GET");
              }
              else if (line.contains("HEAD")) {
                  finalString = finalString.concat("HEAD");
              }
              else if (line.contains("POST")) {
                  finalString = finalString.concat("POST");
              } else if (line.contains("PUT")) {
                  finalString = finalString.concat("PUT");
              } else if (line.contains("DELETE")) {
                  finalString = finalString.concat("DELETE");
              } else if (line.contains("Host:")) {
```

```
                finalString = finalString.concat("\n"+line);
            } else if (line.contains("Connection:")) {
                finalString = finalString.concat("\n"+line);
            }
            else if (line.contains("User-Agent:")) {
                finalString = finalString.concat("\n"+line);
            }

        }
...
```

Si noti che per poter effettuare correttamente il parsing dei pacchetti HTTP è necessario dapprima convertire i byte del payload di livello applicativo secondo la codifica **UTF-8**, che è quella di default per HTTP.

Codice 3.4: Lo standard de facto di codifica per HTTP è UTF_8

```
...
final Charset doublebyte = StandardCharsets.UTF_8;
final CharBuffer encoded = doublebyte.decode(bufferB);
StringBuffer sb = new StringBuffer(encoded);
String s = sb.toString();
bufferB.position(pos);
String finalString = new String();
String[] lines = s.split("\r\n|\r|\n");
...
```

Per ogni pacchetto riportato nel log se ne riporta inoltre un timestamp, oltre che ad una stringa relativa al tipo di connettività usata (i.e. 3/4G oppure WiFi), rilevata mediante il metodo statico della classe Utils **typeConnection()**

Codice 3.5: Report delle informazioni sul pacchetto all'interno del log

```
...
if(!finalString.isEmpty()) {
            String information = new String();
            information=information.concat("---> \n");
            information=information.concat("\n"+Utils.DataAndTime());
            information=information.concat("\n"+Utils.typeConnection(context));
            information=information.concat("\nHTTP");
            information=information.concat("\n"+finalString);
            Log.i("REQUEST", finalString);
            Utils.addElementtoLbqS(information);
        }
...
```

Il flusso degli eventi descritto si riferisce, come si è intuito, al solo flusso di pacchetti HTTP in uscita dal dispositivo verso la rete. Considerazioni del tutto equivalenti si sono effettuate sul flusso HTTP in ingresso al dispositivo. Menzione speciale merita il log dei pacchetti relativi al protocollo DNS. Il flusso degli eventi che si susseguono per realizzare il logging dei pacchetti DNS è del tutto analogo al flusso relativo ad HTTP, tuttavia su alcune versioni di Android i pacchetti DNS non vengono "visti". Questo problema sarà discusso successivamente in **Limitazioni riscontrate**

Capitolo 4

Esempi di funzionamento e di logging

In questo capitolo si illustrano alcuni esempi corredati da screenshot del device utilizzato, per meglio comprendere il funzionamento pratico del modulo software proxy realizzato.

4.1 Esempio realizzato tramite utilizzo di un browser

Avviando l'applicazione ed utilizzando semplicemente un browser per la navigazione web è possibile osservare il flusso di pacchetti scambiati durante l'interazione client-server tra il dispositivo ed un host contattato tramite, appunto, il browser.

A titolo di esempio, vediamo il flusso di pacchetti di livello applicativo in una interazione HTTP tra il dispositivo ed un server contattato, ad esempio **www.gazzetta.it**, un sito che supporta HTTP. Innanzitutto il browser genera richieste DNS per il sito richiesto e riceve risposte contenti l'IP address da contattare per la specifica risorsa (un esempio è illustrato in figura 4.1).

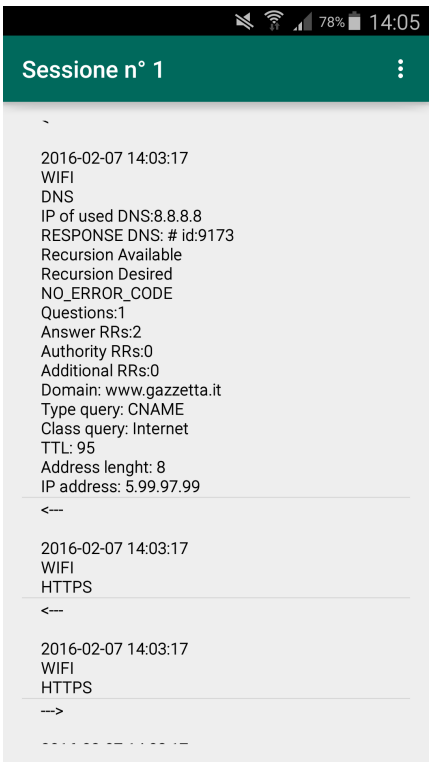


Figura 4.1: Logging di una sessione HTTP: all'inizio il browser genera richieste DNS.

Ricevute le risposte alle "request" DNS generate, il browser contatterà ora l'IP del sito **www.gazzetta.it** (e tutti gli IP degli host che possiedono le risorse per **www.gazzetta.it**, spesso site su diverse server di proprietà di Provider Content Delivery Network) per ottenere dati relativi alla struttura HTML del sito da fornire in visualizzazione all'utente finale, e riceverà delle risposte in accordo al corretto funzionamento del protocollo HTTP, come ad esempio illustrato in figura 4.2.

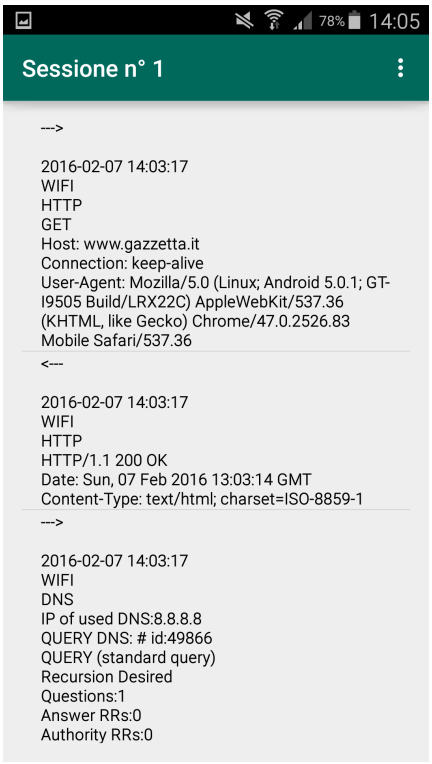


Figura 4.2: Logging di una sessione HTTP: la richiesta e la risposta ricevuta.

Si noti che per ogni pacchetto uscente dal dispositivo, si è scelto di immetterlo nel logging con l'intestazione "–>", mentre per ogni pacchetto in ingresso dalla rete al dispositivo si è scelto di utilizzare l'intestazione "<–".
In ogni pacchetto riportato nel file di log è riportato inoltre un timestamp, oltre ad una stringa che assume due valori **WIFI** oppure **3G (o 4G)** in base alla connessione di rete utilizzata in quel momento dal dispositivo.
Per quanto riguarda il protocollo HTTP, come si evince dalla figura, si è scelto di riportare il **method** utilizzato nell'interazione, oltre a vari campi ulteriori dell'intestazione HTTP tra cui **Host**, **Connection**, e lo **User-Agent** utilizzato. Nel caso di una risposta HTTP si è scelto di riportare campi specifici di una risposta di tale protocollo, come ad esempio **Server**, **Content-Type** e **Date**
Dalla figura 4.1 si può notare come si sia riusciti a riconoscere e ad intercettare anche i pacchetti del protocollo **HTTPS**. Essendo **HTTPS** un protocollo che prevede cifratura dell'intero pacchetto di livello applicativo si è scelto di non implementare alcun parser per tale protocollo dato che, essendo sottoposti a cifratura, i dati dell'header HTTPS sono invisibili e pertanto "non loggabili" (in verità, si è notato durante alcune sessioni di test come molti provider di servizi Web, ad esempio Google, utilizzino HTTPS non cifrando l'header ma solo i dati utili).

4.2 Esempio di log di pacchetti DNS

Vediamo ora un ulteriore esempio, illustrato in figura 4.3, spostando il focus sulle richieste DNS:

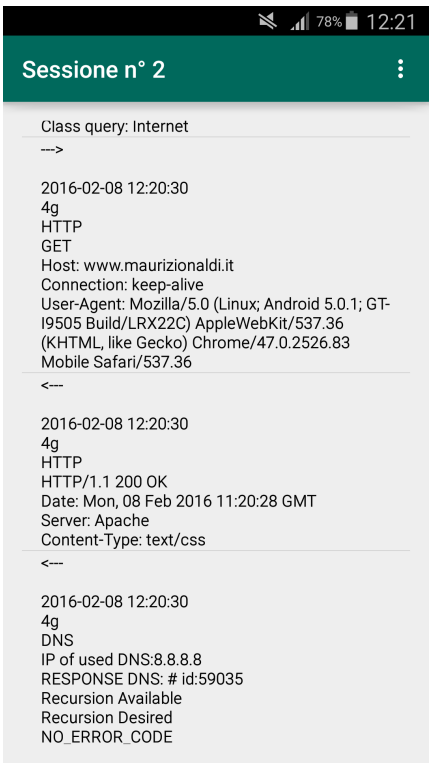


Figura 4.3: Un esempio di DNS con rete 4G.

Si noti che per quanto concerne i pacchetti DNS (in questo caso una risposta DNS), si è scelto di riportare nel log alcune proprietà richieste dal client e possibilmente fornite dal DNS Server durante l'interazione (ad esempio sulla ricorsione, sul numero di richieste inoltrate per la risoluzione,ecc.). Si è riportato inoltre l'indirizzo IP del particolare DNS Server contattato (si ricordi che su Android è 8.8.8.8 di default, ovvero quello di Google, ma potrebbe essere modificato manualmente dall'utente o tramite la già menzionata **addDnsServer()** di VPNService), e l'id associato a ciascuna richiesta/risposta DNS.

Infine, ricordando che si è scelto di realizzare un file di log apposito per tenere traccia degli stati TCP raggiunti dalle varie connessioni (tale file di log è salvato nella cartella "Documents" nel file system di Android) se ne riporta un esempio per poterne comprendere meglio la struttura in figura 4.4.

4.3 Esempio di log di pacchetti POP3

Ammettiamo ora di voler scaricare una e-mail dal nostro mail server. Oggi tutti noi utilizziamo un browser per richiedere tale servizio, o magari una applicazione client di posta elettronica. In entrambi i casi il protocollo di livello applicativo può risultare cifrato. Inoltre la porta utilizzata non sarebbe quella di default del protocollo. Per poter eseguire un esempio di parsing e di logging di un protollo applicativo afferente ad un servizio e-mail si è ricorso ad una connessione Telnet con il mail server, attraverso l'utilizzo di una applicazione che facesse da **client Telnet**. Un passo preliminare è stato quello di individuare un provider che offrisse un mail service, in grado di "sacrificarsi" ai fini

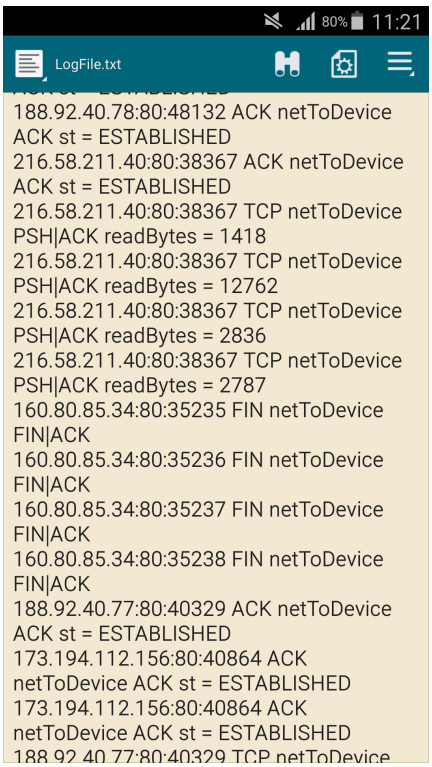


Figura 4.4: File di log per tenere traccia degli stati TCP.

dell'esempio in negativo (inizialmente si è provato con account di gmail e di hotmail che però hanno negato la possibilità di effettuare la richiesta in chiaro, richiedendo invece una connessione cifrata SSL). Creato perciò l'account di **falberto89@virgilio.it**, si è instaurata una connessione con il mail server alla porta 110.

4.3.1 La sessione di richieste SMTP tramite il TelnetClient

Avviando l'app che serve da client Telnet (nel caso dell'esempio **TelnetClient**, disponibile gratuitamente nel Play Store), ed effettuando la richiesta di connessione:

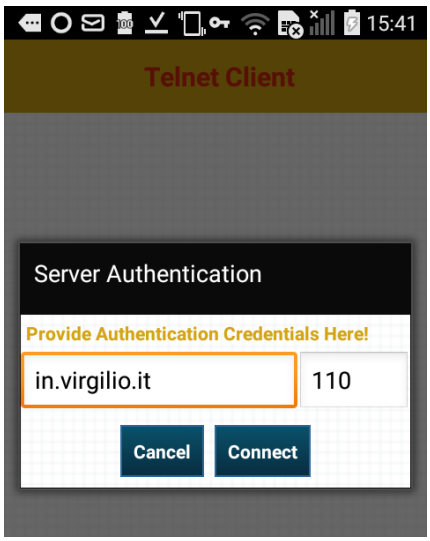
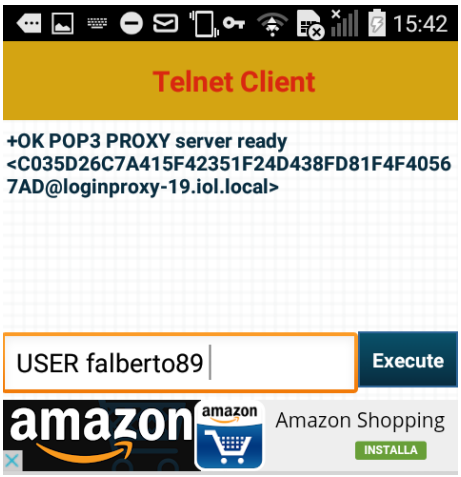


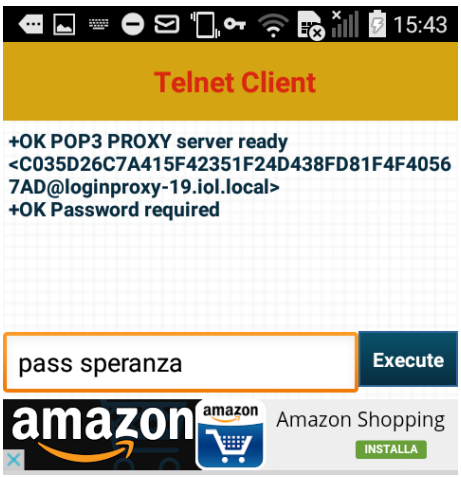
Figura 4.5: Richiesta connessione Telnet al mail server utilizzando TelnetClient.

Si inseriscono quindi username e password, attraverso i comandi **USER falberto89** e **PASS speranza**, ricevendo i relativi messaggi di ok. A questo punto è possibile impartire

tutti i comandi desiderati, dialogando con il proprio mail server. Nello specifico si eseguiranno un comando **LIST** ed un comando **RETR 3**, scaricando così il contenuto del proprio messaggio. Infine si è eseguito il comando **QUIT** per terminare la sessione (i passaggi salienti di queste operazioni sono illustrate in figura 4.6).

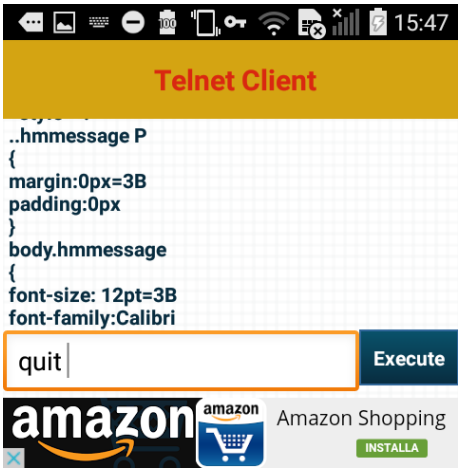


(a) *Comando USER <username>.*



(b) *Comando PASS <password>.*

(c) *Risposta al comando LIST e inserimento comando RETR <nr del messaggio>.*



(d) *Comando QUIT.*

Figura 4.6: I comandi impartiti al mail server tramite il client Telnet.

4.3.2 I pacchetti POP3 nel file di log

Nel file di log viene registrata tutta l'attività. A titolo di esempio, si illustra uno scambio di messaggi tra client e server in figura 4.7 in seguito ad una richiesta di tipo LIST

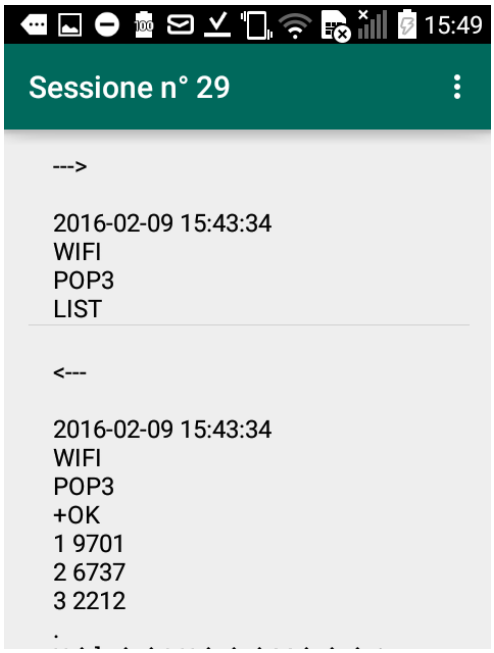


Figura 4.7: Logging di una sessione POP3: la richiesta e la risposta ricevuta al comando LIST.

In figura 4.8 è invece mostrata una richiesta di tipo RETR, specificando che si è voluto richiedere l'email numero 3.

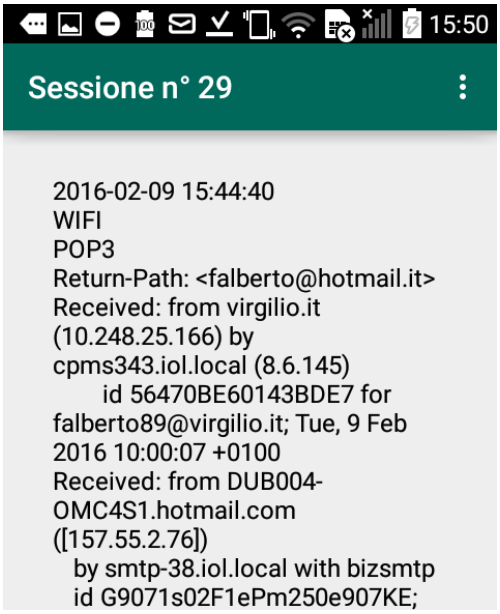


Figura 4.8: Logging di una sessione POP3: parte del messaggio di risposta al comando RETR 3.

4.4 Ancora sui protocolli di posta

Le stesse considerazioni e gl stessi risultati ottenuti per POP3 possono essere tranquillamente ottenuti per IMAP e SMTP. Ad esempio è illustrato in figura 4.9 uno screen di una sessione relativa ad IMAP.

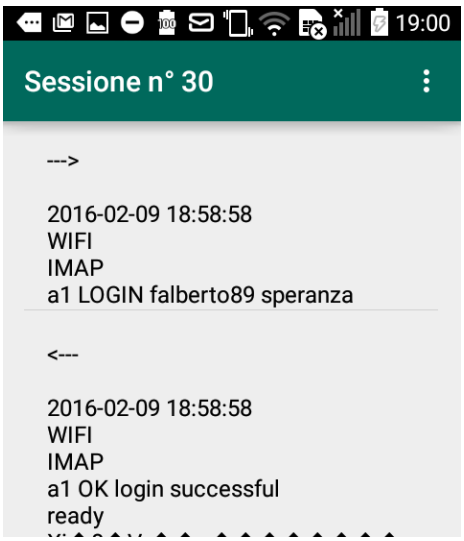


Figura 4.9: Logging di una sessione IMAP: la richiesta e la risposta ricevuta al comando a1 LOGIN <falberto89> <speranza>.

4.5 Protocollo “unknown”

In questa sezione si vuole invece mostrare una sessione monitorata e riportata nel log di uno scambio di dati afferenti ad un protocollo non presente nel sottoinsieme dei protocolli monitorati, pertanto considerato un protocollo “unknown” nel modulo implementato. Si è scelto, senza perdita di generalità, di testare l’applicazione **Zello**, disponibile gratuitamente sul Play Store, che utlizza UDP come protocollo di trasporto, e offre un servizio che consente di trasformare il proprio smartphone o tablet in un walkie talkie. In questo caso aperta l’applicazione è possibile riascoltare ad esempio un segnale di prova registrato, come illustrato in figura 4.10.

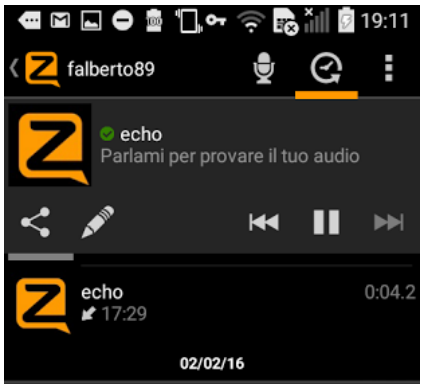


Figura 4.10: Uno screen dell’applicazione utilizzata.

In figura 4.11 è illustrato uno screen che illustra come effettivamente si sia tracciato lo scambio dati, si sia supportata con successo la trasmissione stessa, ma non si sia stati in grado di riconoscere il protocollo e parsarne il contenuto.

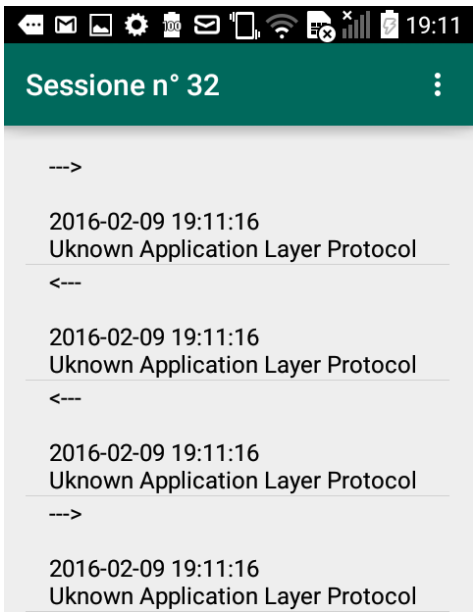
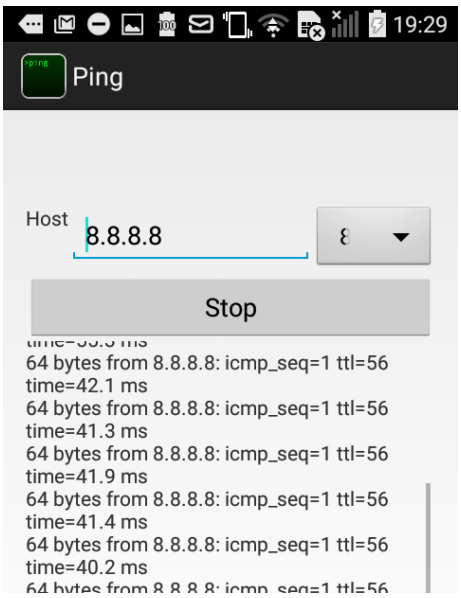


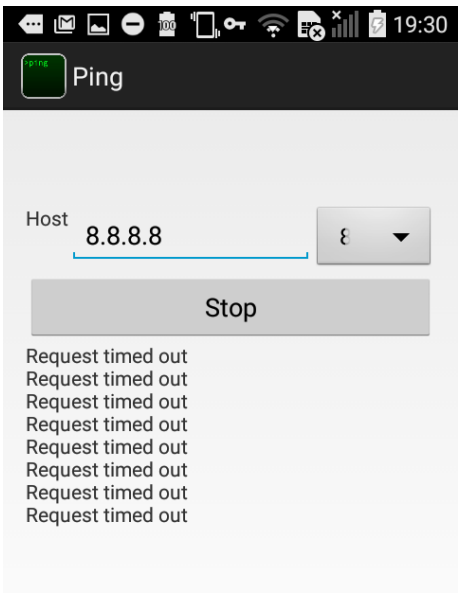
Figura 4.11: Logging di una sessione di un protocollo non noto.

4.6 Esempio ulteriore: un ping

Come esempio finale si propone l'invio di un ping, ad esempio con destinazione **8.8.8.8**, contattando dunque il server DNS di Google. Si è utilizzata l'app **Ping**, disponibile gratuitamente sullo store. In figura 4.12(a) è mostrato un ping che è stato effettuato con una navigazione con il modulo software proxy implementato disattivato, mentre in figura 4.12(b) è mostrato un ping effettuato con il modulo software proxy implementato attivato. Nel primo caso il ping è andato a buon fine, mentre nel secondo caso non è stato così. Il proxy server implementato, come implicitamente descritto nella trattazione delle scelte progettuali infatti, infatti, non supporta ICMP, perciò tale risultato non è affatto sorprendente.



(a) *Un ping effettuato con l'app disattivata.*



(b) *Un ping effettuato con l'app attivata.*

Figura 4.12: Un ping lanciato da un dispositivo Android senza (a) e con l'applicazione Proxy In The Middle attiva (b).

L'applicazione progettata prevede un supporto a questa eventualità, in particolare dunque nel log l'applicazione proxy segnala che dalla interfaccia TUN è stato prelevato un datagramma che utilizza un protocollo di trasporto che non è né TCP né UDP, e pertanto avvisa l'utente (come illustrato in figura 4.13).

Figura 4.13: Logging di una sessione in cui è lanciato un ping.

Capitolo 5

Limitazioni riscontrate

Su alcune versioni di Android (in particolare versioni stock di Android 4, ma anche versioni di Cyanogenmod) si sono riscontrati dei comportamenti notevoli. Si è già fatta notare in precedenza la presenza del metodo **addDnsServer()** in VPNService, un metodo che permette di impostare l'indirizzo IP del DNS server che l'interfaccia virtuale deve forzare per l'utilizzo. La sola presenza di tale metodo implicitamente indica che l'interfaccia virtuale opererà in qualche modo sui pacchetti che essa riconosce essere appartenenti al protocollo DNS (di livello applicativo). Uno dei comportamenti notevoli riscontrabili su determinate versioni di Android consiste nell'occultamento dei pacchetti DNS da parte dell'interfaccia virtuale messa su da VPNService. Essa infatti riconoscerà sicuramente i pacchetti DNS, come già illustrato, e (in questo schema di comportamento notevole) provvederà in modo indipendente a contattare il DNS Server specificato tramite il metodo **addDnsServer()**, processando il flusso di richieste-risposte DNS e restituendo indietro le risposte alle applicazioni richiedenti. Su tali versioni i pacchetti DNS sono quindi occultati dall'interfaccia virtuale, motivo per cui le letture dall'interfaccia (di pacchetti in uscita dal dispositivo verso la rete esterna) non restituiranno mai pacchetti il cui protocollo di livello applicativo è DNS, poichè gestiti in modo autonomo dalla stessa interfaccia virtuale VPNService. L'effetto che si ha utilizzando il modulo software proxy sulle versioni di Android interessate da tale comportamento è che nel log dei pacchetti di livello applicativo non si vedrà alcun pacchetto DNS, poichè non è possibile intercettarne alcuno in quanto gestiti in modo autonomo dall'interfaccia virtuale.
Un ulteriore comportamento notevole riscontrabile su determinate versioni di Android consiste nella modifica degli stessi pacchetti DNS da parte dell'interfaccia virtuale. In questo schema però l'effetto che si ha è la completa caduta della rete, se non sopperita adeguatamente. Sulle versioni di Android interessate da tale comportamento notevole ciò che accade è che, ancora una volta, l'interfaccia virtuale riconosce i pacchetti DNS, e la cosa interessante è che vi opera **modificandoli**. La modifica che viene operata è una modifica giudicabile concettualmente come **errata**, ed è difatti riconosciuto come un **bug noto** alla comunità di sviluppatori, che è stato fixato nelle versioni successive di Android. La modifica operata dall'interfaccia virtuale in questo schema di comportamento notevole è una modifica a livello IP. In particolare l'interfaccia virtuale modifica l'indirizzo IP sorgente dei pacchetti in uscita dal dispositivo riconosciuti come pacchetti DNS, ed imposta come indirizzo sorgente quello proprio dell'interfaccia virtuale stessa. Tale indirizzo, casualmente scelto dallo sviluppatore ed impostato tramite il metodo **addAddress()** di VPNService, può essere chiaramente un indirizzo IP non routable nè sulla rete in generale nè tantomeno sulla LAN in cui si trova il dispositivo su cui è installato il modulo software proxy (a meno di assegnare all'interfaccia virtuale proprio l'IP reale del dispositivo su cui esso è attivo; un altro caso interessante potrebbe essere quello in cui si assegna fortuitamente all'interfaccia virtuale l'indirizzo reale di un dispositivo reale nella LAN o,

in generale, nella rete: in questo caso si è praticamente sotto le ipotesi di IP Spoofing). Nel caso in cui sia presente tale comportamento notevole dunque, l'effetto ottenuto è che dal dispositivo escono su rete esclusivamente dei pacchetti DNS aventi come IP sorgente l'indirizzo dell'interfaccia virtuale. Pertanto **non si riceveranno mai pacchetti di risposta** e la connessione alla rete apparirà come fosse non disponibile. Nelle versioni di Android interessate da questo comportamento notevole i pacchetti DNS non vengono occultati dall'interfaccia virtuale ma modificati. E' pertanto possibile notare la presenza di tali pacchetti, ed è possibile anche fixare manualmente tale comportamento, ad esempio aprendo delle socket UDP (DNS gira di default su UDP 53) verso il DNS Server e interagendo con esso tramite detti canali, costruendo degli opportuni pacchetti IP di risposta da ritornare indietro all'interfaccia virtuale.

Capitolo 6

Conclusioni ed eventuali sviluppi futuri

La specifica di progetto richiedeva l'implementazione di un modulo software proxy su piattaforma Android, lasciando ampio spazio alle idee sulla implementazione finale. Nelle scelte di progetto si è deciso di non vincolare il dispositivo a procedure di rooting, ed è questo considerato un valore aggiunto per il modulo svilluppato, in quanto non vincola un eventuale utilizzatore finale a nessun tipo di operazione differente dalla semplice installazione del modulo, rilasciato sotto forma di applicazione, sul proprio dispositivo. Si è scelto di realizzare il modulo servendosi dell'ausilio di VPNService, un servizio Android che permette di metter su una interfaccia virtuale alla quale poter dirottare tutto il traffico di rete in entrata e in uscita, in modo da facilitare l'operazione di monitoring della rete. Dopo aver oltrepassato le difficoltà iniziali relative all'utilizzo della stessa VPNService, collegate alla necessità di doverne effettuare una corretta configurazione, si è poi proceduto a realizzare una architettura di gestione dei pacchetti intercettati grazie all'interfaccia virtuale. Tale architettura è necessaria per collegare il dispositivo alla rete esterna in modo corretto, peculiarità essenziale altrimenti non fornita dall'esclusivo utilizzo di VPNService (essa infatti fornisce la possibilità di metter su l'interfaccia virtuale, di configurare il dispositivo affinchè tutto il traffico passi attraverso di esso, ma non si occupa di gestire alcuna forma di comunicazione, un compito lasciato allo sviluppatore). Dopo aver correttamente costruito l'architettura ci si è occupati di fornire il modulo software di tutto il necessario per effettuare correttamente le attività di monitoring e logging dei pacchetti di rete intercettati. Infine ci si è occupati di dettagli applicativi, riguardanti il lato view (tenendo presente il pattern Model-View-Controller) dell'applicativo finale e la possibilità di fornire all'utente la feature aggiuntiva di scelta di un set di app installate sul proprio dispositivo da escludere all'attività di monitoring e logging realizzata dal modulo. Si ritiene di essere arrivati ad uno stato di realizzazione vicino a quello di una possibile release finale di un applicativo possibilmente disponibile al rilascio su di uno store di Applicazioni. Possibili migliorie e sviluppi futuri in ottica di release ufficiale sono discussi nel seguito. Uno dei possibili sviluppi futuri concerne la possibilità di visualizzare i pacchetti intercettati a runtime. Si è già predisposta una classe ShowSession, che però si è deciso di non utilizzare in quanto non ancora considerabile come stabile, pensata allo scopo di realizzare tale feature aggiuntiva.
Un ulteriore sviluppo futuro concerne la stabilità del modulo software proxy realizzato e la possibilità di ottenere un comportamento omogeneo in tutti i dispositivi. Si è già discusso nel capitolo **Limitazioni riscontrate** come in alcune versioni del sistema operativo Android si riscontrino dei comportamenti notevoli nell'utilizzare una interfaccia VPNService. Si potrebbe pensare di individuare il set di versioni interessate da tali comportamenti e rilasciare differenti versioni del modulo software proxy in base alle differenti versioni ed alle diverse disomogeneità.

Capitolo 7

Breve Manuale d'uso

Si è scelto di dedicare il capitolo finale di questa relazione alla realizzazione di un breve manuale d'uso, per chiudere il cerchio sulla comprensione a tutto tondo del modulo software proxy realizzato.
Per quanto concerne l'installazione, è possibile effettuarla semplicemente tramite l'utilizzo di un tool come Android Studio oppure semplicemente caricando il file .apk sul file system del dispositivo.
Dopo aver eseguito una corretta installazione, è possibile avviare l'applicazione accedendo così alla prima activity (Figura 7.1)

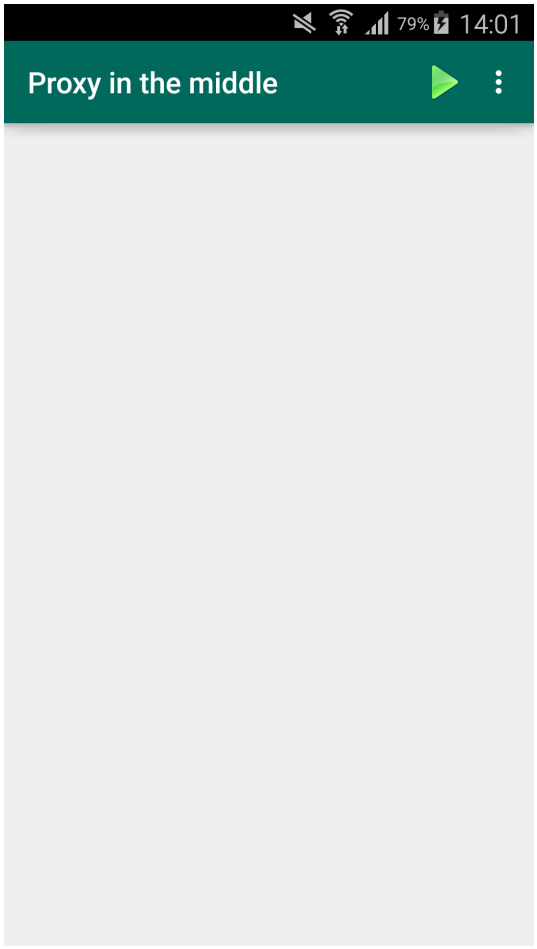


Figura 7.1: L'activity principale

In alto a destra (Figura 7.1) è possibile notare un pulsante di play ed un pulsante che si occupa di fornire un menù all'utente. Attraverso tale menù l'utente può accedere alle

due activity che si occupano rispettivamente di:

- visualizzare a schermo tutte le app installate sul dispositivo, in modo che l'utente possa scegliere quelle da escludere all'utilizzo del modulo software proxy, semplicemente toccandole sullo schermo
- riassumere le app scelte per l'esclusione all'utilizzo del modulo software proxy, e rimuoverle eventualmente da questa lista speciale

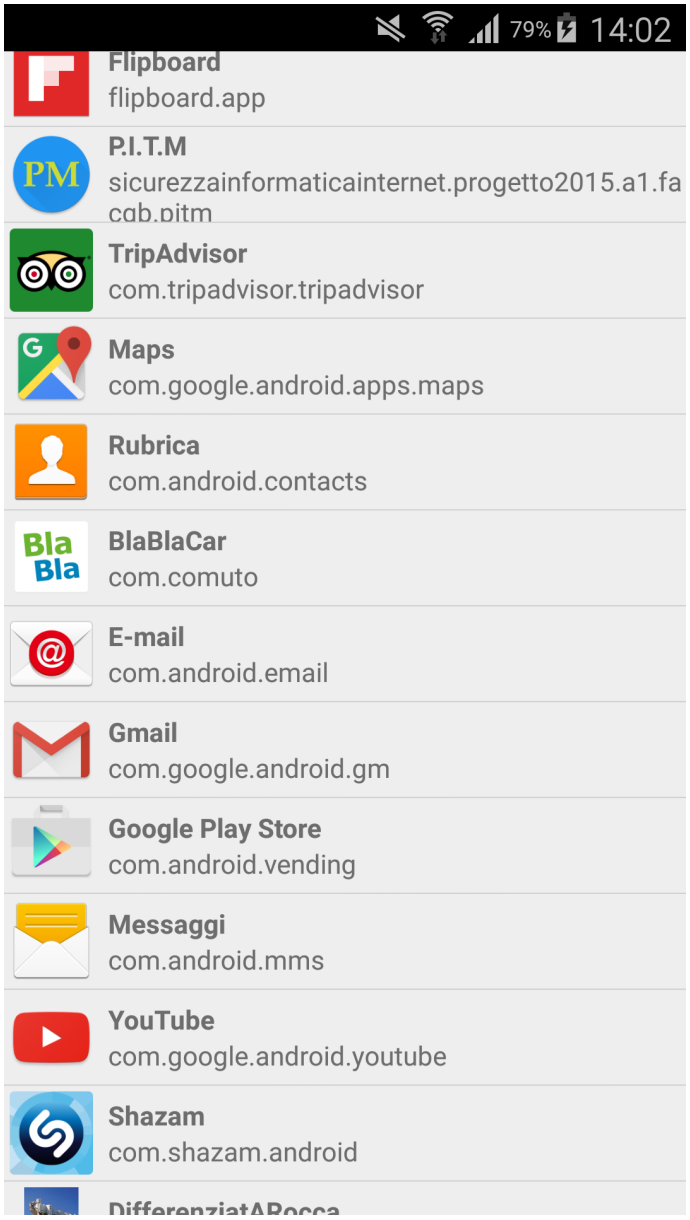


Figura 7.2: L'activity in cui l'utente può selezionare le app da escludere all'utilizzo del modulo software proxy

Dopo aver scelto le app da escludere all'utilizzo del modulo software proxy, è possibile iniziare una sessione in cui effettuare monitoring della rete, schiacchiando il pulsante di play nell'activity principale precedentemente illustrata. Dopo aver schiacciato il pulsante l'utente deve fornire manualmente il permesso[1] a VPNService di metter su l'interfaccia virtuale (Figura 7.4).

[1]Permettere alle applicazioni di intercettare i pacchetti solleva molte questioni riguardo la sicurezza, in

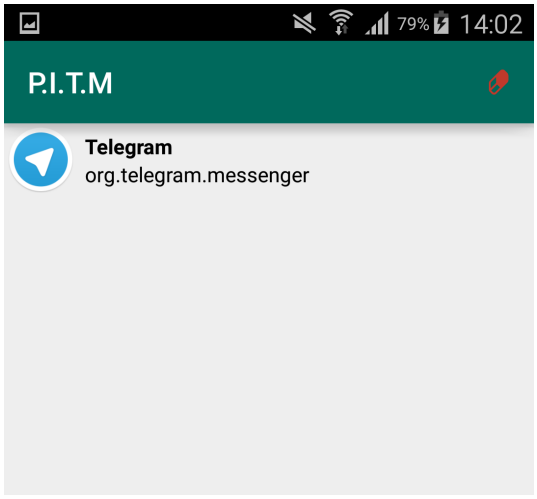


Figura 7.3: L'activity in cui l'utente può visualizzare le app da escludere all'utilizzo del modulo software proxy, ed eventualmente fare una clean di tale lista toccando il pulsante con la gomma rossa in alto a destra

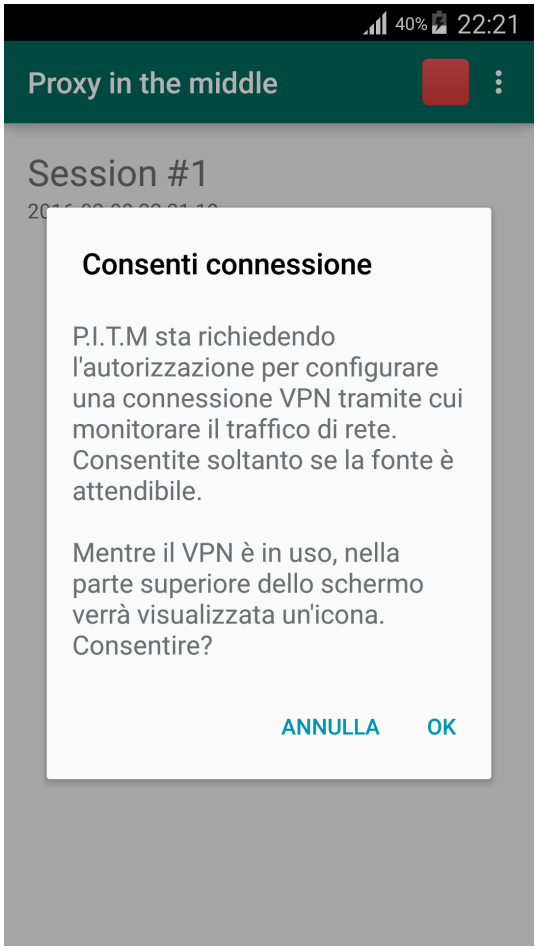


Figura 7.4: L'utente deve fornire esplicitamente il permesso al modulo software proxy di metter su l'interfaccia virtuale VPNService

Da questo momento in poi l'utente ha la possibilità di navigare sulla rete normalmente e, quando lo ritiene opportuno, può tornare nell'applicazione e stoppare l'attività di monitoring della rete schiacciando il pulsante rosso di stop, che si trova esattamente al posto del precedente pulsante verde di play.

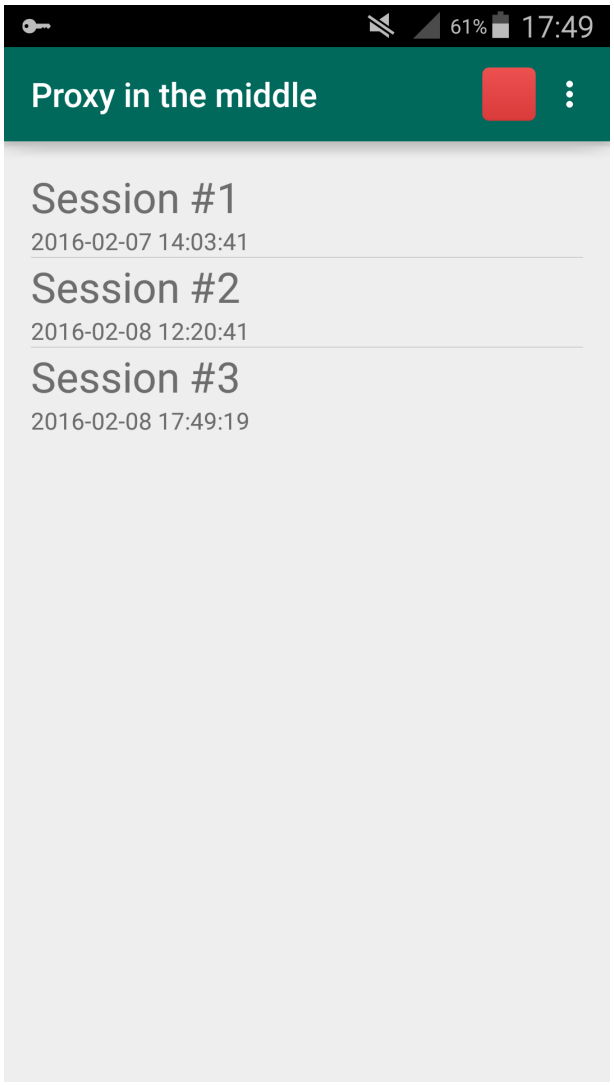


Figura 7.5: Nell'activity principale l'utente può decidere di stoppare in qualsiasi momento la sessione di monitoring schiacciando il pulsante di STOP rosso in alto a destra

Dopo aver stoppato la sessione corrente di monitoring, l'utente può visualizzare infine tutto ciò che il modulo software proxy ha intercettato correttamente, semplicemente toccando il numero di sessione a cui egli è interessato. In una situazione come quella riportata in figura 7.5, ad esempio, un utente può selezionare la Session #3, che rappresenta la terza sessione di monitoring che l'utente ha avviato in ordine temporale. Toccando il numero di una certa sessione, dunque, l'utente può accedere ai contenuti intercettati dal modulo software proxy sottostante, visualizzando una lista dei pacchetti in una activity di tipo scrollview (Figura 7.6), mentre per accedere al file di log TCP l'utente dovrà utilizzare uno

senso informatico, del dispositivo. Una applicazione che utilizza VPNService può facilmente far cadere la connessione di rete del dispositivo; inoltre può, in modo molto semplice, compiere azioni malevole a vario livello. E' per questo motivo che, in qualità di segnalazione all'utente di possibili problemi relativi alla sicurezza del dispositivo, ogni volta che si sta per metter su una interfaccia tramite VPNService viene attivato un Dialog che si occupa di richiedere un permesso esplicito all'utente, il quale se ritiene che l'app provenga da una fonte affidabile può decidere di accordare tale permesso.

strumento per navigare il file system (ad esempio l'applicazione Archivio) in modo da raggiungere la cartella Documents, dove si è scelto di memorizzare tale file (Figura 7.7)

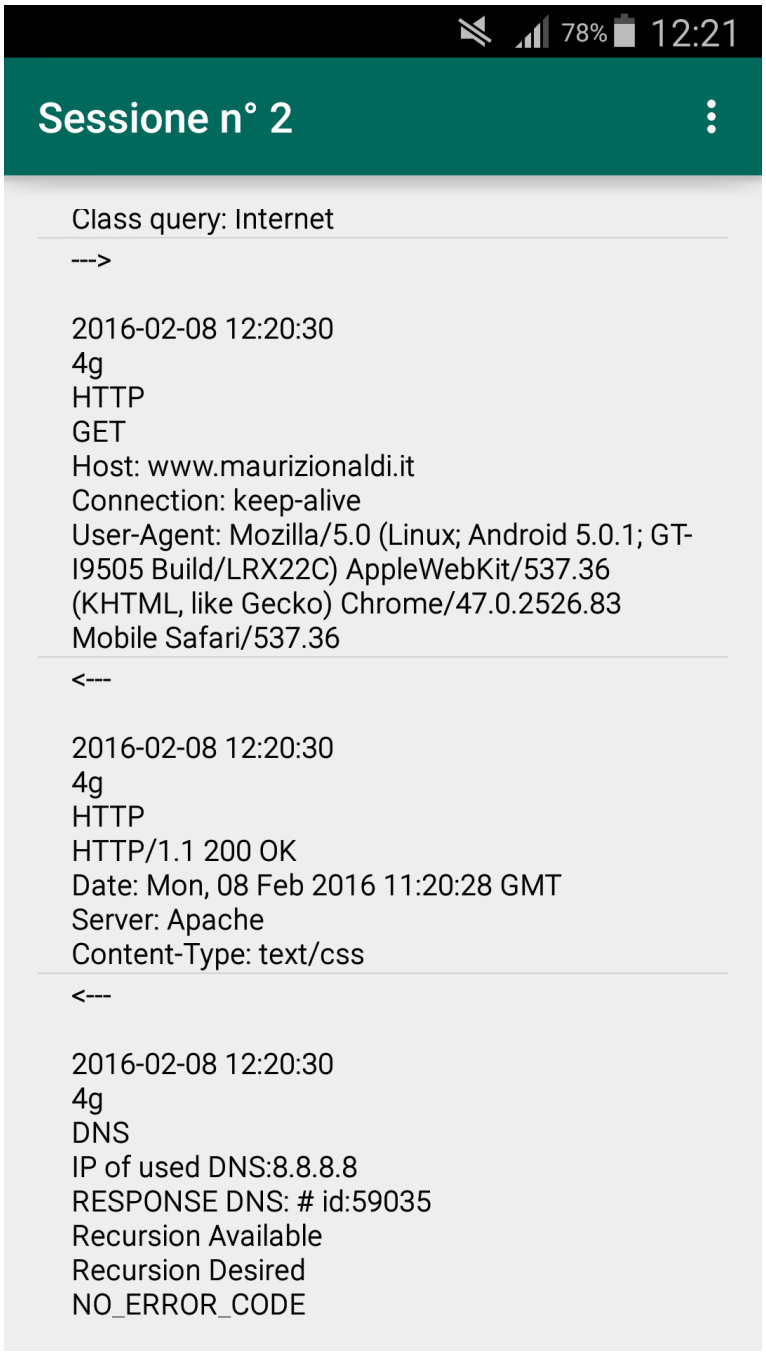


Figura 7.6: L'activity scrollabile in cui l'utente può visualizzare i pacchetti intercettati correttamente dal modulo software proxy, dopo aver selezionato una certa sessione di monitoring dall'activity principale

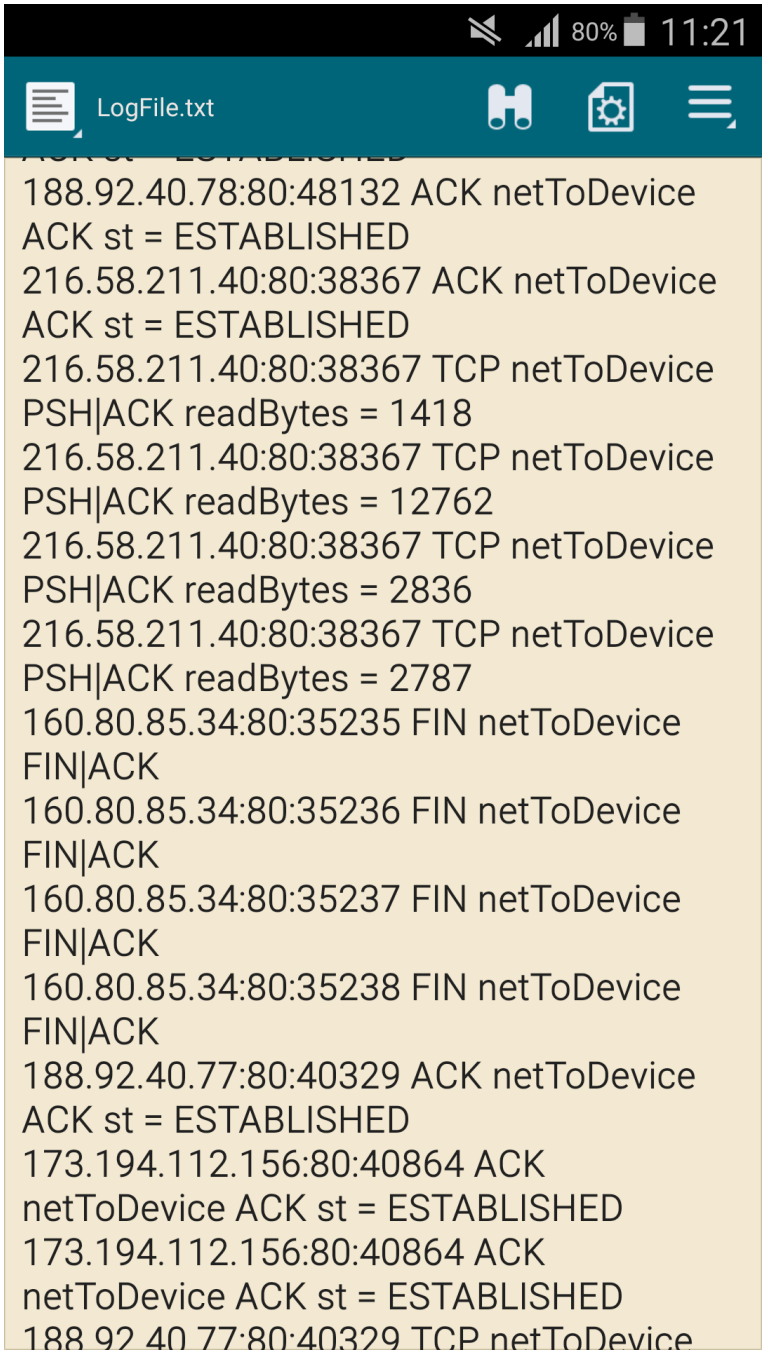


Figura 7.7: Il file di log TCP, che l'utente può visualizzare navigando fino alla cartella Documents di Android, ad esempio tramite l'applicazione Archivio

Bibliografia

[1] Pantieri Lorenzo - Gordini Tommaso, (2012) *L'arte di scrivere con LaTeX*

[2] http://www.lorenzopantieri.net/LaTeX_files/Codici.pdf *L'arte di scrivere con LaTeX*

[3] James F. Kurose, Keith W. Ross, *Reti di calcolatori e internet. Un approccio top-down*, (Marzo 2013)

[4] https://en.wikipedia.org/wiki/Proxy_server

[5] http://clark.tipistrani.it/html/sangiorgio/proxy/index.html

[6] https://www.androidpit.it/come-fare-root-android

[7] https://www.androidpit.it/10-buoni-motivi-root

[8] https://en.wikipedia.org/wiki/Iptables

[9] http://linux.die.net/man/8/iptables

[10] http://serverfault.com/questions/112795/how-can-i-run-a-server-on-linux-on-port-80-as-a-normal-user

[11] http://stackoverflow.com/questions/4577268/iptables-in-android

[12] http://www.lanzagiuseppe.it/nat_65.html

[13] http://tutorials.jenkov.com/java-nio/selectors.html

[14] http://tutorials.jenkov.com/java-nio/buffers.html

[15] http://www.thegeekstuff.com/2014/06/android-vpn-service/

[16] http://developer.android.com/intl/pt-br/reference/android/net/VpnService.html

[17] https://github.com/hexene/LocalVPN

[18] https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html

[19] http://developer.android.com/intl/pt-br/reference/java/util/LinkedHashMap.html

[20] http://tutorials.jenkov.com/java-nio/datagram-channel.html

[21] http://www.kensan.it/articoli/Telnet.php

[22] http://www.anta.net/misc/telnet-troubleshooting/imap.shtml