Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica per il Management

Progettazione e implementazione di un sistema per la raccolta di metadati di traffico di rete su Android

Relatore:
Prof.
Federico Montori

Correlatore:
Dott.
Ivan Zyrianoff

Presentata da:
Paolo Carlo Serri

IV Sessione
Anno Accademico 2024/2025

A nonna Graziella e nonno Sandro

Ringraziamenti

Desidero ringraziare il Prof. Federico Montori per la disponibilità e per l'opportunità di approfondire questi argomenti durante il mio percorso di studi.

Un sincero ringraziamento va al Dott. Ivan Zyrianoff per il supporto e i preziosi consigli forniti durante lo sviluppo del progetto e della tesi.

Ringrazio la mia famiglia per il sostegno costante durante tutto il percorso universitario.

Un grazie anche agli amici e alle persone a me vicine per il supporto e l'incoraggiamento.

Abstract

L'utilizzo diffuso degli smartphone rende il traffico di rete mobile una componente rilevante dell'interazione quotidiana con i servizi Internet. Le applicazioni installate sui dispositivi generano continuamente richieste verso server remoti, producendo pattern di utilizzo che possono variare in funzione del comportamento degli utenti, della posizione geografica e del momento della giornata. Tuttavia, lo studio di questi fenomeni viene spesso condotto utilizzando dataset forniti da operatori di rete o provider di servizi, che risultano generalmente aggregati e non permettono di osservare direttamente il traffico dal punto di vista del singolo dispositivo. In questa tesi viene presentato un sistema progettato per raccogliere metadati relativi alle connessioni generate da dispositivi Android. Il sistema è composto da un'applicazione mobile e da un server backend. L'applicazione utilizza un servizio VPN locale per osservare le connessioni di rete del dispositivo e raccogliere informazioni quali il protocollo utilizzato, la porta di destinazione, la durata della connessione, il timestamp e le coordinate GPS. I dati raccolti vengono inviati a un backend sviluppato con il framework NestJS e memorizzati in un database PostgreSQL. L'infrastruttura è stata progettata per supportare anche l'estensione PostGIS, consentendo possibili analisi geospaziali dei dati. Per verificare il funzionamento del sistema è stata condotta una fase di sperimentazione con dispositivi reali, durante la quale è stato raccolto un dataset composto da oltre 215 mila connessioni di rete. L'analisi dei dati ha permesso di osservare alcune caratteristiche del traffico mobile, tra cui la distribuzione delle connessioni per protocollo, la variazione temporale dell'attività di rete e la relazione tra il volume di traffico e la posizione geografica degli utenti.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei Codici

Capitolo 1

Introduzione

1.1 Contesto

Negli ultimi anni gli smartphone sono diventati uno degli strumenti principali attraverso cui gli utenti accedono ai servizi digitali. Applicazioni di messaggistica, piattaforme social, servizi di streaming e numerose altre applicazioni mobili interagiscono costantemente con infrastrutture remote attraverso la rete Internet.

Questo comportamento genera un flusso continuo di connessioni di rete che riflette non solo l'utilizzo delle applicazioni, ma anche il contesto in cui il dispositivo viene utilizzato. A differenza dei computer tradizionali, gli smartphone accompagnano l'utente durante l'intera giornata e vengono utilizzati in ambienti molto diversi tra loro, come abitazioni, luoghi di lavoro, spazi pubblici o durante gli spostamenti.

Di conseguenza, il traffico di rete generato da dispositivi mobili può essere interpretato come un fenomeno strettamente legato sia alle abitudini di utilizzo delle applicazioni sia alla mobilità dell'utente. In particolare, i pattern di traffico possono variare in funzione del comportamento dell'utente, del luogo in cui il dispositivo viene utilizzato e del momento della giornata. L'analisi di questi dati può quindi contribuire a comprendere meglio alcune caratteristiche del comportamento delle applicazioni e delle interazioni tra utenti e servizi digitali.

1.2 Problema

Lo studio del traffico mobile viene spesso condotto utilizzando dataset raccolti da operatori di rete o da grandi provider di servizi. Sebbene queste fonti consentano di analizzare volumi molto elevati di traffico, i dati disponibili risultano generalmente aggregati e non permettono di osservare direttamente il comportamento delle connessioni dal punto di vista del singolo dispositivo.

Questa limitazione rende più difficile studiare come le connessioni di rete vengano effettivamente generate durante l'utilizzo quotidiano di uno smartphone e come il traffico vari nel tempo o in relazione alla posizione geografica dell'utente.

Inoltre, la raccolta di informazioni sul traffico di rete a livello di dispositivo presenta diverse difficoltà tecniche. I sistemi operativi mobili impongono infatti restrizioni significative sull'accesso alle informazioni di rete e richiedono che le applicazioni rispettino specifici vincoli legati alla sicurezza e alla tutela della privacy degli utenti.

1.3 Soluzione

Per affrontare queste limitazioni, il lavoro presentato in questa tesi propone lo sviluppo di un sistema progettato per raccogliere metadati relativi alle connessioni di rete direttamente da dispositivi Android.

Il sistema sviluppato è composto da due componenti principali: un'applicazione mobile installata sul dispositivo e un backend responsabile della gestione e della memorizzazione dei dati raccolti.

L'applicazione utilizza un servizio VPN locale per osservare le connessioni generate dal dispositivo e raccogliere alcune informazioni relative al traffico, tra cui protocollo utilizzato, porta di destinazione, durata della connessione e coordinate geografiche ottenute tramite GPS. I dati vengono successivamente trasmessi a un backend remoto sviluppato con il framework NestJS, che si occupa della loro elaborazione e memorizzazione.

Il backend utilizza un database PostgreSQL con estensione PostGIS, che consente di associare ai metadati delle connessioni anche informazioni di geolocalizzazione del dispositivo. Questa architettura permette quindi di costruire un dataset strutturato che consente di analizzare alcune caratteristiche del traffico mobile dal punto di vista del dispositivo.

1.4 Valutazione

Per valutare il sistema sviluppato è stata condotta una fase di sperimentazione su dispositivi reali. L'applicazione è stata installata da un gruppo di 12 utenti che l'hanno utilizzata durante le normali attività quotidiane, consentendo la raccolta dei metadati relativi alle connessioni di rete generate dagli smartphone.

Durante la sperimentazione sono state registrate complessivamente 215 324 connessioni di rete. La quasi totalità dei record include anche informazioni di geoloca-

lizzazione del dispositivo (superiore al 99.99% delle connessioni registrate), rendendo possibile analizzare la distribuzione spaziale del traffico osservato.

L'analisi dei dati raccolti ha permesso di osservare diverse caratteristiche del traffico mobile, tra cui la distribuzione dei protocolli utilizzati, l'andamento temporale delle connessioni nel corso della giornata e la presenza di cluster geografici associati ai principali luoghi di utilizzo dei dispositivi.

I risultati dell'analisi e le principali osservazioni emerse durante la fase sperimentale sono presentati nel Capitolo 5, dedicato alla validazione del sistema.

Capitolo 2

Stato dell'arte

In questo capitolo viene affrontato il contesto teorico e tecnico necessario per comprendere le motivazioni e le scelte alla base del lavoro svolto. Inizialmente viene analizzata la relazione tra utilizzo di Internet e la geolocalizzazione, che rappresenta un aspetto importante per comprendere come la mobilità dell'utente influenzi il traffico di rete generato dal dispositivo mobile.

Successivamente vengono analizzate le principali tecniche per l'intercettazione del traffico su dispositivi Android e i vincoli imposti dal modello di sicurezza della piattaforma. In seguito viene presentato nel dettaglio il progetto open-source PCAPdroid [1], utilizzato come base architetturale del sistema sviluppato in questa tesi.

Il capitolo contiene inoltre una panoramica sulle architetture di edge caching e sulle strategie di ottimizzazione del traffico, per poi concludersi con un confronto tra le soluzioni esistenti e il sistema sviluppato in questa tesi.

2.1 Relazione tra utilizzo di Internet e geolocalizzazione

Navigare su Internet da un dispositivo mobile è una tipologia di utilizzo della rete direttamente collegata alla mobilità dell'utente nell'arco della giornata. A differenza dei sistemi desktop tradizionali, lo smartphone accompagna costantemente la persona in diversi contesti come: casa, luogo di lavoro, mezzi di trasporto e aree pubbliche. Ciò fa sì che il traffico di rete generato sia fortemente dipendente dal contesto spaziale e temporale.

Come evidenziato da diversi studi [2, 3, 4, 5], esiste una correlazione tra il tipo di traffico dati generato, il comportamento degli utenti e la posizione geografica

o la fascia oraria. In particolare, modelli di previsione del traffico che integrano informazioni di mobilità, come quello proposto in [4], mostrano che l'utilizzo di dati geospaziali migliora significativamente l'accuratezza delle stime. Allo stesso modo, l'approccio descritto in [5] utilizza modelli spazio-temporali per individuare, ad esempio, pattern che si ripetono nel traffico di rete mobile e prevedere quindi l'andamento di quest'ultimo nel tempo.

Alcuni lavori hanno inoltre analizzato in modo esplicito la relazione tra i contenuti consultati online e il contesto geografico degli utenti. Lo studio presentato in [6] evidenzia come i comportamenti di navigazione possano essere correlati ai punti di interesse frequentati dagli utenti, mostrando una forte relazione tra preferenze online e posizione fisica. Analogamente, il framework proposto in [7] analizza le correlazioni tra contenuti generati o consultati online e la dimensione spaziale, dimostrando come i pattern di utilizzo dei servizi digitali possano essere interpretati anche in funzione del contesto geografico.

Dal punto di vista applicativo, questa correlazione assume rilevanza in diversi ambiti. Poter associare richieste di rete a determinate aree geografiche, consente di comprendere meglio il comportamento degli utenti. In particolare, permette di individuare tipologie di richieste frequenti e analizzare il traffico dati nel tempo. In scenari più avanzati, queste informazioni possono essere utilizzate per ottimizzare la distribuzione dei contenuti e ridurre la latenza percepita dall'utente, migliorando la user-experience [8, 9].

Molte delle ricerche presenti in letteratura si basano su dataset raccolti ad esempio da operatori di rete o provider di servizi. In questi casi l'analisi viene effettuata su dati aggregati e non sempre consente di osservare direttamente il comportamento di un singolo dispositivo mobile.

Nella presente tesi, l'attenzione si sposta invece a livello del dispositivo. L'obiettivo principale non è sviluppare un modello predittivo, ma è realizzare un sistema in grado di raccogliere metadati delle connessioni di rete, associandoli a informazioni temporali e, con consenso esplicito dell'utente, alla posizione geografica del dispositivo. L'approccio descritto consente lo sviluppo di una base di dati strutturata utile per successive analisi di eventuali pattern, nel rispetto dei vincoli imposti dalla piattaforma Android.

La dimensione spaziale introduce una variabile importante per l'analisi del traffico di rete mobile. La mobilità dell'utente trasforma infatti il traffico in un fenomeno dinamico, la cui interpretazione richiede strumenti in grado di integrare informazioni temporali e geografiche.

L'associazione tra traffico di rete e posizione geografica, se da un lato permette

analisi e ottimizzazione, dall'altro solleva questioni rilevanti in termini di tutela della privacy. La combinazione di informazioni temporali, spaziali e relative alle applicazioni utilizzate può infatti contribuire alla ricostruzione di abitudini e pattern comportamentali di una persona. Per questo motivo, qualsiasi sistema che raccolga questo tipo di informazioni deve utilizzare un approccio basato sulla minimizzazione dei dati memorizzati e sulla trasparenza verso l'utente, rispettando i principi di privacy by design.

2.2 VPN (Virtual Private Network)

Le VPN (Virtual Private Network) sono una tecnologia di rete che consente di creare un canale di comunicazione sicuro tra un dispositivo e una rete remota attraverso Internet. Il traffico generato dalle applicazioni viene incapsulato all'interno di un tunnel logico e trasportato attraverso la rete pubblica come se il dispositivo fosse connesso direttamente alla rete privata.

Il meccanismo di funzionamento di una VPN si basa sul concetto di *tunneling*. I pacchetti generati dalle applicazioni vengono incapsulati all'interno di un nuovo pacchetto di rete e instradati attraverso il tunnel VPN verso l'endpoint remoto, dove vengono decapsulati e inoltrati verso la destinazione finale.

Una VPN è generalmente composta da diversi componenti principali:

- un'interfaccia di rete virtuale, che rappresenta il punto di ingresso del traffico nel tunnel;
- un meccanismo di incapsulamento dei pacchetti;
- un sistema di gestione delle connessioni e del routing del traffico;
- eventuali meccanismi di autenticazione e cifratura.

Nel sistema operativo Android, questo meccanismo è reso disponibile alle applicazioni tramite l'API *VpnService* [10], che consente di creare una VPN locale e di intercettare tutto il traffico IP generato dal dispositivo attraverso un'interfaccia virtuale.

2.3 Intercettazione del traffico mobile

La cattura del traffico di rete su dispositivi mobili rappresenta una sfida tecnica importante, soprattutto nel contesto del sistema operativo Android, che utilizza un modello di sicurezza fortemente basato sull'isolamento delle applicazioni.

2.3.1 Modello di sicurezza Android

Android implementa un meccanismo di sandbox applicativa [11] in cui ogni applicazione opera all'interno di un proprio spazio isolato, con un identificativo utente (UID) dedicato. Questo modello impedisce a un'applicazione di accedere direttamente ai dati o al traffico generato da altre applicazioni installate sul dispositivo.

In assenza di privilegi di root, non è possibile utilizzare strumenti tradizionali di packet sniffing a livello di sistema. Inoltre, la diffusione della cifratura end-to-end tramite HTTPS limita ulteriormente la possibilità di accedere al contenuto delle comunicazioni.

Di conseguenza, qualsiasi meccanismo di intercettazione deve basarsi esclusivamente sulle API ufficiali fornite dalla piattaforma.

2.3.2 Intercettazione tramite VpnService

L'unico strumento che permette l'intercettazione del traffico in uscita da un dispositivo Android non modificato è il servizio *VpnService*. Con la creazione di una VPN locale, l'applicazione può ricevere tutti i pacchetti IP generati dal sistema prima di essere inoltrati verso la rete esterna.

Dal punto di vista dello stack di rete, i pacchetti intercettati appartengono ai livelli inferiori del modello TCP/IP. In particolare, VpnService consente di accedere ai pacchetti a livello IP (network layer), che includono informazioni di instradamento come indirizzo sorgente, indirizzo di destinazione e protocollo di trasporto utilizzato.

Il payload del pacchetto IP contiene il segmento del livello di trasporto, che nel caso delle comunicazioni più comuni corrisponde ai protocolli TCP o UDP. Nel caso di TCP, l'header del segmento include campi come numeri di sequenza, acknowledgment e flag di controllo, utilizzati per garantire l'affidabilità della comunicazione e la corretta gestione dello stato della connessione.

La struttura degli header IPv4 e TCP, da cui vengono estratte molte delle informazioni utilizzate per identificare e gestire le connessioni di rete, è illustrata in Figura 2.1.

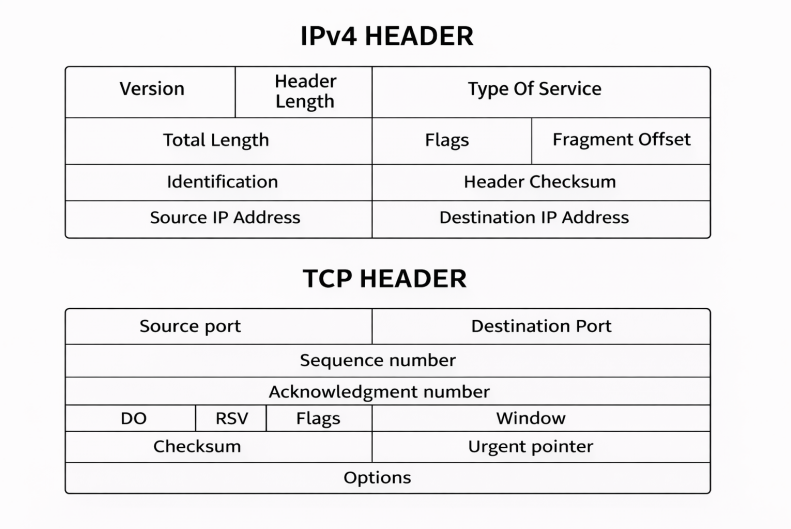


Figura 2.1: Struttura header IPv4 e header TCP

L'accesso ai pacchetti grezzi, però, non equivale ad avere flussi TCP già ricostruiti. Le applicazioni richiedono connessioni affidabili, in particolare una connessione TCP richiede il tracciamento dei sequence number, degli acknowledgment (ACK), di eventuali ritrasmissioni e dei meccanismi di controllo della congestione [12]. Un forwarding incompleto o non coerente con lo stato interno della connessione può generare pacchetti fuori sequenza, timeout o chiusure premature della comunicazione. Quando il sistema non osserva solamente i pacchetti ma deve intervenire nell'operazione di rinvio, occorre implementare una gestione strutturata dei flussi di trasporto, simile alla logica di uno stack TCP/IP in user-space.

La complessità di questi meccanismi mostra chiaramente come intercettare il traffico non si limiti alla semplice lettura dei pacchetti IP, ma richiede una comprensione approfondita dello stack di rete e del comportamento delle applicazioni.

La ricostruzione corretta dei flussi TCP richiede una gestione delle *five-tuple* (indirizzo IP sorgente, porta sorgente, indirizzo IP di destinazione, porta di destinazione e protocollo utilizzato), necessaria per identificare in modo univoco ogni connessione attiva e mappare i pacchetti intercettati verso i corrispondenti socket in user-space.

In aggiunta all'identificazione delle connessioni, occorre gestire il riassemblaggio dei segmenti TCP fuori ordine, la traduzione dei sequence number e degli ACK tra i due lati della comunicazione, nonché l'eventuale riscrittura dei parametri di stato della connessione, come avviene nei meccanismi di NAT (Network Address Translation).

Altre criticità derivano dalla gestione delle opzioni TCP, come Selective Acknowledgment (SACK) e window scaling. Il meccanismo SACK consente al destinatario

di segnalare in modo selettivo i segmenti ricevuti correttamente, permettendo al mittente di ritrasmettere solo i dati mancanti invece dell'intera finestra di trasmissione. Il window scaling, invece, estende la dimensione massima della finestra TCP per migliorare le prestazioni nelle reti ad alta latenza o con grande capacità di banda.

Ulteriori complessità sono legate alla frammentazione IP e alle variazioni della Maximum Transmission Unit (MTU), che possono causare la suddivisione dei pacchetti in più frammenti durante il percorso di rete. La gestione corretta di questi casi richiede meccanismi di riassemblaggio e controllo dei timeout, aumentando la complessità dell'implementazione di un sistema di forwarding trasparente.

2.3.3 Strategie per l'intercettazione del traffico

Tra le possibili soluzioni esplorate, sono state analizzate diverse strategie principali:

1. forwarding manuale dei pacchetti intercettati tramite VpnService;
2. utilizzo di motori *tun2socks*;
3. integrazione di soluzioni VPN complete (ad esempio OpenVPN);
4. utilizzo di strumenti open-source progettati per l'analisi del traffico.

In generale, queste alternative evidenziano un compromesso tra controllo completo del traffico e stabilità del sistema: maggiore è il livello di intervento sul flusso dei pacchetti, maggiore è il rischio di introdurre instabilità o malfunzionamenti.

Questa considerazione evidenzia come la cattura del traffico su dispositivi mobili non possa essere affrontata solo come problema di parsing dei pacchetti, ma richieda una valutazione complessiva dell'impatto sull'utente e sulla stabilità del sistema. In un contesto reale, la continuità della connettività rappresenta un requisito imprescindibile.

Forwarding manuale tramite VpnService Consiste nell'implementare manualmente il forwarding dei pacchetti intercettati tramite VpnService. Tale approccio richiede la gestione diretta dei flussi TCP, inclusa la ricostruzione delle connessioni, la gestione di sequence number e acknowledgment, la riscrittura degli indirizzi e delle porte e la sincronizzazione dei pacchetti in ingresso e in uscita.

Sebbene teoricamente percorribile, questa soluzione comporta un livello di complessità elevato e un rischio concreto di introdurre instabilità nella connettività del dispositivo, specialmente in presenza di ritrasmissioni o variazioni di stato della rete.

Utilizzo di motori tun2socks Integrazione di motori *tun2socks* [13], che permettono di convertire il traffico intercettato in connessioni gestite da uno stack SOCKS in user-space. Tale soluzione consente di delegare la gestione dello stack TCP a un componente dedicato.

Tuttavia, l'integrazione di tun2socks richiede l'utilizzo di librerie native e componenti JNI (Java Native Interface), ovvero meccanismi che consentono al codice Java di interagire con librerie native implementate in linguaggi come C o C++. Questo comporta un aumento della complessità di build e manutenzione.

Inoltre, molte implementazioni open-source presentano limitazioni in termini di documentazione, aggiornamento o compatibilità con le versioni recenti di Android, rendendo difficile un'integrazione stabile in un'applicazione Android standard.

Un aspetto critico emerso dall'analisi riguarda l'utilizzo di tun2socks in assenza di un server proxy esterno. Il modello di funzionamento tradizionale di tale strumento prevede infatti il flusso: TUN $\rightarrow$ tun2socks $\rightarrow$ proxy SOCKS5 $\rightarrow$ Internet.

Il protocollo SOCKS5 è un protocollo proxy di livello applicativo che permette di inoltrare connessioni TCP e UDP attraverso un server intermedio, il quale stabilisce la connessione verso la destinazione finale per conto del client. In questo schema, tun2socks non implementa uno stack TCP/IP completo, ma delega la gestione delle connessioni e dello stato dei flussi al server proxy.

L'eliminazione del proxy richiederebbe l'integrazione di un motore TCP user-space capace di ricostruire le connessioni, mantenere lo stato dei flussi e tradurre i pacchetti catturati in socket reali. Questa soluzione introduce un importante livello di difficoltà, in particolare sulla piattaforma Android, dove l'integrazione di stack di rete nativi richiede l'utilizzo di componenti NDK, binding JNI e librerie compilate specificamente per l'architettura del dispositivo.

Inoltre, l'assenza di artefatti precompilati stabili (AAR o librerie native aggiornate) rende complicata un'integrazione diretta di implementazioni esistenti in un progetto Android moderno. Per tali ragioni, l'utilizzo di tun2socks senza proxy, pur teoricamente possibile, risulta poco praticabile nel contesto del presente lavoro.

Soluzioni VPN complete (es. OpenVPN) Sono state considerate anche implementazioni complete di VPN basate su OpenVPN [14] o architetture simili. Tali soluzioni forniscono uno stack di rete completo e robusto, ma risultano spesso incomplete o sovradimensionate rispetto all'obiettivo di raccolta dei soli metadati del traffico.

L'integrazione di una VPN completa introduce vincoli architetturali aggiuntivi e aumenta il carico computazionale, risultando poco adeguata per un sistema orientato

all'analisi dei flussi piuttosto che alla gestione avanzata del routing.

Strumenti open-source orientati all'analisi Infine, sono stati analizzati strumenti open-source progettati specificamente per l'analisi del traffico su Android. In molti casi, queste soluzioni implementano internamente una delle strategie precedentemente descritte, ad esempio basandosi su VpnService e ricostruendo lo stato delle connessioni in user-space.

Tra questi, PCAPdroid si distingue per l'implementazione di VpnService e per la gestione interna delle connessioni, che consente di ricostruire i flussi TCP osservati senza richiedere una ricostruzione manuale completa dello stack TCP/IP né forwarding personalizzato.

Rispetto alle alternative esaminate, questa soluzione rappresenta un compromesso efficace tra capacità di intercettazione e stabilità della connettività. L'approccio adottato consente di osservare i metadati delle connessioni in modo non invasivo, evitando modifiche strutturali al sistema o l'introduzione di componenti nativi complessi.

Nel complesso, l'analisi comparativa evidenzia come le soluzioni che offrono un controllo completo del traffico comportino un aumento significativo della complessità e dei rischi operativi. Al contrario, l'utilizzo di VpnService tramite uno strumento strutturato come PCAPdroid consente di mantenere un equilibrio tra osservabilità del traffico e stabilità del sistema, come riassunto nella Tabella 2.1.

Tabella 2.1: Confronto tra le principali soluzioni analizzate

Soluzione	Proxy esterno	Stack TCP user-space	Complessità	Stabilità	Adeguatezza
Forwarding manuale	No	Necessario	Molto alta	Bassa	No
tun2socks + SOCKS	Sì	Delegato	Alta	Media	Parziale
tun2socks con stack integrato	No	Integrato	Molto alta	Media	Parziale
ICS-OpenVPN	No	Integrato	Alta	Alta	No (overkill)
TunMode	No	Parziale	Media	Bassa	No
PCAPdroid	No	Gestione connessioni	Media	Alta	Sì

Analizzando le soluzioni adottate in ambito industriale si ottiene un'ulteriore conferma di come sia difficile raccogliere dati di rete in modo trasparente in ambito Android. Le applicazioni VPN commerciali e open-source moderne integrano infatti uno stack TCP/IP completo in user-space o si affidano a motori di rete consolidati, evitando implementazioni manuali del forwarding basate esclusivamente su VpnService. Questa scelta mostra come la gestione corretta dei flussi TCP e UDP richieda componenti dedicati e maturi, difficilmente sostituibili con soluzioni semplificate.

2.4 PCAPdroid

Tra gli strumenti open-source disponibili per l'analisi del traffico di rete su dispositivi Android, PCAPdroid [1] rappresenta una delle soluzioni più complete e mature. Il progetto è nato con l'obiettivo di permettere l'analisi del traffico generato dalle applicazioni installate sul dispositivo senza richiedere permessi di root e senza modificare il sistema operativo.

A differenza dei tradizionali strumenti di packet sniffing, che richiedono accesso privilegiato al sistema, PCAPdroid si basa esclusivamente sulle API ufficiali fornite da Android, rendendo il progetto compatibile con dispositivi non modificati e distribuibile tramite i normali canali applicativi.

Utilizzo di VpnService e interfaccia TUN PCAPdroid si basa sull'utilizzo del servizio *VpnService*, che consente di creare una VPN locale a livello di sistema [10].

Dal punto di vista implementativo, la creazione della VPN locale avviene tramite l'utilizzo della classe *VpnService.Builder*, che consente di configurare parametri come indirizzo IP virtuale, route di default e server DNS utilizzati dall'interfaccia TUN. Un esempio semplificato di inizializzazione di una VPN locale tramite *VpnService* è riportato nell'Algoritmo 2.1.

Algoritmo 2.1 Configurazione di una VPN locale tramite VpnService

```
1: procedure InitializeVpnService
2:     builder ← new VpnService.Builder
3:     builder.setSession("PCAPdroid")
4:     builder.setMtu(1500)
5:     builder.addAddress("10.0.0.2", 32)
6:     builder.addRoute("0.0.0.0", 0)
7:     builder.addDnsServer("8.8.8.8")
8:     vpnInterface ← builder.establish()
9:     return vpnInterface
10: end procedure
```

Il metodo *establish()* restituisce un file descriptor associato all'interfaccia TUN virtuale, tramite il quale l'applicazione può leggere e scrivere i pacchetti IP intercettati.

Grazie a questo meccanismo, il sistema operativo invia tutto il traffico IP in uscita dal dispositivo verso un'interfaccia virtuale di tipo TUN [10], gestita dall'applicazione. Un'interfaccia TUN (network TUNnel) è un dispositivo di rete virtuale che opera a livello IP e permette alle applicazioni di leggere e scrivere direttamente pacchetti IP dallo spazio utente, senza passare attraverso uno stack di rete tradizionale del

sistema operativo. L'interfaccia TUN fornisce quindi pacchetti IP grezzi, permettendo all'applicazione di intercettare le comunicazioni prima che vengano inoltrate verso la rete esterna. In questo modo vengono osservate le connessioni generate dalle diverse applicazioni installate, mantenendo la connettività del dispositivo. Vantaggi di VpnService:

- è supportato ufficialmente dalla piattaforma Android;
- non richiede privilegi di root;
- non modifica la configurazione di rete del dispositivo;
- garantisce il consenso esplicito dell'utente tramite autorizzazione di sistema.

Tuttavia, l'interfaccia TUN fornisce solamente pacchetti IP e non flussi TCP già ricostruiti. Questo implica la necessità di gestire internamente lo stato delle connessioni.

Gestione delle connessioni e ricostruzione dei flussi In quanto l'interfaccia TUN ritorna pacchetti IP, PCAPdroid implementa una logica di gestione delle connessioni per ricostruire lo stato dei flussi TCP nel tempo. Internamente, l'applicazione mantiene una struttura dati che associa le connessioni attive alla corrispondente five-tuple (IP sorgente, porta sorgente, IP destinazione, porta destinazione e protocollo utilizzato).

Questa struttura aggiorna dinamicamente lo stato delle connessioni attive e invia notifiche in caso di chiusura. Il momento della chiusura della connessione rappresenta un punto informativo consolidato, in quanto rende disponibili metriche aggregate complete come:

- indirizzo IP e porta di destinazione;
- protocollo utilizzato;
- durata della connessione;
- numero totale di byte trasmessi e ricevuti.

Ciò evita di memorizzare dati parziali e riduce la complessità associata all'analisi dei singoli pacchetti. Dal punto di vista architetturale, l'utilizzo di VpnService implica che l'applicazione svolga temporaneamente il ruolo di gateway per il traffico del dispositivo. Questo comporta responsabilità aggiuntive nella gestione delle risorse, nella prevenzione di blocchi e nella corretta chiusura delle connessioni. Un errore nella gestione del flusso dei pacchetti potrebbe infatti compromettere la stabilità della connettività. La progettazione di uno strumento di intercettazione stabile richiede quindi equilibrio tra osservabilità dei dati di traffico di rete e trasparenza nei confronti del sistema operativo, evitando qualsiasi interferenza non strettamente ne-

cessaria con il normale funzionamento della rete. La Figura 2.2 illustra l'architettura e il flusso logico di PCAPdroid.

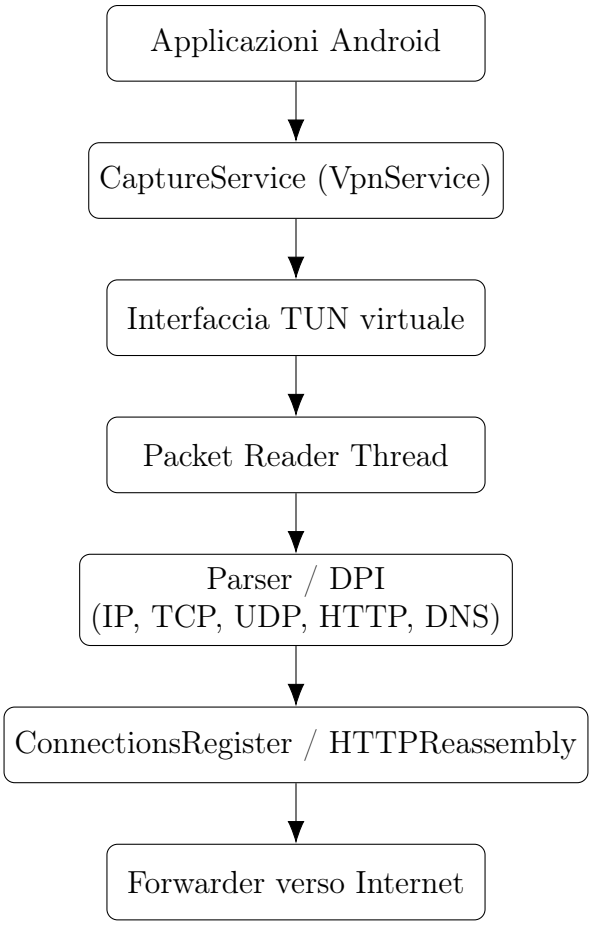


Figura 2.2: Architettura logica di PCAPdroid in modalità VPN-based

Osservabilità del traffico e limiti dovuti a TLS Il diffondersi del protocollo HTTPS e della cifratura TLS [15] rende difficile l'accesso al contenuto applicativo delle comunicazioni. PCAPdroid non implementa tecniche di tipo Man-in-the-Middle né installa certificati personalizzati per intercettare il payload cifrato. In conseguenza, l'analisi si limita ai metadati disponibili a livello di rete, ossia:

- indirizzo IP di destinazione;
- porta;
- protocollo;
- volume di traffico;
- dominio, se disponibile tramite il campo SNI (Server Name Indication).

Questa scelta progettuale rende lo strumento non invasivo e coerente con un utilizzo orientato all'analisi dei flussi piuttosto che all'ispezione del contenuto.

Motivazioni della scelta come base architetturale Tra le soluzioni presenti per l'intercettazione del traffico su Android, PCAPdroid risulta essere un compromesso efficace tra capacità di analisi e affidabilità operativa.

Soluzioni alternative che prevedono forwarding manuale del traffico o la ricostruzione completa dello stack TCP/IP in user-space, generano un aumento significativo della complessità del progetto e potrebbero rendere meno stabile la connettività del dispositivo nella rete. L'utilizzo di VpnService permette invece di intercettare il traffico senza interrompere la navigazione e senza richiedere configurazioni aggiuntive.

Per queste ragioni, PCAPdroid è stato utilizzato come base architetturale, attraverso un fork dedicato. Nel contesto dei progetti open-source, un fork indica una copia del repository originale che può essere modificata e sviluppata in modo indipendente rispetto al progetto principale. In questo caso, il fork è stato utilizzato per estendere alcune funzionalità senza modificare il meccanismo di intercettazione di PCAPdroid.

2.5 Edge caching

L'edge caching è oggi considerato una delle strategie più efficaci per ridurre la latenza e migliorare l'utilizzo delle risorse nei sistemi distribuiti. L'idea di fondo è semplice: avvicinare i contenuti agli utenti, posizionandoli in nodi periferici della rete così da limitare le richieste verso i server centrali e ridurre i tempi di risposta.

Nel contesto delle reti mobili, l'efficacia delle strategie di caching dipende fortemente dalla disponibilità di informazioni affidabili sui pattern di traffico e sul comportamento degli utenti nello spazio e nel tempo. Sistemi in grado di raccogliere metadati di traffico direttamente dai dispositivi mobili possono quindi rappresentare un importante elemento abilitante per tecniche di edge caching avanzate. In particolare, la disponibilità di dataset aggiornati dinamicamente consente lo sviluppo di strategie di caching proattivo o adattivo, orientate all'ottimizzazione dell'utilizzo dell'infrastruttura e alla riduzione della latenza percepita dagli utenti.

Questo approccio trova applicazione in contesti diversi, dalle Content Delivery Network (CDN) ai nodi di Multi-access Edge Computing (MEC), fino alle architetture IoT distribuite. In tutti questi scenari, la prossimità tra utente e nodo di caching consente di alleggerire il carico sul backbone e di migliorare la qualità percepita del servizio.

La scelta dei contenuti da memorizzare non è neutra: dipende dalla località dei dati e dalla ripetitività delle richieste in una determinata area. Nel caso delle reti

mobili, il problema si complica ulteriormente perché il traffico varia nello spazio e nel tempo. Gli utenti generano richieste differenti a seconda della posizione geografica e dell'orario; un sistema di caching efficace deve quindi adattarsi dinamicamente ai pattern di utilizzo.

In letteratura si distingue comunemente tra caching reattivo e caching proattivo. Nel primo caso i contenuti vengono memorizzati a seguito di richieste già osservate; nel secondo si tenta di anticipare le richieste future, pre-posizionando i contenuti in base a modelli previsionali. Il caching proattivo, in particolare, richiede strumenti accurati di previsione del traffico.

Una panoramica sistematica delle tecniche di edge caching in ambienti IoT è fornita in [8], dove viene evidenziato come la distribuzione geografica dei dispositivi e il comportamento degli utenti rappresentino fattori centrali per l'ottimizzazione delle risorse. L'architettura CACHE-IT proposta in [9] sviluppa ulteriormente questo approccio introducendo meccanismi di caching proattivo in scenari eterogenei, basati su informazioni contestuali e pattern di utilizzo.

Il lavoro presentato in [16] amplia la prospettiva introducendo strategie retention-aware e meccanismi di multi-path routing, con l'obiettivo di migliorare l'efficienza complessiva della distribuzione dei contenuti in reti wireless.

Un elemento ricorrente in tutti questi contributi è la necessità di disporre di dati affidabili sui pattern di traffico e sulla distribuzione geografica degli utenti. I modelli predittivi proposti in [4] e [5] mostrano come la componente spazio-temporale sia determinante per incrementare l'accuratezza delle stime sul traffico futuro.

Tuttavia, gran parte di questi studi assume la disponibilità di dataset già raccolti a livello infrastrutturale. Meno attenzione viene dedicata alle problematiche legate alla raccolta dei dati direttamente sul dispositivo mobile, in un contesto caratterizzato da vincoli stringenti di sicurezza e privacy.

Nel caso degli smartphone, la possibilità di associare pattern di traffico a coordinate geografiche apre scenari di ottimizzazione basati sulla distribuzione spaziale delle richieste. L'individuazione di aree con elevata concentrazione di traffico verso specifici servizi può infatti supportare strategie di caching localizzato e contribuire alla riduzione della latenza.

Nel presente lavoro non viene implementato un sistema di edge caching attivo. Il contributo si colloca piuttosto in una fase preliminare: il sistema sviluppato consente la raccolta strutturata dei metadati delle connessioni generate su dispositivi Android non rootati, associandoli a informazioni temporali e, quando disponibile, spaziali.

Una base dati di questo tipo può costituire un punto di partenza per successive analisi spazio-temporali e per eventuali strategie di caching proattivo. La disponi-

bilità di metadati associati a coordinate geografiche permette, ad esempio, di ipotizzare analisi di aggregazione spaziale e identificazione di hotspot di traffico, utili per decisioni di caching mirate.

In questo senso, l'edge caching rappresenta uno dei principali ambiti applicativi per l'utilizzo di dati spazio-temporali, evidenziando l'importanza di disporre di meccanismi affidabili di raccolta e strutturazione dei metadati di traffico.

2.6 Posizionamento del lavoro rispetto allo stato dell'arte

Diversi studi hanno analizzato il traffico generato dalle applicazioni mobili per comprenderne le caratteristiche e identificare pattern di utilizzo delle applicazioni e dei servizi di rete. Uno dei primi lavori in questo ambito è presentato in [17], che analizza il traffico generato da smartphone reali evidenziando come le applicazioni mobili producano pattern di comunicazione eterogenei e fortemente dipendenti dal comportamento dell'utente.

Altri lavori hanno esplorato la possibilità di raccogliere dati di traffico direttamente dai dispositivi mobili. Il sistema AntMonitor [18] propone una piattaforma di monitoraggio che consente di raccogliere e analizzare informazioni sul traffico generato dalle applicazioni su dispositivi Android senza richiedere privilegi di root. Tuttavia, AntMonitor è stato progettato principalmente come piattaforma sperimentale per la raccolta di misurazioni di traffico, nell'ambito della ricerca accademica.

Studi successivi hanno mostrato come sia possibile estrarre informazioni significative anche in presenza di traffico cifrato. In particolare, il lavoro presentato in [19] dimostra come sia possibile identificare le applicazioni che generano traffico di rete analizzando esclusivamente i metadati dei flussi, senza accedere al contenuto delle comunicazioni.

In gran parte della letteratura l'analisi del traffico mobile viene comunque affrontata principalmente da un punto di vista infrastrutturale o di rete core. L'attenzione si concentra soprattutto sulla previsione dei volumi del traffico, sulla classificazione dei flussi o sull'ottimizzazione della distribuzione dei contenuti, mentre risulta meno approfondita la realizzazione concreta di strumenti di intercettazione a livello di singolo dispositivo.

Gli studi orientati verso la previsione del traffico, come [4] e [5], si basano su dataset aggregati e su modelli di apprendimento automatico per stimare l'andamento futuro dei flussi. Analogamente, le ricerche sull'edge caching [8, 9, 16] analizzano

strategie di replica e posizionamento dei contenuti nei nodi periferici della rete, partendo da informazioni già strutturate sui pattern di utilizzo.

Un aspetto che emerge chiaramente è che molti di questi contributi assumono come punto di partenza la disponibilità di dati già raccolti. Risulta invece meno frequente un'analisi approfondita delle difficoltà tecniche legate alla fase di acquisizione dei dati direttamente su dispositivi mobili non rootati, dove occorre operare nel rispetto dei vincoli di sicurezza della piattaforma.

In questo scenario, l'utilizzo di VpnService costituisce una delle poche soluzioni effettivamente praticabili, pur tenendo conto dei limiti derivanti dalla cifratura TLS e dal modello di sicurezza Android.

Il lavoro presentato in questa tesi si inserisce in questo spazio, concentrandosi sulla progettazione di un'infrastruttura di raccolta compatibile con i vincoli della piattaforma Android e pensata come base per eventuali fasi successive di analisi e modellazione.

Dal punto di vista degli strumenti disponibili, esistono diverse applicazioni progettate per l'analisi del traffico su Android. Tra queste si possono citare, ad esempio, strumenti come NetGuard [20] o tPacketCapture [21], che utilizzano il servizio *VpnService* per intercettare il traffico di rete senza richiedere privilegi di root. Tali applicazioni sono principalmente orientate al monitoraggio del traffico in tempo reale o alla visualizzazione delle connessioni generate dalle applicazioni installate sul dispositivo.

Il sistema sviluppato in questa tesi si differenzia da questi strumenti sia per l'obiettivo principale del progetto sia per l'architettura di raccolta dei dati. Mentre molte applicazioni esistenti sono progettate per il debugging o l'analisi interattiva del traffico, il sistema proposto è orientato alla raccolta strutturata e persistente dei metadati delle connessioni, associati a informazioni temporali e, quando disponibile, alla posizione geografica del dispositivo.

Inoltre, lo scopo del lavoro non è sviluppare un ulteriore modello predittivo, ma affrontare una fase che spesso rimane implicita negli studi presenti in letteratura: la raccolta dei dati direttamente sul dispositivo mobile. Il sistema realizzato opera su smartphone Android non rootati e utilizza esclusivamente le API ufficiali della piattaforma, evitando interventi invasivi o modifiche del sistema operativo.

Questa scelta implica necessariamente il confronto con i vincoli imposti dal modello di sicurezza della piattaforma Android e dalla diffusione della cifratura TLS. In molti studi l'accesso ai flussi di traffico è dato per acquisito, mentre in un contesto reale l'osservazione sul dispositivo è limitata ai soli metadati disponibili a livello di rete. L'architettura proposta tiene conto di queste restrizioni e rinuncia espli-

citamente a tecniche come attacchi Man-in-the-Middle o installazione di certificati personalizzati.

Anche il tema della privacy assume un ruolo diverso rispetto a quanto accade in parte della letteratura orientata alle prestazioni. La raccolta è limitata ai metadati strettamente necessari e il sistema è progettato secondo principi di minimizzazione e trasparenza, lasciando all'utente il controllo sull'attivazione e sull'utilizzo del servizio.

Va inoltre sottolineato che, mentre molti contributi analizzano il traffico in ambienti simulati o a livello infrastrutturale, il sistema sviluppato in questa tesi costituisce un'implementazione funzionante che integra intercettazione, memorizzazione locale e sincronizzazione remota in un'unica architettura modulare.

Il contributo non risiede quindi nella definizione di un nuovo modello matematico, ma nella realizzazione di un'infrastruttura di acquisizione compatibile con i vincoli concreti della piattaforma mobile. Tale infrastruttura può rappresentare una base per successive analisi spazio-temporali o per strategie di ottimizzazione del traffico.

Dal punto di vista metodologico, l'approccio adottato rovescia la prospettiva tipica: invece di partire da un dataset già disponibile per applicare modelli predittivi, viene progettato e implementato l'intero processo di raccolta, dalla fase di intercettazione fino alla persistenza e alla sincronizzazione. Questo consente di mantenere un controllo esplicito sulle informazioni trattate, sui limiti tecnici derivanti dalla cifratura e sulle implicazioni in termini di tutela della privacy.

In questo senso, il lavoro presentato in questa tesi fornisce un'infrastruttura di raccolta dei dati che può costituire la base per successive attività di analisi e ottimizzazione del traffico.

Capitolo 3

Metodologia

Questo capitolo descrive la metodologia adottata per la progettazione e lo sviluppo del sistema proposto. In particolare vengono illustrati gli obiettivi progettuali che hanno guidato la definizione dell'architettura e le principali scelte tecniche adottate durante lo sviluppo.

Successivamente viene presentata l'architettura logica del sistema e la strategia utilizzata per l'intercettazione e l'aggregazione del traffico di rete. Il capitolo prosegue con la descrizione del modello dati utilizzato per rappresentare le connessioni osservate e con l'analisi delle strategie di persistenza e sincronizzazione tra dispositivo mobile e backend remoto.

Infine vengono illustrate le modalità di visualizzazione e analisi dei dati raccolti e le soluzioni adottate per garantire la tutela della privacy e la conformità ai principi del GDPR.

3.1 Obiettivi progettuali

La progettazione del sistema è stata guidata da diversi vincoli tecnici e normativi, legati principalmente alle limitazioni della piattaforma Android, ai requisiti di stabilità della connettività e agli aspetti di tutela della privacy.

In particolare, l'assenza di privilegi di root nei dispositivi Android limita l'accesso diretto al traffico generato dalle applicazioni e rende necessario utilizzare esclusivamente meccanismi ufficialmente supportati, come il servizio `VpnService`. Inoltre, la diffusione della cifratura TLS nelle comunicazioni moderne rende possibile osservare principalmente metadati di rete, come indirizzi IP, domini, protocollo e volume di traffico.

Alla luce di questi vincoli, il sistema è stato progettato con i seguenti obiettivi principali:

1. intercettare e strutturare i metadati del traffico generato da applicazioni su dispositivi Android non rootati, utilizzando esclusivamente API ufficiali;
2. garantire stabilità e continuità della navigazione;
3. definire un modello dati aggregato e coerente, adatto a analisi successive;
4. supportare un'architettura multiutente con identificazione sicura;
5. mantenere modularità ed estendibilità dell'architettura;
6. rispettare principi di tutela della privacy e di trasparenza verso l'utente.

I vincoli e gli obiettivi delineati costituiscono l'insieme delle motivazioni per cui sono state adottate le scelte architetturali illustrate nelle sezioni successive. La metodologia assunta rappresenta quindi il tentativo di sviluppare un sistema che rispetti limiti tecnici concreti, mantenendo stabilità operativa e potenziale analitico del dato raccolto.

3.2 Architettura logica complessiva del sistema

Tenendo conto dei vincoli descritti nel paragrafo precedente, il sistema è stato progettato secondo un'architettura modulare a livelli, con una separazione tra intercettazione del traffico, aggregazione dei dati, persistenza locale e sincronizzazione remota.

La divisione in livelli consente di mantenere separate le varie funzionalità del sistema. Ogni componente ha una responsabilità ben definita, così da ridurre eventuali dipendenze e limitare l'accoppiamento tra le parti del sistema. La fase di cattura del traffico rimane quindi stabile e indipendente dalle scelte relative alla persistenza o alla comunicazione con il backend.

A livello logico, l'architettura può essere descritta attraverso quattro macrolivelli:

1. **Traffic Interception Layer**
2. **Data Aggregation Layer**
3. **Local Persistence Layer**
4. **Remote Backend Layer**

I primi tre livelli operano sul dispositivo mobile, mentre l'ultimo è collocato su un server remoto. Nel lato Android è comunque possibile distinguere ulteriormente aggregazione, persistenza e sincronizzazione. La comunicazione tra dispositivo e backend avviene tramite protocollo HTTPS e autenticazione basata su token.

Il dispositivo si occupa di raccogliere e strutturare i dati intercettati, mentre il backend gestisce autenticazione, associazione sicura tra utente e record e persistenza centralizzata.

La struttura a livelli del sistema è rappresentata in Figura 3.1.

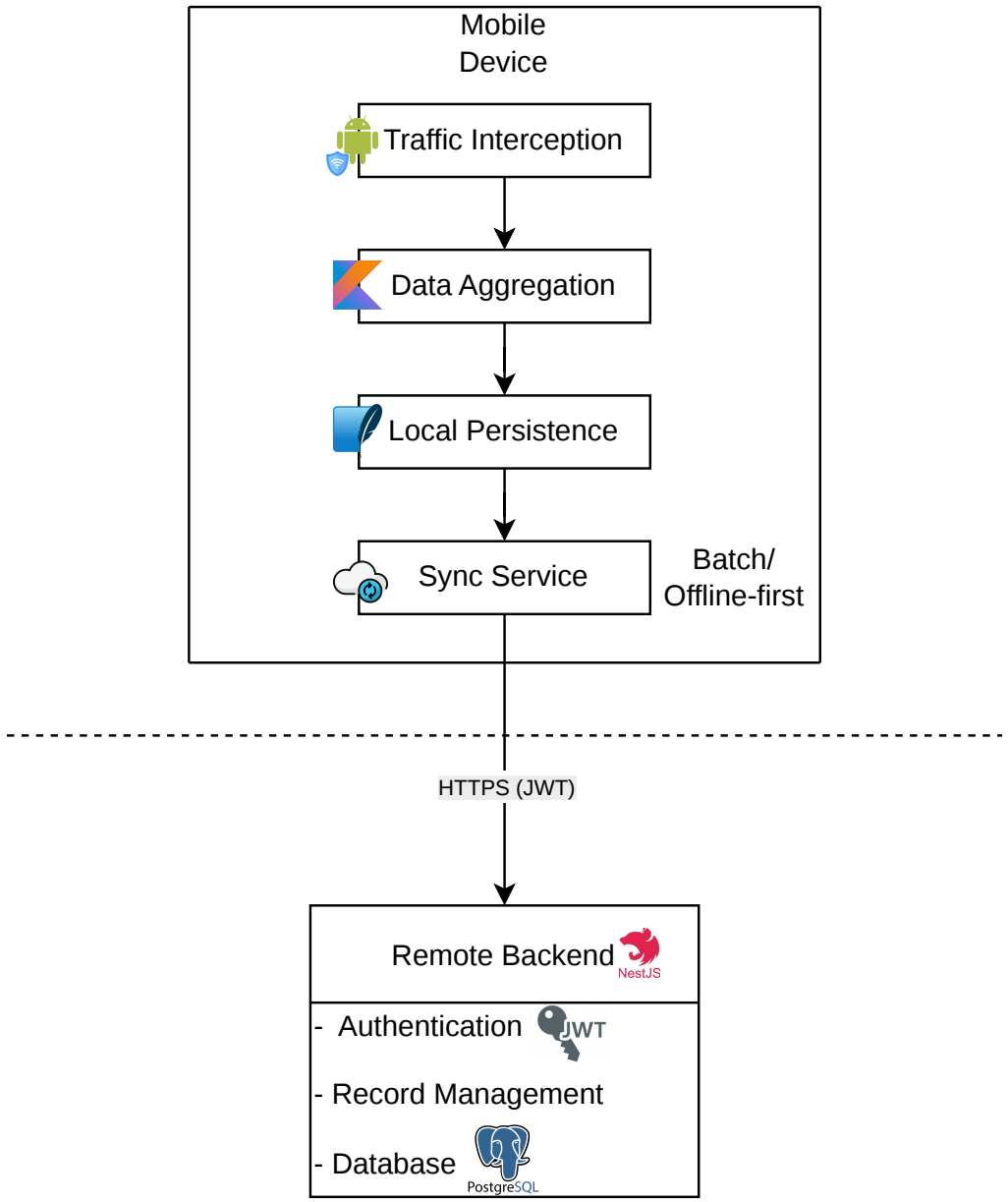


Figura 3.1: Architettura logica a livelli del sistema, con separazione tra componente mobile e backend remoto.

3.2.1 Traffic Interception Layer

Questo livello è responsabile dell'osservazione del traffico di rete generato dalle applicazioni installate sul dispositivo. Tale funzionalità è realizzata tramite il servizio Android `VpnService`, che consente di creare una VPN locale e di ottenere accesso ai pacchetti IP attraverso un'interfaccia virtuale di tipo TUN.

Nel sistema sviluppato, la gestione del tunnel VPN e la ricostruzione delle connessioni non vengono implementate direttamente dall'applicazione, ma sono delegate al motore di cattura integrato nel sistema. Questo componente riceve i pacchetti intercettati dall'interfaccia TUN e li associa alle rispettive connessioni di trasporto sulla base degli identificativi di rete.

Il livello di intercettazione si limita quindi ad osservare i metadati delle connessioni senza accedere al contenuto cifrato delle comunicazioni. I dettagli dell'integrazione con il motore di cattura e del processo di ricostruzione delle connessioni sono descritti nella Sezione 3.3. L'isolamento di questo livello rappresenta un aspetto fondamentale: la logica di cattura non deve dipendere dalla persistenza locale né dalla disponibilità del backend. Eventuali problemi di rete o di sincronizzazione non devono influire sulla stabilità della connessione del dispositivo. L'avvio e l'arresto del servizio di cattura non sono gestiti direttamente, ma tramite un componente intermedio che coordina il ciclo di vita della VPN e l'interazione con i livelli sottostanti.

3.2.2 Data Aggregation Layer

Il secondo livello trasforma dati di basso livello in unità informative strutturate. Il sistema aggrega le informazioni relative a una connessione al momento della sua chiusura, invece di memorizzare singoli pacchetti. Questa scelta consente di ottenere record completi, che includono:

- identificativi di rete (IP, dominio, protocollo);
- quantità di byte trasmessi e ricevuti;
- informazioni temporali, come ad esempio la durata della connessione;
- eventuali coordinate geografiche, se disponibili.

La scelta di aggregare le informazioni alla fine di ogni connessione permette di evitare di salvare dati parziali o duplicati e consente di ottenere metriche complete e coerenti. Questa modalità consente anche di limitare il numero di operazioni di scrittura sul database locale, riducendo il peso totale sulle prestazioni del dispositivo.

3.2.3 Local Persistence Layer

Il livello di persistenza locale è responsabile della memorizzazione locale dei record aggregati. Il database locale non rappresenta l'archivio definitivo dei dati, ma una cache persistente che garantisce robustezza anche in assenza di connettività. I dati vengono salvati immediatamente sul dispositivo, indipendentemente dalla disponibilità del server backend. Questo consente al sistema di seguire un modello *offline-first*, dove la raccolta delle informazioni non dipende dalla connettività. Ogni record fornisce uno stato che permette di distinguere i dati già trasmessi dai dati ancora da sincronizzare. Ciò consente di eseguire la sincronizzazione in modo incrementale e mantenere coerenza tra livello locale e remoto. La separazione tra aggregazione e persistenza consente di mantenere indipendente la costruzione del record dalla gestione dello storage, riducendo l'impatto di eventuali modifiche strutturali.

3.2.4 Remote Backend Layer

Il backend è progettato secondo una separazione tra livello di autenticazione, livello di gestione dei record e livello di persistenza, al fine di ridurre l'accoppiamento tra le componenti server-side e facilitarne l'estendibilità.

Il server remoto riceve i dati in modalità batch, li associa all'utente autenticato e li memorizza in modo centralizzato. L'architettura del server è stateless: l'identità dell'utente non viene determinata tramite sessioni persistenti, ma esclusivamente attraverso il token di autenticazione. L'associazione tra utente e record avviene quindi in modo sicuro lato server, senza che il client possa dichiarare direttamente la propria identità. La sincronizzazione avviene in modo asincrono e incrementale e vengono trasmessi solo i record non ancora inviati, in modo da ridurre traffico superfluo e consumo energetico. Se il server risulta non raggiungibile, i dati rimangono memorizzati nel database locale e vengono ritrasmessi successivamente.

3.2.5 Separazione delle responsabilità

L'architettura nel suo complesso mantiene una netta separazione delle responsabilità:

- il componente di intercettazione non conosce dettagli di persistenza o sincronizzazione;
- la persistenza non interviene nella cattura del traffico;
- il backend non decide quali dati raccogliere, si limita a riceverli e associarli all'utente autenticato;

- il client non controlla l'identità utente lato server.

Questa impostazione riduce l'accoppiamento tra componenti, favorisce la manutenibilità e rende il sistema facilmente estendibile, ad esempio verso analisi avanzate lato server. L'architettura logica descritta costituisce la base metodologica su cui si sviluppa l'implementazione tecnica illustrata nel capitolo successivo.

3.3 Strategia di intercettazione e aggregazione

La raccolta dei dati nel sistema segue tre fasi principali: intercettazione del traffico tramite VPN locale, ricostruzione delle connessioni di trasporto e aggregazione dei metadati a connessione conclusa. Le sottosezioni seguenti descrivono come il motore di cattura gestisce i pacchetti intercettati e come tali informazioni vengono trasformate in record strutturati utilizzati dai livelli successivi dell'architettura.

La raccolta dati viene effettuata tramite una VPN locale basata sul servizio Android `VpnService`, che consente di intercettare i pacchetti IP attraverso un'interfaccia virtuale di tipo TUN.

Il sistema utilizza un motore di cattura derivato dal progetto open-source PCAPdroid, già in grado di gestire il tunnel VPN e la ricostruzione delle connessioni di trasporto. Le motivazioni relative alla scelta di questa soluzione e il confronto con altre alternative sono stati discussi nel Capitolo 2.

Nelle sezioni seguenti vengono descritti il processo di integrazione con il motore di cattura e le scelte progettuali adottate per trasformare i dati osservati in connessioni aggregate persistenti.

3.3.1 Strategia di integrazione del motore di cattura

L'integrazione è stata realizzata in modo non invasivo. Il funzionamento interno del tunnel VPN non è stato modificato, ma è stato implementato un livello applicativo che intercetta i metadati esclusivamente quando le connessioni risultano concluse.

Attraverso l'interfaccia TUN, i pacchetti IP generati dalle applicazioni vengono associati alle rispettive connessioni attive, che vengono identificate tramite parametri di trasporto (indirizzi IP, porte e protocollo). Durante la connessione, il sistema aggiorna le metriche associate fino al momento della chiusura; in quel momento i dati aggregati vengono trasferiti al livello superiore dell'applicazione.

Oltre ai metadati di trasporto già disponibili nel motore originale, il sistema è stato esteso per includere informazioni aggiuntive rilevanti per l'analisi, come even-

tuali attributi HTTP non cifrati, timestamp e coordinate geografiche del dispositivo, qualora autorizzate dall'utente.

Ciò è stato fatto tramite integrazioni mirate nel codice del fork, mantenendo la logica di cattura separata da quella di dominio senza alterare il funzionamento fondamentale del tunnel VPN.

In questo modo viene preservata la stabilità del servizio di intercettazione e allo stesso tempo viene arricchito il modello dati.

3.3.2 Aggregazione a fine connessione

Una scelta importante riguarda il momento in cui il dato viene trasformato in unità persistente. Il sistema non salva singoli pacchetti né eventi intermedi, ma genera un record soltanto alla chiusura della connessione.

La connessione completata diventa l'unità informativa del sistema, che quindi consente di recuperare metriche complete ed evitare di salvare dati parziali o incoerenti.

3.3.3 Esclusione della memorizzazione a livello di pacchetto

È stata esclusa intenzionalmente la memorizzazione a livello di singolo pacchetto. Un approccio packet-level avrebbe comportato un volume di dati significativamente maggiore, una gestione più complessa dello stato dei flussi e un possibile impatto sulle prestazioni del dispositivo. Operare a livello di connessione aggregata rappresenta quindi un compromesso tra livello di dettaglio e sostenibilità operativa, coerente con il contesto di un'applicazione mobile non privilegiata.

3.4 Modello dati

La definizione del modello dati ha rappresentato uno dei passaggi fondamentali della progettazione. Il modello progettato ha permesso di lavorare su un'entità compatta ma sufficientemente espressiva, senza scendere a livelli di dettaglio eccessivi che avrebbero reso complessa la gestione dei dati nel tempo.

3.4.1 Struttura del record di connessione

Ogni record corrisponde a una connessione terminata e raccoglie informazioni come:

- identificativi di rete (indirizzo IP di destinazione, porta, protocollo, eventuale dominio);

- metriche di traffico (byte trasmessi e ricevuti);
- informazioni temporali (inizio e fine connessione, durata complessiva);
- attributi applicativi non cifrati, se disponibili (ad esempio metodo e path HTTP);
- coordinate geografiche, con consenso esplicito fornito.

L'obiettivo non è quello di descrivere in modo esaustivo ogni dettaglio della comunicazione, ma di ottenere una rappresentazione coerente che metta in relazione informazioni temporali, spaziali e del traffico dati.

Durante le prime fasi dello sviluppo erano stati presi in considerazione modelli separati per la connessione di rete e per le richieste HTTP. Questa configurazione richiedeva una correlazione esplicita tra livelli differenti e introduceva maggiore complessità nella gestione della persistenza.

Si è optato quindi per una struttura unificata, nella quale la connessione aggregata costituisce l'entità centrale e può includere, quando presenti, anche attributi applicativi non cifrati. Questa revisione ha semplificato il flusso complessivo dei dati e ha reso più lineare la fase di sincronizzazione con il backend. L'architettura è stata raffinata iterativamente durante lo sviluppo.

3.4.2 Stato di sincronizzazione e principi di progettazione

Ogni record contiene uno stato di sincronizzazione, che consente di distinguere tra dati già trasmessi al server remoto e dati ancora memorizzati esclusivamente nella cache locale. Questo attributo non appartiene al dominio della connessione in sé, ma è una proprietà utile alla gestione del dato all'interno del sistema. La sua introduzione permette di supportare il modello offline-first e di eseguire la sincronizzazione in modo progressivo senza duplicazioni.

Il modello dati è stato definito seguendo alcuni criteri:

- mantenere coerenza interna del record;
- raccogliere esclusivamente le informazioni utili;
- consentire l'estendibilità della struttura senza dover modificare la logica di base.

La modellazione del dato costituisce quindi il punto di collegamento tra la fase di intercettazione e le successive fasi di persistenza e sincronizzazione. Essa fornisce una rappresentazione strutturata su cui costruire analisi successive.

3.5 Strategia di persistenza e sincronizzazione

La gestione del ciclo di vita del dato non termina con la generazione del record aggregato. Una volta intercettate le informazioni utili a connessione terminata e creato il relativo record, il sistema sviluppa funzionalità di persistenza e sincronizzazione pensate per garantire robustezza, coerenza e funzionamento anche in condizioni di connettività non stabile.

Il ciclo di vita del dato, dall'intercettazione alla sincronizzazione remota, è illustrato in Figura 3.2.

Figura 3.2: Ciclo di vita del record dalla cattura del traffico alla sincronizzazione con il backend.

3.5.1 Cache persistente locale

I record generati non vengono immediatamente inviati al server remoto, ma vengono salvati inizialmente in un database locale, che funge da cache persistente dell'applicazione.

Si è scelto di separare la fase di raccolta dei dati dalla fase di sincronizzazione perché l'intercettazione del traffico è un'operazione delicata che deve rimanere leggera e non bloccante. Un invio diretto dei dati al server per ogni connessione avrebbe comportato un carico di rete costante e un rischio elevato di perdita delle informazioni in caso di instabilità di connettività. Il sistema non dipende dalla disponibilità immediata della rete o del server per poter continuare a funzionare correttamente.

La scelta di un approccio offline-first permette di:

- disaccoppiare intercettazione e sincronizzazione;
- garantire continuità della raccolta dati anche in assenza di connettività;
- evitare la perdita di informazioni in caso di errori di rete o crash dell'applicazione;
- ritentare l'invio dei dati non sincronizzati in un secondo momento.

Il database locale non rappresenta l'archivio definitivo, ma uno strato intermedio che assicura consistenza e affidabilità del flusso complessivo.

3.5.2 Sincronizzazione batch incrementale

La sincronizzazione con il backend avviene in modo asincrono e non bloccante. Il sistema verifica periodicamente la presenza di record non ancora sincronizzati e procede al loro invio in modalità batch.

L'invio batch consente di ridurre il numero di richieste HTTP, limitare il consumo energetico e ottimizzare l'utilizzo della rete. Solo i record non ancora trasmessi vengono selezionati per l'invio, evitando eventuali duplicati.

In caso di esito positivo lo stato di sincronizzazione del record viene aggiornato localmente. Se invece si verifica un errore (ad esempio indisponibilità del server), i dati rimangono nella cache locale e verranno ritrasmessi successivamente.

Il sistema prevede inoltre meccanismi di controllo per evitare sincronizzazioni concorrenti che potrebbero generare inconsistenze o duplicazioni. Il funzionamento del processo di sincronizzazione può essere riassunto nel seguente pseudo-codice 3.1.

Algoritmo 3.1 Sincronizzazione batch dei record locali

```
1:  procedure SyncPendingRecords
2:      if syncRunning = true then
3:          return
4:      end if
5:      syncRunning ← true
6:      BATCH_SIZE ← N
7:      if user is not authenticated then
8:          return ERROR
9:      end if
10:     pendingRecords ← repository.getPendingRecords(BATCH_SIZE)
11:     while pendingRecords is not empty do
12:         dtoBatch ← mapRecordsToDTO(pendingRecords)
13:         success ← backendClient.sendBatch(dtoBatch)
14:         if success then
15:             repository.markAsSynced(pendingRecords)
16:         else
17:             return ERROR
18:         end if
19:         pendingRecords ← repository.getPendingRecords(BATCH_SIZE)
20:     end while
21:     return SUCCESS
22: end procedure
```

L'Algoritmo 3.1 descrive il processo di sincronizzazione dei record memorizzati nel database locale verso il backend remoto.

Nelle linee 1–4 viene effettuato un controllo preliminare per evitare l'esecuzione concorrente di più processi di sincronizzazione. La variabile `syncRunning` agisce come meccanismo di mutua esclusione: se una sincronizzazione è già in corso, la procedura termina immediatamente. Questo accorgimento previene duplicazioni dei dati e possibili inconsistenze nello stato dei record.

Alle linee 5–6 viene inizializzato lo stato di sincronizzazione (`syncRunning`) e viene definita la dimensione del batch (`BATCH_SIZE`), che stabilisce il numero massimo di record inviati al backend in una singola richiesta. L'utilizzo di batch consente di ridurre il numero di richieste HTTP e migliorare l'efficienza della comunicazione.

Nelle linee 7–9 viene verificato che l'utente sia autenticato. In assenza di autenticazione la procedura viene interrotta, poiché il backend non sarebbe in grado di associare i record all'utente corretto. L'identificazione dell'utente avviene infatti tramite token JWT utilizzato nelle richieste verso il server.

Alla linea 10 il sistema recupera dal database locale un primo insieme di record non ancora sincronizzati, fino al limite definito dal batch.

Le linee 11–20 rappresentano il ciclo principale dell'algoritmo. Finché esistono record non sincronizzati, il sistema esegue le seguenti operazioni:

- i record locali vengono trasformati in oggetti DTO (Data Transfer Object), adatti alla trasmissione verso il backend;
- il batch di DTO viene inviato al server remoto tramite una richiesta HTTP autenticata;
- se l'invio ha esito positivo, i record corrispondenti vengono marcati come sincronizzati nel database locale.

In caso di errore durante la trasmissione (linee 16–17), la procedura viene interrotta e i record rimangono nella cache locale. Questo permette al sistema di ritentare la sincronizzazione in un momento successivo, mantenendo un modello di consistenza eventuale.

Al termine di ogni iterazione (linea 18-20), il sistema verifica nuovamente la presenza di record non sincronizzati nel database locale. Se il backlog non è vuoto, il ciclo continua con l'invio del batch successivo.

Infine, alla linea 21, la procedura restituisce un esito positivo quando tutti i record sono stati sincronizzati oppure quando non sono presenti ulteriori dati da inviare.

L'algoritmo presentato rappresenta una formalizzazione della logica di sincronizzazione implementata nel servizio `SyncService` dell'applicazione Android, respon-

sabile della gestione del trasferimento dei record dal database locale al backend remoto.

3.5.3 Identificazione tramite token e associazione server-side

L'applicazione mobile associa localmente ogni record a un identificativo univoco generato in modo pseudonimo, utilizzato esclusivamente a livello di cache persistente. La comunicazione con il backend è protetta tramite autenticazione basata su token JWT. Dopo il login iniziale viene rilasciato un token che viene utilizzato per autenticare le richieste successive.

Il client non trasmette esplicitamente l'identità dell'utente nei dati inviati. L'associazione tra record e utente avviene esclusivamente lato server, sulla base del token fornito nella richiesta autenticata.

In questo modo avviene la separazione dei dati tra utenti differenti, viene evitata l'esposizione di identificativi sensibili all'interno del payload e il server rimane stateless rispetto alla sessione applicativa.

3.5.4 Tolleranza ai guasti e coerenza tra livelli

La combinazione di cache locale persistente e sincronizzazione incrementale consente di realizzare un sistema tollerante ai guasti.

In caso di:

- assenza temporanea di connessione;
- riavvio dell'applicazione;
- errore nella comunicazione con il backend;

i dati rimangono salvati localmente e possono essere sincronizzati in un momento successivo, senza perdita di informazioni.

La coerenza tra livello locale e livello remoto è garantita dall'uso dello stato di sincronizzazione e dalla logica di aggiornamento successiva alla conferma di avvenuta ricezione. Nel complesso, la strategia adottata consente di mantenere indipendenti raccolta, persistenza e sincronizzazione, preservando stabilità operativa lato client e consistenza dei dati lato server.

3.6 Visualizzazione e analisi dei dati

La fase di visualizzazione dei dati rappresenta il collegamento tra la raccolta delle informazioni di rete e la loro interpretazione da parte dell'utente. Le scelte adottate in questa fase sono strettamente coerenti con il modello dati definito precedentemente.

Poiché il sistema utilizza la connessione aggregata come unità informativa principale, anche la visualizzazione fa riferimento a questa entità. Non viene mostrato l'insieme completo dei dati raccolti, ma solamente quelli più significativi e facilmente interpretabili dall'utente, che caratterizzano una determinata connessione o richiesta effettuata. Questa scelta consente di fornire una rappresentazione sintetica del traffico di rete, senza introdurre complessità non necessarie e in completa coerenza con il modello dati adottato.

La consultazione dei dati avviene secondo due modalità complementari:

- una lista ordinata, ovvero un elenco di connessioni aggregate, che consente di osservare i dettagli principali di ogni comunicazione;
- una modalità aggregata, che rielabora i record raccolti per ottenere rappresentazioni sintetiche sotto forma di grafici.

La differenza tra visualizzazione dettagliata e aggregata consente di distinguere l'analisi della singola connessione dall'analisi su distribuzioni e volumi complessivi.

Le elaborazioni necessarie alla generazione dei grafici vengono effettuate sui dati presenti nella cache persistente locale. Questa scelta è coerente col paradigma *offline-first* adottato dal sistema, poiché la lettura e l'analisi dei dati non dipendono dalla disponibilità del backend remoto. Di conseguenza, l'utente può accedere alle informazioni raccolte e alle relative statistiche anche in assenza di connettività.

Le operazioni di aggregazione non vengono effettuate a livello di interfaccia grafica, ma sono state delegate al livello di persistenza attraverso query strutturate sul database locale. Ciò riduce il carico computazionale lato UI e migliora la scalabilità del sistema.

A partire dai record delle connessioni aggregate, vengono calcolate alcune aggregazioni principali, tra cui:

- Distribuzione del traffico per protocollo;
- Applicazioni che generano il maggior volume di traffico;
- Distribuzione delle connessioni in base alla durata.

Con queste rappresentazioni è possibile ottenere una visione sintetica dei pattern di utilizzo della rete, senza dover eseguire elaborazioni complesse lato server.

La visualizzazione include inoltre un filtro temporale che consente di concentrarsi su uno specifico intervallo di dati. Il filtro è applicato selezionando un sottoinsieme dei record in base agli attributi temporali presenti nel modello dati.

Questa funzionalità permette di confrontare il comportamento di rete in diversi intervalli e costituisce una base per eventuali estensioni future orientate ad analisi spazio-temporali più approfondite.

Nel complesso, la strategia di visualizzazione adottata privilegia coerenza con il modello dati, facilità di interpretazione e indipendenza dal backend, mantenendo il sistema coerente a sostenibilità e compatibilità con il contesto mobile.

3.7 Privacy e conformità al GDPR

La progettazione del sistema è stata effettuata seguendo un approccio coerente con il principio di *privacy by design*, previsto dall'Articolo 25 del Regolamento (UE) 2016/679, noto come General Data Protection Regulation (GDPR) [22]. Sin dalle prime fasi sono state prese scelte architetturali orientate alla minimizzazione dei dati e alla protezione dell'identità dell'utente. Le riflessioni riguardanti la protezione delle informazioni non sono state introdotte come elemento successivo alla fase di sviluppo, ma hanno guidato le decisioni sulla definizione dell'architettura.

3.7.1 Minimizzazione e raccolta metadati

Il sistema raccoglie solo i metadati di rete, senza alcuna forma di ispezione del contenuto delle comunicazioni cifrate. La scelta di non intervenire sul payload delle richieste rappresenta una decisione consapevole e coerente con i vincoli tecnici imposti dalla piattaforma Android e con l'obiettivo di limitare il trattamento delle informazioni potenzialmente sensibili. In fase di trasmissione verso il backend remoto, il principio di minimizzazione viene nuovamente applicato. Le coordinate geografiche, qualora presenti e autorizzate dall'utente, vengono approssimate prima dell'invio. La precisione della posizione geografica è stata ridotta con l'obiettivo di limitare la possibilità di accedere alle coordinate precise dell'utente. Questa impostazione mantiene alto il valore analitico del dato e allo stesso tempo garantisce un livello adeguato di tutela della riservatezza.

3.7.2 Separazione tra modello locale e remoto

Un ulteriore elemento di tutela riguarda la distinzione tra modello di persistenza locale e modello di trasmissione verso il server remoto. Il database locale conserva le informazioni necessarie alla consultazione e all'analisi sul dispositivo, mentre il modello utilizzato per la comunicazione remota applica trasformazioni e riduzioni selettive degli attributi. Questa separazione consente di differenziare le politiche di trattamento tra uso locale e archiviazione centralizzata, riduce l'accoppiamento tra i due livelli e limita la circolazione di dati non strettamente necessari. L'identificazione dell'utente non avviene tramite l'invio esplicito di identificativi nel payload delle

richieste. L'associazione tra record e soggetto autenticato viene determinata lato server esclusivamente sulla base del token di autenticazione fornito nella richiesta. Il client non può dichiarare autonomamente la propria identità e il backend mantiene un'architettura stateless rispetto alla sessione applicativa. Le credenziali vengono trasmesse al backend tramite connessione HTTPS. Le password non vengono mai memorizzate in chiaro: durante la registrazione vengono sottoposte a hashing prima della persistenza nel database.

Il meccanismo di associazione tra record e utente autenticato è rappresentato in Figura 3.3.

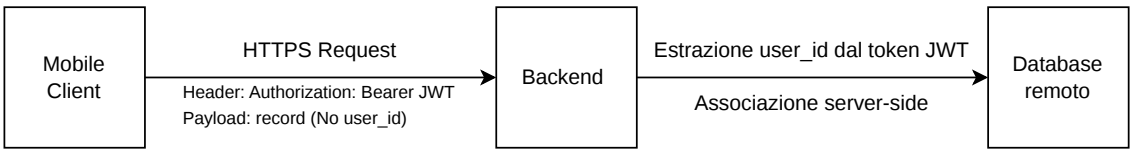


Figura 3.3: Associazione tra record e utente tramite token JWT, senza trasmissione esplicita dell'identità nel payload.

3.7.3 Diritti dell'utente e meccanismi implementati

Inoltre, il sistema offre meccanismi concreti per l'esercizio dei diritti riconosciuti dal Regolamento (UE) 2016/679 (GDPR) [22]. In particolare:

- l'utente può consultare in qualsiasi momento le connessioni registrate tramite l'interfaccia applicativa (diritto di accesso);
- è possibile esportare i dati raccolti in formato CSV, consentendo la portabilità delle informazioni (diritto alla portabilità);
- è disponibile un endpoint dedicato alla cancellazione dei dati associati all'utente, con eliminazione dalla cache locale e anche lato server (diritto all'oblio).

L'adozione di tali meccanismi è parte integrante della progettazione architetturale, e ha permesso di rispettare principi di trasparenza, controllo del dato e responsabilizzazione dell'utente.

Capitolo 4

Implementazione

In questo capitolo viene descritta l'implementazione concreta del sistema presentato nel Capitolo 3. Dopo aver illustrato le scelte architetturali e le motivazioni progettuali, si analizza come tali decisioni siano state effettivamente realizzate a livello di codice e quali componenti software siano stati sviluppati.

Il capitolo è organizzato in due parti principali. La prima è dedicata all'applicazione Android, che costituisce il punto di raccolta dei dati e il fulcro dell'interazione con l'utente. La seconda riguarda il backend remoto, responsabile dell'autenticazione, della ricezione dei dati, della loro persistenza su database e della gestione delle funzionalità legate alla privacy.

L'obiettivo di questa sezione non è ripetere le scelte metodologiche già discusse, ma mostrare nel dettaglio come l'architettura sia stata tradotta in classi, servizi e moduli concreti, evidenziando le principali soluzioni tecniche adottate durante lo sviluppo.

4.1 Implementazione lato Android

Prima di analizzare nel dettaglio la suddivisione architetturale, vengono descritte le principali tecnologie adottate per lo sviluppo dell'applicazione Android.

La Tabella 4.1 sintetizza lo stack tecnologico impiegato e il ruolo che ciascun componente svolge all'interno del sistema.

Tabella 4.1: Stack tecnologico lato Android

Categoria	**Tecnologia**	**Ruolo nel sistema**
Linguaggio	Kotlin	Linguaggio ufficiale per lo sviluppo Android, utilizzato per l'intera applicazione mobile.
Intercettazione traffico	VpnService	API ufficiale Android per la creazione di una VPN locale e l'intercettazione dei pacchetti IP in uscita.
Motore di cattura	Fork di PCAPdroid	Gestione del tunnel VPN e ricostruzione delle connessioni TCP/UDP senza implementare uno stack user-space personalizzato.
Persistenza locale	SQLite	Database embedded utilizzato come cache persistente secondo paradigma offline-first.
Networking	OkHttp	Client HTTP per comunicazione sicura con il backend remoto e invio batch dei dati.
Serializzazione JSON	Gson	Conversione tra oggetti Kotlin e formato JSON per la trasmissione dei dati.
Visualizzazione grafici	MPAndroidChart	Libreria per la rappresentazione grafica dei dati aggregati (protocollo, applicazioni, durata).

4.1.1 Struttura generale del progetto

L'applicazione Android è stata sviluppata in linguaggio Kotlin ed è organizzata su più livelli, in linea con l'architettura definita nel Capitolo 3. La struttura del progetto è stata pensata per separare in modo chiaro le diverse responsabilità, per evitare che componenti con ruoli differenti risultino eccessivamente dipendenti tra loro.

La suddivisione dei livelli implementati è riassunta nella Tabella 4.2.

Tabella 4.2: Suddivisione dei livelli dell'applicazione Android

Layer	**Classe principale**	**Responsabilità**
Capture	CaptureService	Inizializzazione e gestione del servizio VPN tramite VpnService e integrazione con il fork di PCAPdroid.
Domain	TrackerService	Costruzione del modello aggregato della connessione e coordinamento tra cattura e persistenza.
Persistence	TrackerRepository	Gestione del database SQLite locale e operazioni di inserimento, lettura e aggiornamento dei record.
Sync	SyncService	Sincronizzazione batch dei record non sincronizzati verso il backend remoto.
UI	Activity / Fragment	Visualizzazione dei dati raccolti e interazione con l'utente.

Questa suddivisione non è soltanto concettuale, ma si riflette concretamente nell'organizzazione dei package e delle classi del progetto. Ad ogni livello sono associate responsabilità specifiche, in modo da ridurre l'accoppiamento tra componenti e facilitare eventuali modifiche o estensioni future del sistema.

4.1.2 Integrazione con il motore di cattura

È stata effettuata un'analisi dell'architettura del fork di PCAPdroid, con l'obiettivo di individuare i punti più idonei per l'integrazione.

Le principali componenti analizzate sono state:

- **VpnService**: responsabile della creazione del tunnel VPN locale, unico meccanismo consentito da Android per intercettare il traffico generato da altre applicazioni;

- **CaptureService**: entry point del sistema di cattura e responsabile del ciclo di vita del servizio;
- **ConnectionsRegister**: componente incaricata della gestione dello stato delle connessioni attive e terminate;
- **ConnectionDescriptor**: struttura contenente i metadati associati a ciascuna connessione;
- **HTTPReassembly**: modulo dedicato alla ricostruzione delle richieste HTTP a partire dai chunk di payload;
- **AppsResolver**: utilizzato per la mappatura tra UID di sistema e nome dell'applicazione.

L'analisi ha permesso di comprendere il flusso interno dei dati e di individuare i punti del codice più adatti per intercettare i metadati necessari al sistema, mantenendo inalterato il funzionamento della pipeline originale di PCAPdroid.

L'avvio della cattura non avviene direttamente tramite `CaptureService`, ma è mediato dalla classe `TesiCaptureController`, introdotta come livello di coordinamento tra interfaccia utente e servizio VPN. Questa scelta evita che la UI interagisca direttamente con il servizio di sistema e consente di centralizzare la logica di avvio e arresto della cattura.

Il controller utilizza la classe `CaptureHelper`, già prevista dall'architettura di PCAPdroid, che rappresenta l'entry point corretto e supportato per l'inizializzazione della VPN. Il sistema Android richiede infatti che l'utente conceda esplicitamente il consenso tramite il metodo `VpnService.prepare()`, che può generare una richiesta di autorizzazione.

Tentativi di avvio diretto di `CaptureService`, senza passare dal meccanismo previsto dal framework, possono portare a stati incoerenti o a un'inizializzazione incompleta della pipeline interna di PCAPdroid. La sequenza di avvio gestita da `CaptureHelper` garantisce invece che tutte le verifiche preliminari, inclusa la gestione del consenso VPN e la preparazione delle impostazioni di cattura, vengano eseguite correttamente prima dell'attivazione del servizio.

Ottenuto il consenso, il servizio viene avviato con `startForegroundService()`, nel rispetto delle restrizioni introdotte dalle versioni recenti di Android per l'esecuzione dei servizi in background.

La verifica dello stato della cattura non viene gestita tramite flag locali all'interno dell'interfaccia utente, ma interrogando direttamente lo stato reale del servizio VPN. Questa scelta evita inconsistenze nel caso in cui il sistema operativo interrompa o riavvii il servizio, garantendo coerenza tra lo stato effettivo della cattura e quello visualizzato all'utente.

All'interno del metodo `onCreate()` di `CaptureService` è stato inserito il primo punto di integrazione con il modulo sviluppato nella tesi, mediante l'invocazione di `TrackerService.init()`. In questa fase vengono inizializzati:

- il repository per l'accesso al database SQLite locale;
- un identificativo anonimo dell'utente;
- il servizio di sincronizzazione batch verso il backend remoto.

L'inizializzazione avviene all'avvio del servizio VPN e non a livello di Activity, in modo da rendere il sistema di tracciamento indipendente dall'interfaccia grafica e coerente con il ciclo di vita del servizio di cattura.

La logica di intercettazione rimane quindi separata dalla logica di persistenza e sincronizzazione del sistema.

4.1.3 Intercettazione delle connessioni e logging finale

Dopo aver individuato il punto corretto per l'avvio del servizio VPN, l'attenzione si è concentrata sull'identificazione del momento più opportuno per intercettare e registrare le informazioni relative alle connessioni di rete.

L'analisi della classe `ConnectionsRegister` ha evidenziato che le connessioni vengono gestite tramite una struttura circolare (*ring buffer*). Internamente il registro mantiene un array di oggetti `ConnectionDescriptor` di dimensione fissa, nel quale ogni elemento rappresenta lo stato corrente di una specifica comunicazione di rete.

Le nuove connessioni vengono inserite nel buffer tramite il metodo `newConnections()`, che aggiunge un nuovo descrittore nella posizione corrente del buffer. Quando la capacità massima del buffer viene raggiunta, le nuove connessioni sovrascrivono progressivamente le più vecchie secondo una logica circolare.

Gli aggiornamenti dello stato delle connessioni (ad esempio byte trasferiti, numero di pacchetti o stato della connessione) vengono gestiti dal metodo `connectionsUpdates()`, che modifica il `ConnectionDescriptor` già presente nel buffer.

Algoritmo 4.1 Gestione delle connessioni nel registro circolare

```
1: procedure NewConnections(conns)
2:     for all conn in conns do
3:         insert conn into ring_buffer at position tail
4:         tail ← (tail + 1) mod buffer_size
5:     end for
6: end procedure
7: procedure ConnectionsUpdates(updates)
8:     for all update in updates do
9:         conn ← find connection in ring_buffer by id
10:        apply update to conn
11:        if conn.status ≠ ACTIVE and conn.loggedFinal = false then
12:            conn.loggedFinal ← true
13:            TrackerService.logFinalConnection(conn)
14:        end if
15:    end for
16: end procedure
```

La presenza di un meccanismo di rollover implica che le connessioni più vecchie possano essere sovrascritte una volta raggiunta la capacità massima del buffer. Per questo motivo è necessario registrare i metadati della connessione al momento della sua chiusura, prima che il relativo descrittore possa essere eventualmente riutilizzato dal sistema.

In una prima fase era stato valutato il logging al momento della creazione della connessione, all'interno di `newConnections()`. Tuttavia, in questa fase i metadati risultano ancora parziali: byte trasferiti, durata e numero di pacchetti non sono definitivi. Per questo motivo tale soluzione è stata scartata.

È stato quindi individuato come punto di integrazione il metodo `connectionsUpdates()`, che viene invocato dal `CaptureService` ogni volta che lo stato di una connessione cambia. In particolare, il logging viene eseguito esclusivamente quando la connessione non si trova più nello stato `STATUS_ACTIVE`, ovvero alla sua chiusura definitiva.

La scelta di registrare la connessione solo al termine del suo ciclo di vita consente di ottenere un record completo e consistente, comprendente:

- timestamp di inizio e fine comunicazione;
- durata complessiva;
- byte trasmessi e ricevuti;
- numero di pacchetti inviati e ricevuti;
- indirizzo IP e porta di destinazione;
- applicazione associata (UID e nome).

Per evitare duplicazioni dovute a eventuali aggiornamenti multipli dello stato, è stato introdotto un flag booleano (`loggedFinal`) all'interno del `ConnectionDescriptor`. Il logging viene eseguito solo se la connessione non è più attiva e il flag non è ancora impostato. Una volta registrata, la connessione viene marcata come già elaborata.

Il recupero del nome dell'applicazione avviene tramite `AppsResolver`, che consente la mappatura tra UID di sistema e descrizione leggibile dell'app. I metadati finali vengono quindi passati al metodo `TrackerService.logFinalConnection()`, che si occupa della costruzione del record completo e della successiva persistenza su database locale.

Questa integrazione consente di mantenere separata la logica di intercettazione (gestita dal fork di PCAPdroid) dalla logica di tracciamento e memorizzazione (gestita dal modulo sviluppato nella tesi), limitando le modifiche al codice originale e preservandone il funzionamento.

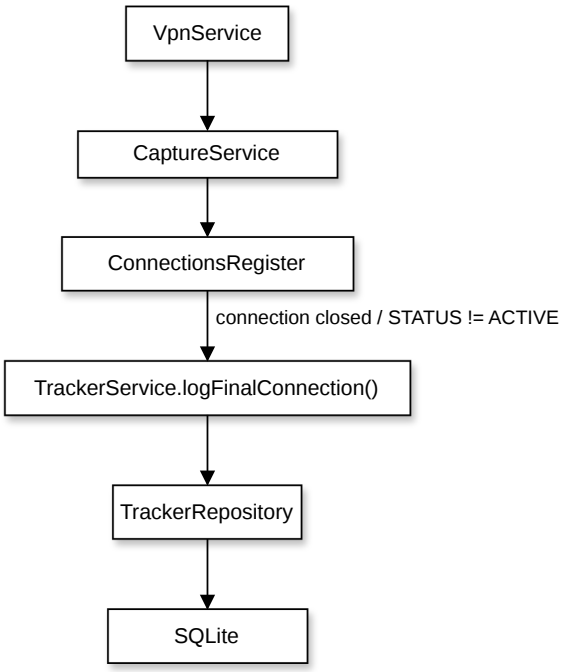


Figura 4.1: Flusso di intercettazione e registrazione di una connessione alla sua chiusura.

4.1.4 Estrazione delle richieste HTTP

Oltre al tracciamento dei metadati generali delle connessioni di rete, è stata introdotta una fase aggiuntiva di analisi a livello applicativo per intercettare informazioni relative alle richieste HTTP in chiaro.

A tal fine è stata analizzata la classe `HTTPReassembly` del fork di PCAPdroid, responsabile della ricostruzione dei messaggi HTTP a partire dai chunk di payload intercettati dal tunnel VPN. Questa componente si occupa di:

- ricostruire gli header HTTP a partire dai dati contenuti nel payload dei pacchetti intercettati;
- decodificare eventuali contenuti compressi indicati nell'header `Content-Encoding`;
- ricostruire il messaggio HTTP completo a partire da più segmenti di payload TCP, nel caso in cui header e corpo della richiesta o della risposta siano distribuiti su più pacchetti di rete.

La Figura 4.2 mostra la struttura generale di una richiesta HTTP, composta dalla request line, da una serie di header e da un eventuale corpo del messaggio.

Request line	`GET /index.html HTTP/1.1`

Headers	`Host: www.example.com`
	`User-Agent: Mozilla/5.0`
	`Accept: text/html`
	`Accept-Language: en-US`

Body (optional)	[optional data]

Figura 4.2: Struttura generale di una richiesta HTTP.

L'obiettivo dell'integrazione non era modificare il meccanismo di ricostruzione, ma introdurre un punto di estrazione dei metadati applicativi utili alla tesi, in particolare:

- metodo HTTP (GET, POST, PUT, HEAD);
- host della richiesta;
- path della risorsa richiesta.

Parsing della request line

Durante l'elaborazione del primo chunk di una richiesta HTTP, la classe `HTTPReassembly` analizza la cosiddetta *request line*. In questa fase sono stati aggiunti campi dedicati alla memorizzazione del metodo (`mMethod`) e del percorso richiesto (`mPath`).

Il path viene normalizzato rimuovendo eventuali parametri di query (stringhe successive al carattere "?"), al fine di:

- evitare la memorizzazione di parametri dinamici o potenzialmente sensibili;
- ridurre la variabilità dei dati raccolti;
- mantenere un livello coerente con gli obiettivi di minimizzazione dei dati.

Parsing degli header e campo Host

Durante la lettura degli header viene intercettato il campo `Host`, dal quale viene estratto il dominio della richiesta. È stato introdotto un ulteriore campo interno (`mHost`) per conservarne il valore.

Nel momento in cui risultano disponibili metodo, host e path, e solo per traffico in uscita (`is_sent`), viene invocato il metodo:

```
TrackerService.onHttpRequest()
```

che memorizza temporaneamente tali informazioni.

Per evitare registrazioni multiple della stessa richiesta, considerando che l'elaborazione può avvenire su più chunk, è stato introdotto un flag booleano (`mHttpRequestLogged`) che garantisce l'esecuzione del logging una sola volta per richiesta.

Limiti legati al traffico HTTPS

È importante sottolineare che questa forma di estrazione è possibile esclusivamente per traffico HTTP non cifrato. Nel caso di connessioni HTTPS, il contenuto applicativo risulta cifrato a livello TLS e non è accessibile tramite il tunnel VPN locale senza tecniche di man-in-the-middle, che non rientravano negli obiettivi del progetto.

Di conseguenza, per connessioni HTTPS vengono registrati soltanto i metadati disponibili a livello di connessione (indirizzo IP, porta, durata, byte trasferiti), mentre metodo e path restano non valorizzati.

Integrazione con il logging finale

Le informazioni raccolte a livello HTTP non vengono salvate direttamente su database, ma vengono temporaneamente mantenute nel `TrackerService`. Al momento della chiusura definitiva della connessione (come descritto nella Sezione 4.1.3), tali dati vengono associati al record finale, se il protocollo risulta HTTP.

Questa soluzione consente di mantenere separati:

- il livello di ricostruzione del payload (gestito da PCAPdroid);
- il livello di persistenza e tracciamento (gestito dal modulo sviluppato nella tesi).

L'integrazione risulta quindi non invasiva rispetto al codice originale e coerente con la separazione delle responsabilità definita nell'architettura del sistema.

4.1.5 TrackerService e livello di dominio

Il `TrackerService` costituisce il punto di raccordo tra il sistema di intercettazione del traffico e l'architettura applicativa sviluppata nella tesi.

Dal punto di vista architetturale, esso implementa il *Domain Layer*: riceve i metadati provenienti dal livello di cattura, li trasforma in un modello strutturato e ne delega la persistenza al repository locale.

Implementazione e inizializzazione

Il servizio è implementato come `object` Kotlin, garantendo un'unica istanza globale e un punto centralizzato di raccolta delle connessioni.

L'inizializzazione avviene nel metodo `init()`, invocato all'avvio del servizio VPN, e comprende:

- istanziazione del `TrackerRepository`;
- generazione di un identificativo anonimo utente (UUID);
- inizializzazione del `LocationService`;
- avvio del `SyncService` con meccanismo di retry automatico.

Questa scelta rende il sistema di tracciamento indipendente dall'interfaccia grafica e coerente con il ciclo di vita del servizio di cattura.

Registrazione alla chiusura della connessione

Il metodo `logFinalConnection()` viene invocato esclusivamente alla chiusura definitiva della connessione, quando tutte le metriche risultano complete.

In tal modo è possibile garantire:

- coerenza dei dati memorizzati;
- assenza di duplicazioni;
- riduzione delle scritture su database;
- robustezza rispetto al meccanismo di rollover del buffer interno di PCAPdroid.

Costruzione del modello aggregato

Alla ricezione dei metadati finali, il servizio costruisce un oggetto `NetworkRequestRecord`, che rappresenta una connessione aggregata conclusa. Il modello include:

- identificazione dell'applicazione (nome e UID);
- protocollo;
- dominio o indirizzo IP di destinazione;
- byte trasmessi e ricevuti;
- numero di pacchetti;
- timestamp di inizio e fine;
- durata complessiva;
- coordinate GPS opzionali;
- stato di sincronizzazione (`synced`).

La geolocalizzazione, ottenuta tramite `LocationService`, è opzionale e non condiziona la registrazione della connessione.

Integrazione delle informazioni HTTP

Nel caso di traffico HTTP non cifrato, le informazioni applicative (metodo, host e path) intercettate in `HTTPReassembly` vengono temporaneamente memorizzate nel servizio e associate al record finale al momento della chiusura della connessione.

Per traffico HTTPS tali campi restano non valorizzati, coerentemente con le limitazioni imposte dalla cifratura TLS.

Attivazione della sincronizzazione

Dopo l'inserimento nel database locale, il servizio verifica il numero di record non sincronizzati. Al superamento della soglia configurata (`SYNC_THRESHOLD`), viene avviata una sincronizzazione batch verso il backend remoto.

Questo meccanismo implementa un approccio offline-first, riducendo l'overhead di rete e migliorando l'efficienza complessiva del sistema.

4.1.6 Modello dati e persistenza locale

Come descritto nel Capitolo 3, l'unità informativa fondamentale del sistema è la connessione aggregata conclusa. A livello implementativo, tale entità è rappresentata dalla classe `NetworkRequestRecord`, che costituisce il modello dati centrale dell'applicazione.

Struttura del modello

La classe `NetworkRequestRecord` è definita come `data class` Kotlin e raccoglie in un'unica struttura:

- metadati di identificazione dell'utente (`userUuid`);
- informazioni sull'applicazione che ha generato la connessione (nome e UID);
- parametri di rete (protocollo, dominio, indirizzi IP e porte);
- metriche di traffico (byte e pacchetti trasmessi e ricevuti);
- informazioni temporali (timestamp di inizio e fine, durata);
- coordinate geografiche opzionali;
- attributi HTTP non cifrati (metodo, path e host), disponibili solo in caso di traffico HTTP.

Il modello include inoltre due campi legati alla gestione della cache locale:

- `id`, chiave primaria autoincrementale generata dal database;
- `synced`, utilizzato per indicare se il record è già stato sincronizzato con il backend remoto.

Questa struttura unificata rappresenta l'evoluzione rispetto a una versione iniziale del sistema, nella quale connessioni e richieste HTTP venivano memorizzate in tabelle separate. L'adozione di un modello aggregato ha semplificato la correlazione tra livelli di trasporto e applicativo e ha reso più lineare la fase di sincronizzazione.

Struttura del database SQLite

La persistenza locale è realizzata tramite SQLite, attraverso la classe `TesiDbHelper`, che estende `SQLiteOpenHelper`.

Il database locale, denominato `tesi.db`, contiene una tabella principale chiamata `network_requests`. Ogni riga rappresenta una connessione conclusa intercettata tramite il servizio VPN.

La tabella include:

- chiave primaria autoincrementale (`id`);
- campi testuali e numerici coerenti con gli attributi del modello;
- campo `synced` con valore di default pari a 0.

Il campo `synced` assume valore:

- 0 per record non ancora inviati al backend;
- 1 per record sincronizzati correttamente.

La gestione della versione del database è affidata al parametro `DB_VERSION`. In caso di aggiornamento dello schema, il database viene ricreato tramite eliminazione della tabella esistente. Questa scelta è coerente con il ruolo del database locale come cache persistente e non come archivio definitivo dei dati.

Repository e separazione dell'accesso ai dati

L'accesso al database è incapsulato nella classe `TrackerRepository`, che implementa il pattern Repository separando:

- la logica di persistenza (SQLite);
- la logica di dominio (`TrackerService`);
- il livello di interfaccia utente.

Il metodo principale di inserimento è `insertNetworkRequest()`, che converte un oggetto `NetworkRequestRecord` in una struttura `ContentValues` e lo inserisce nella tabella `network_requests`.

Oltre alle operazioni di inserimento, il repository fornisce:

- recupero dei record non sincronizzati;
- aggiornamento dello stato di sincronizzazione;
- conteggio dei record presenti;
- eliminazione completa dei dati locali;
- query aggregate per la generazione dei grafici.

Le operazioni di aggregazione (ad esempio traffico per protocollo o per applicazione) vengono eseguite direttamente tramite query SQL. Questa scelta riduce il carico computazionale sul livello UI e mantiene la logica di elaborazione dei dati all'interno del livello di persistenza.

Ruolo del database come cache persistente

È importante sottolineare che il database locale non rappresenta l'archivio definitivo delle informazioni, ma una cache persistente coerente con il modello offline-first adottato dal sistema.

I dati vengono salvati immediatamente al termine della connessione, indipendentemente dalla disponibilità del backend remoto. La presenza del campo `synced` consente di distinguere tra dati già trasmessi e dati ancora da sincronizzare, garantendo coerenza tra livello locale e remoto.

In questo modo la persistenza locale rimane disaccoppiata dalla connettività di rete, preservando stabilità e continuità operativa del sistema. Il database locale è accessibile esclusivamente dall'applicazione stessa e non da terze parti.

4.1.7 Sincronizzazione offline-first

La sincronizzazione dei dati verso il backend remoto è gestita dalla classe `SyncService`.

Sincronizzazione a batch

Il metodo principale `syncOnce()` esegue una singola operazione completa di sincronizzazione:

- verifica che non sia già in corso un'altra sincronizzazione;
- controlla che l'utente sia autenticato;
- recupera un batch di record non sincronizzati dal database locale;
- converte i record in formato DTO;
- invia il batch al backend remoto;
- marca come sincronizzati i record inviati con successo.

Controllo della concorrenza

Per evitare sincronizzazioni parallele, il servizio utilizza una variabile condivisa (`isSyncRunning`) che impedisce l'avvio di nuove operazioni finché una sincronizzazione è in corso. Questa scelta previene race condition e possibili duplicazioni nell'invio dei dati.

Gestione degli errori e consistenza

In caso di errore di rete o risposta negativa del server, il servizio interrompe il ciclo di sincronizzazione e mantiene i record nello stato `synced = 0`.

L'aggiornamento del campo `synced` a 1 avviene esclusivamente dopo la conferma di ricezione da parte del backend. Questo meccanismo garantisce consistenza tra livello locale e remoto e consente di ritentare l'invio in un secondo momento senza perdita di informazioni.

Retry automatico su disponibilità di rete

Il servizio implementa inoltre un meccanismo di retry automatico basato su `ConnectivityManager`.

Attraverso la registrazione di un `NetworkCallback`, il sistema intercetta il momento in cui una connessione Internet diventa disponibile. Se esistono record non sincronizzati e non è già in corso una sincronizzazione, viene avviato automaticamente un nuovo tentativo di invio.

4.1.8 Comunicazione e autenticazione

La comunicazione con il backend remoto avviene tramite API REST protette da autenticazione JWT.

L'implementazione è suddivisa in tre componenti: `NetworkRequestMapper`, che converte i modelli locali in oggetti *DTO* (*Data Transfer Object*), ovvero strutture dati semplificate utilizzate esclusivamente per la trasmissione delle informazioni tra client e server; `BackendClient`, che gestisce l'invio delle richieste HTTP; e `SessionManager`, che si occupa della gestione persistente del token di autenticazione.

Separazione tra modello persistente e DTO

Il modello `NetworkRequestRecord`, utilizzato per la cache locale, non viene inviato direttamente al backend.

Prima della trasmissione, ogni record viene convertito in un oggetto `NetworkRequestDto` tramite la classe `NetworkRequestMapper`.

Nel DTO non è presente l'identificativo utente (`userUuid`), poiché l'associazione tra record e utente avviene esclusivamente lato server.

Invio batch

I record convertiti in DTO vengono incapsulati in un oggetto `BatchDto`, che rappresenta il corpo della richiesta JSON inviata al backend.

Il client responsabile dell'invio è la classe `BackendClient`, che utilizza la libreria OkHttp per:

- serializzare il batch in formato JSON;
- costruire la richiesta HTTP;
- aggiungere automaticamente l'header `Authorization: Bearer <token>`;
- gestire le risposte del server.

In caso di risposta HTTP 401, il token viene considerato non valido e l'utente viene automaticamente disconnesso.

Gestione della sessione

La gestione del token JWT è delegata alla classe `SessionManager`, che utilizza `SharedPreferences` per salvare e recuperare il token in modo persistente.

Le operazioni di autenticazione (login e registrazione) sono gestite dalla classe `AuthClient`, che comunica con gli endpoint `/auth/login` e `/auth/register`. In caso di login riuscito, il token restituito dal backend viene memorizzato localmente.

Il `SessionManager` salva e recupera il token JWT da `SharedPreferences`.

4.1.9 Privacy e minimizzazione dei dati

Le scelte implementative relative alla gestione dei dati riflettono i vincoli descritti nei capitoli precedenti e si traducono in soluzioni tecniche concrete all'interno dell'applicazione.

Limitazione al livello di metadato

Il sistema registra esclusivamente informazioni disponibili a livello di connessione: indirizzi IP, porte, durata, byte e pacchetti scambiati.

Nel caso di traffico HTTPS non viene effettuata alcuna analisi del contenuto applicativo. Metodo e path HTTP risultano valorizzati solo per traffico non cifrato, mentre per connessioni TLS rimangono nulli.

Trasformazione dei dati prima dell'invio

I record persistiti localmente (`NetworkRequestRecord`) vengono convertiti in oggetti `NetworkRequestDto` tramite la classe `NetworkRequestMapper` prima dell'invio al backend.

Il DTO include esclusivamente i campi necessari alla trasmissione e non contiene identificativi utente. L'associazione tra dati e utente avviene esclusivamente lato server tramite il token JWT presente nell'header `Authorization`.

Geolocalizzazione

La geolocalizzazione è opzionale e non blocca la registrazione delle connessioni.

Nel `NetworkRequestMapper`, le coordinate vengono arrotondate a due decimali prima dell'invio, riducendo la precisione spaziale rispetto al dato originale memorizzato localmente.

Anonimizzazione tramite hashing

È stata implementata una utility (`HashUtils`) basata su SHA-256 con salt statico, utilizzabile per l'anonimizzazione di campi quali dominio o indirizzo IP prima della trasmissione al backend.

L'integrazione è centralizzata nel livello di mapping, consentendo l'attivazione di politiche di hashing senza modifiche al modello di persistenza locale.

Esportazione e cancellazione dei dati

L'utente può esportare i propri dati tramite l'endpoint:

```
GET /network-requests/me/data/export
```

La descrizione completa delle API è riportata nella Tabella 4.5 (Sezione 4.2.5).

Il file CSV restituito dal backend viene salvato nella directory *Download* tramite `MediaStore`.

La cancellazione completa dei dati può essere richiesta tramite:

```
DELETE /network-requests/me/data
```

Anche questo endpoint è documentato nella Tabella 4.5.

In caso di esito positivo, i dati vengono eliminati sia dal backend sia dalla cache locale SQLite tramite `clearAllNetworkRequests()`.

L'applicazione consente inoltre la consultazione diretta delle connessioni raccolte attraverso l'interfaccia utente, permettendo all'utente di visualizzare i dati memorizzati localmente.

Informativa iniziale

Al primo avvio viene mostrato un messaggio informativo relativo all'utilizzo del servizio VPN locale e all'invio dei dati al backend. L'utente deve confermare la presa visione prima di proseguire.

4.1.10 Interfaccia utente

L'interfaccia utente rappresenta il livello attraverso il quale l'utente interagisce con il sistema e costituisce il punto di accesso alle funzionalità di cattura, visualizzazione e gestione dei dati raccolti.

La progettazione della UI è stata orientata a due obiettivi principali: da un lato garantire semplicità d'uso e chiarezza informativa, dall'altro mantenere una netta separazione tra presentazione e logica applicativa. Le Activity non implementano logiche di business complesse, ma si limitano a coordinare i servizi sottostanti, nel rispetto dell'architettura multilivello descritta nelle sezioni precedenti.

Struttura generale L'applicazione è organizzata in più schermate, ciascuna con una responsabilità ben definita.

All'avvio, una Activity di ingresso verifica la presenza di un token JWT salvato e indirizza l'utente verso la schermata di login o verso la dashboard principale. Le schermate di autenticazione consentono la registrazione e l'accesso, delegando la comunicazione con il backend ai componenti dedicati.

La *MainActivity* costituisce la dashboard principale dell'applicazione e rappresenta il punto centrale di orchestrazione dei servizi. Pur non contenendo logica di dominio, coordina l'interazione tra interfaccia utente, servizio di cattura, repository locale e componenti di comunicazione con il backend.

Nell'`onCreate` vengono:

- inizializzati il `SessionManager` e il `TesiCaptureController`;
- mostrato un avviso informativo al primo avvio, che dichiara i fini e le modalità dell'applicazione;
- richiesti eventuali permessi (notifiche e localizzazione);
- configurati i listener associati ai controlli dell'interfaccia.

L'aggiornamento dello stato della cattura avviene verificando periodicamente la presenza della VPN attiva (`TRANSPORT_VPN`), tramite `ConnectivityManager`.

Dalla dashboard l'utente può quindi:

- avviare o arrestare la cattura del traffico;
- visualizzare info-panel della sincronizzazione;
- accedere alla lista delle connessioni raccolte;
- esportare i propri dati in formato CSV;
- richiedere la cancellazione completa dei dati;
- effettuare il logout.

La Figura 4.3 mostra la dashboard principale dell'applicazione durante una sessione attiva di cattura.

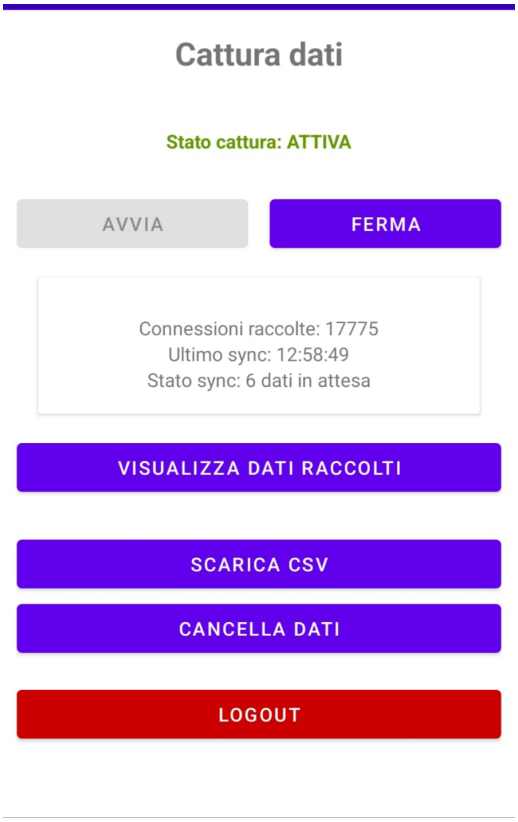


Figura 4.3: Dashboard principale dell'applicazione Android.

La schermata non interagisce direttamente con il database né con il servizio VPN, ma delega le operazioni ai componenti dedicati, mantenendo una chiara separazione delle responsabilità.

Visualizzazione delle connessioni La visualizzazione dei dati raccolti avviene tramite la *TesiDataActivity*, che utilizza una *RecyclerView* per mostrare le connessioni aggregate memorizzate nel database locale.

Per garantire reattività dell'interfaccia, vengono caricati esclusivamente gli ultimi record disponibili, evitando di trasferire in memoria l'intero contenuto del database. Ogni riga della lista rappresenta una connessione conclusa e riporta informazioni quali applicazione sorgente, protocollo, destinazione e volume di traffico scambiato.

La Figura 4.4 mostra un esempio della schermata di consultazione delle connessioni intercettate.

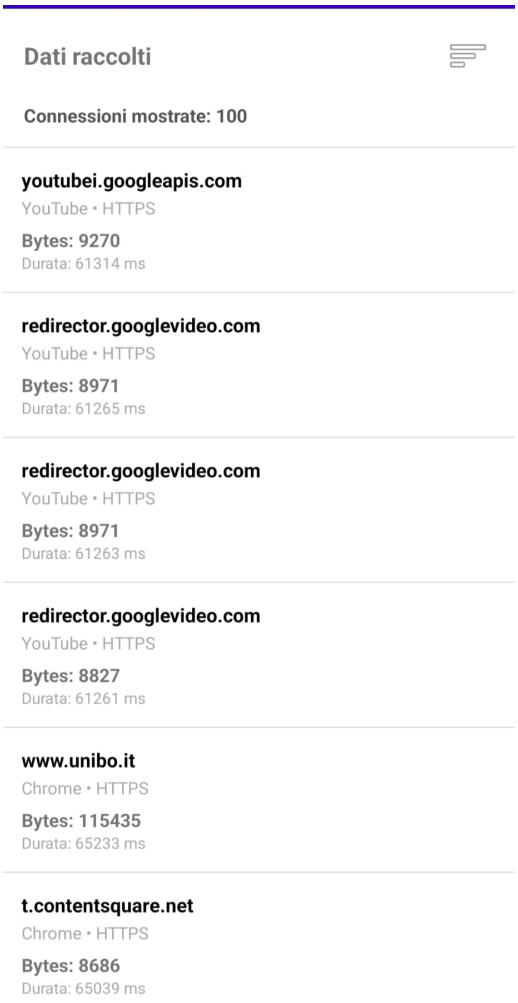


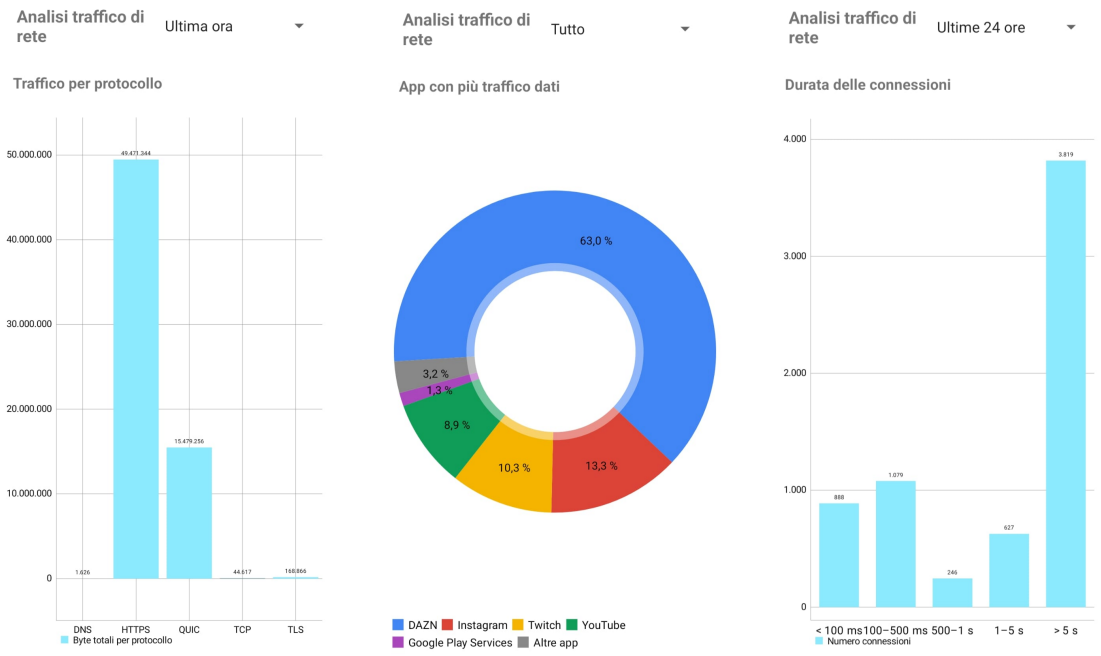
Figura 4.4: Schermata di visualizzazione delle connessioni aggregate.

L'adapter associato alla RecyclerView si limita alla presentazione dei dati e non esegue alcuna operazione di persistenza o sincronizzazione. In questo modo la UI rimane un livello puramente descrittivo.

Dal punto di vista funzionale, questa schermata realizza concretamente il diritto di accesso ai dati: l'utente può consultare in ogni momento le informazioni raccolte localmente, senza dipendere dalla disponibilità del backend remoto.

Analisi grafica Accanto alla lista testuale, l'applicazione permette di accedere a una sezione dedicata all'analisi grafica dei dati. La *TesiAnalysisActivity* ospita tre grafici distinti, navigabili tramite *ViewPager2* e *TabLayout*.

La Figura 4.5 mostra i tre grafici principali dell'applicazione, relativi rispettivamente al traffico aggregato per protocollo, alla ripartizione del traffico per applicazione e alla distribuzione delle connessioni in base alla durata.



(a) Traffico aggregato per protocollo

(b) Applicazioni con maggior traffico

(c) Distribuzione delle connessioni per durata

Figura 4.5: Sezione di analisi grafica.

I grafici sono implementati tramite la libreria MPAndroidChart e ricevono dati già aggregati dal repository locale. Le operazioni di calcolo e raggruppamento avvengono infatti lato database, mentre i fragment grafici si occupano esclusivamente della rappresentazione visiva.

È inoltre previsto un filtro temporale che consente di limitare l'analisi a un determinato intervallo di tempo. Il filtro viene selezionato dall'utente e notificato ai fragment tramite un'interfaccia dedicata, evitando accoppiamenti diretti tra Activity e componenti grafici.

Gestione dei diritti dell'utente L'interfaccia integra direttamente le funzionalità necessarie a garantire i diritti previsti dalla normativa europea in materia di protezione dei dati.

L'utente può esportare i propri dati in formato CSV, esercitando il diritto alla portabilità. Il file viene scaricato dal backend remoto e salvato nella cartella *Download* del dispositivo utilizzando le API ufficiali di Android, nel rispetto del modello di sicurezza *Scoped Storage*.

È inoltre possibile richiedere la cancellazione completa dei dati. In caso di esito positivo, i dati vengono eliminati sia dal backend sia dalla cache locale, assicurando coerenza tra i livelli di persistenza.

Al primo avvio dell'applicazione viene mostrato un messaggio informativo che descrive l'utilizzo del servizio VPN locale e le modalità di trattamento dei dati, introducendo un primo livello di trasparenza e consenso esplicito.

4.2 Implementazione lato Backend

4.2.1 Architettura generale del backend

Il backend è implementato come server REST sviluppato con NestJS e scritto in TypeScript. Espone un insieme di API HTTP utilizzate dall'applicazione Android per l'autenticazione degli utenti e per la trasmissione dei dati raccolti.

Dal punto di vista funzionale, il backend svolge quattro compiti principali:

- autenticazione e registrazione degli utenti;
- ricezione di richieste di rete in modalità batch;
- gestione dei dati associati all'utente autenticato (recupero, esportazione, cancellazione);
- persistenza su database PostgreSQL.

Il sistema è progettato come componente *stateless*. Il server non mantiene sessioni lato backend: ogni richiesta autenticata include un token JWT nell'header `Authorization`. L'identità dell'utente viene estratta esclusivamente dal token validato lato server e non da parametri forniti dal client.

L'architettura interna segue una separazione netta delle responsabilità:

- **Controller** gestiscono il livello HTTP (routing, estrazione dei parametri, impostazione delle risposte);
- **Service** implementano la logica applicativa;
- **Prisma** fornisce l'accesso al database e incapsula le operazioni di persistenza.

I controller non accedono direttamente al database e non contengono logica di dominio. I service non dipendono dal trasporto HTTP, ma operano su oggetti tipizzati (DTO) e delegano la persistenza a Prisma tramite `PrismaService`.

L'autenticazione è basata su JSON Web Token (JWT). La validazione del token è centralizzata tramite una strategia dedicata (`JwtStrategy`) e un guard (`JwtAuthGuard`) applicato agli endpoint protetti. In questo modo l'associazione tra dato e utente avviene esclusivamente lato server, senza accettare identificativi utente provenienti dal client.

Nel complesso, il backend costituisce un componente indipendente dall'applicazione Android: non intercetta traffico, non elabora logiche di cattura e non assume conoscenze specifiche del client. Riceve esclusivamente dati strutturati tramite API REST e li memorizza in modo coerente e sicuro sul database relazionale.

L'organizzazione modulare del backend e il flusso di una richiesta autenticata sono rappresentati in Figura 4.6.

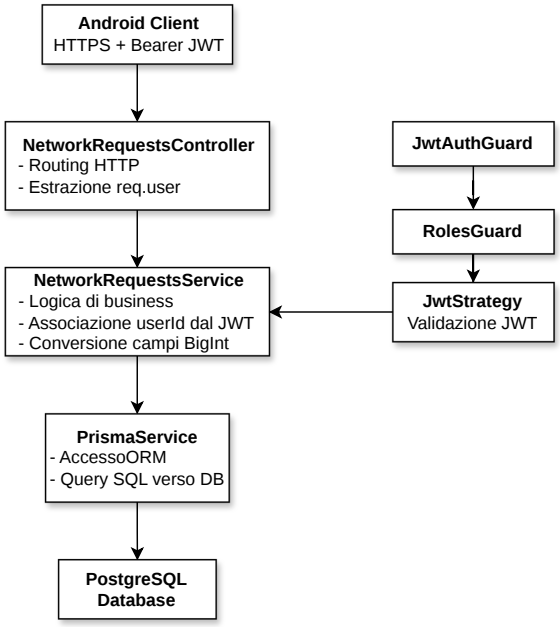


Figura 4.6: Architettura modulare del backend con separazione tra livello HTTP, logica applicativa, accesso ai dati tramite Prisma, meccanismi di autenticazione JWT e controllo RBAC.

Componenti principali del backend La Tabella 4.3 riassume i principali componenti dell'architettura backend e il ruolo che ciascun modulo svolge all'interno del sistema.

Tabella 4.3: Principali componenti dell'architettura backend

Componente	**Modulo**	**Responsabilità principale**
AppModule	root	Modulo radice che importa PrismaModule, AuthModule e NetworkRequestsModule e collega l'applicazione come sistema unico.
main.ts	root	Punto di ingresso dell'applicazione NestJS; avvia il server HTTP e inizializza il modulo principale.
PrismaModule	prisma	Espone PrismaService tramite Dependency Injection per l'accesso centralizzato al database.
PrismaService	prisma	Estende PrismaClient, gestisce connessione al database PostgreSQL e retry in fase di avvio.
schema.prisma	prisma	Definizione del modello dati, delle relazioni e dell'enum Role per il database PostgreSQL.
AuthController	auth	Espone gli endpoint di autenticazione (login, register) e delega la logica a AuthService.
AuthService	auth	Gestisce autenticazione, hashing password (bcrypt), generazione JWT e validazione credenziali.
JwtStrategy	auth	Valida il token JWT e popola req.user con userId e ruolo.
JwtAuthGuard	auth	Protegge gli endpoint verificando validità e presenza del token JWT.
RolesDecorator	common/decorators	Definisce i ruoli richiesti per un endpoint tramite metadata (@Roles).
RolesGuard	common/guards	Implementa il controllo RBAC verificando il ruolo presente nel JWT.
NetworkRequestsController	network-requests	Espone gli endpoint REST per upload batch, recupero dati, export CSV e cancellazione GDPR.

Componente	Modulo	Responsabilità principale
NetworkRequestsService	network-requests	Implementa la logica di business: salvataggio batch, conversione BigInt, export CSV, cancellazione dati.
BatchNetworkRequestDto	network-requests/dto	Valida struttura e formato del batch ricevuto dal client.
CreateNetworkRequestDto	network-requests/dto	Definisce e valida i campi di una singola richiesta di rete.
seed.ts	auth	Script iniziale per creazione utenti USER e ADMIN tramite upsert.

4.2.2 Entry point e struttura modulare

main.ts

Il file `main.ts` costituisce il punto di ingresso dell'applicazione NestJS. L'avvio avviene tramite la funzione asincrona `bootstrap()`, che crea un'istanza dell'applicazione a partire da `AppModule`:

```
const app = await NestFactory.create(AppModule);
await app.listen(process.env.PORT ?? 3000);
```

Il server HTTP viene avviato sulla porta specificata tramite variabile d'ambiente `PORT`; in assenza di configurazione esplicita viene utilizzata la porta 3000 come valore di default.

Il file non contiene logica di business né accesso al database. La sua responsabilità è esclusivamente quella di inizializzare il contesto applicativo e rendere disponibili le API REST definite nei moduli importati.

L'uso di variabili d'ambiente consente di separare configurazione e codice, facilitando l'esecuzione del backend in ambienti differenti senza modifiche alla base sorgente.

AppModule

`AppModule` rappresenta il modulo radice dell'applicazione. Attraverso il decorator `@Module`, definisce la composizione dei moduli che costituiscono il backend:

- `PrismaModule`, responsabile dell'accesso al database PostgreSQL;
- `AuthModule`, dedicato all'autenticazione e alla gestione dei token JWT;
- `NetworkRequestsModule`, che implementa la gestione delle richieste di rete e delle operazioni GDPR.

Il modulo principale svolge una funzione di collegamento tra le dipendenze: non implementa logica applicativa, ma collega i diversi moduli funzionali in un'unica struttura coerente.

La suddivisione in moduli distinti consente di isolare le responsabilità, ridurre l'accoppiamento tra componenti e mantenere il backend organizzato per ambiti funzionali ben definiti.

4.2.3 Modello dati e persistenza

La persistenza dei dati è gestita tramite Prisma ORM con database PostgreSQL come sistema relazionale sottostante. Lo schema del database è definito nel file `schema.prisma`, che rappresenta la fonte di verità della struttura dati.

Schema Prisma

Lo schema definisce due modelli principali: `User` e `NetworkRequest`, oltre a un enum `Role` per la gestione dei ruoli utente.

Modello User Il modello `User` rappresenta l'utente autenticato del sistema ed è definito con:

- identificativo primario UUID (`@default(uuid())`);
- email univoca (`@unique`);
- password hashata;
- ruolo (`USER` o `ADMIN`);
- timestamp di creazione.

È definita una relazione uno-a-molti con `NetworkRequest`, che consente a ogni utente di essere associato a più richieste di rete.

L'utilizzo di UUID come chiave primaria evita identificativi sequenziali e riduce il rischio di enumerazione degli utenti.

Modello NetworkRequest Il modello `NetworkRequest` rappresenta una connessione aggregata ricevuta dal client Android. Ogni record è associato a un utente tramite la foreign key `userId`.

I campi principali del modello `NetworkRequest` sono descritti nella Tabella 4.4.

Tabella 4.4: Campi principali del modello NetworkRequest nel database PostgreSQL

Campo	**Tipo**	**Descrizione**
id	UUID	Identificativo univoco del record generato automaticamente.
userId	UUID	Chiave esterna che associa il record all'utente autenticato.
protocol	String	Protocollo di trasporto utilizzato (TCP, UDP, HTTP, HTTPS, ecc.).
domain	String (opzionale)	Dominio di destinazione associato alla connessione di rete.
dstPort	Integer (opzionale)	Porta di destinazione utilizzata dalla connessione.
bytesTx	BigInt	Numero totale di byte trasmessi durante la connessione.
bytesRx	BigInt	Numero totale di byte ricevuti durante la connessione.
startTs	BigInt	Timestamp di inizio della connessione (millisecondi).
endTs	BigInt	Timestamp di fine della connessione (millisecondi).
durationMs	BigInt	Durata complessiva della connessione in millisecondi.
latitude	Float (opzionale)	Coordinata geografica approssimata dell'utente.
longitude	Float (opzionale)	Coordinata geografica approssimata dell'utente.
createdAt	DateTime	Timestamp di creazione del record lato server.

I campi relativi a byte e timestamp sono definiti come `BigInt`. Questa scelta consente di evitare overflow nel caso di volumi di traffico elevati o timestamp rappresentati in millisecondi.

La relazione tra `User` e `NetworkRequest` è esplicitata tramite il campo:

```
User @relation(fields: [userId], references: [id])
```

In questo modo ogni richiesta è sempre associata a un utente valido a livello di vincolo referenziale.

PrismaService e gestione della connessione

L'accesso al database è centralizzato nella classe `PrismaService`, che estende `PrismaClient` ed è registrata come provider NestJS tramite `PrismaModule`.

Il servizio implementa l'interfaccia `OnModuleInit` e stabilisce esplicitamente la connessione al database all'avvio dell'applicazione tramite il metodo `$connect()`.

È stato introdotto un meccanismo di retry con numero limitato di tentativi e intervallo temporale fisso. Questa scelta consente al backend di attendere la disponibilità del database in scenari containerizzati, evitando errori immediati di inizializzazione.

Centralizzare Prisma in un unico service:

- evita la creazione di istanze multiple del client;
- consente l'iniezione tramite Dependency Injection;
- mantiene separata la logica di dominio dall'accesso diretto al database.

4.2.4 Autenticazione e gestione JWT

L'autenticazione degli utenti è implementata nel modulo `AuthModule`, che integra Prisma per l'accesso al database e `JwtModule` per la generazione e validazione dei token.

Il modulo registra:

- `AuthController`, responsabile degli endpoint HTTP;
- `AuthService`, che contiene la logica di autenticazione;
- `JwtStrategy`, per la validazione dei token;

Il `JwtModule` è configurato con secret letto da variabile d'ambiente (`JWT_SECRET`) e con scadenza del token impostata a 1 giorno. Il secret non è hardcoded nel codice sorgente.

AuthController

Il controller espone due endpoint:

- `POST /auth/login`
- `POST /auth/register`

Il controller non accede direttamente al database e non contiene logica di sicurezza. Riceve i parametri dal body della richiesta e delega completamente l'elaborazione a `AuthService`.

AuthService

`AuthService` implementa la logica di autenticazione.

Nel flusso di login:

1. recupera l'utente tramite `PrismaService`;
2. verifica la password tramite `bcrypt.compare`;
3. genera un payload contenente `sub` (userId) e `role`;
4. firma il JWT tramite `JwtService`.

Le password non vengono mai memorizzate in chiaro, ma sono salvate come hash bcrypt.

Nel flusso di registrazione vengono applicati controlli espliciti su:

- presenza dei campi obbligatori;
- validità del formato email;
- lunghezza minima della password;
- unicità dell'email nel database.

Gli errori sono gestiti tramite eccezioni HTTP specifiche (`BadRequestException`, `ConflictException`), garantendo risposte coerenti con lo standard REST.

JwtStrategy

La validazione del token è centralizzata nella classe `JwtStrategy`, che estende `PassportStrategy`.

Il token viene estratto dall'header `Authorization` nel formato:

```
Authorization: Bearer <token>
```

Dopo la verifica della firma e della scadenza, il metodo `validate()` restituisce un oggetto contenente:

- `userId`, derivato dal campo `sub`;
- `role`, per eventuali controlli di autorizzazione.

Tali informazioni vengono rese disponibili in `req.user` per gli endpoint protetti.

JwtAuthGuard

La protezione delle rotte avviene tramite `JwtAuthGuard`, che estende `AuthGuard('jwt')`.

Il guard intercetta le richieste HTTP e:

- blocca richieste prive di token;
- blocca token invalidi o scaduti;
- consente l'accesso solo a utenti autenticati.

In questo modo il backend rimane stateless: l'identità dell'utente non è memorizzata lato server, ma viene ricostruita a ogni richiesta tramite il token firmato.

4.2.5 Gestione delle richieste di rete

La gestione delle richieste di rete è implementata nel `NetworkRequestsModule`, che collega `NetworkRequestsController`, `NetworkRequestsService` e `PrismaModule`.

Il modulo espone le API REST utilizzate dall'applicazione Android per l'invio batch dei dati raccolti e per l'esercizio dei diritti GDPR.

NetworkRequestsController

Il controller definisce i seguenti endpoint:

Tabella 4.5: Endpoint REST esposti dal backend

Metodo	Endpoint	Auth	Ruolo	Descrizione
POST	/network-requests/batch	JWT	USER	Upload batch delle connessioni raccolte.
GET	/network-requests/me/data	JWT	USER	Recupero dei dati associati all'utente autenticato.
GET	/network-requests/me/data/export	JWT	USER	Esportazione dei dati in formato CSV.
DELETE	/network-requests/me/data	JWT	USER	Cancellazione permanente dei dati dell'utente.
GET	/network-requests/admin/export	JWT	ADMIN	Esportazione globale dei dati (accesso amministrativo).

Tutti gli endpoint sono protetti da `JwtAuthGuard`. L'endpoint amministrativo è ulteriormente protetto da `RolesGuard` e dal decorator `@Roles('ADMIN')`.

Il controller non accede direttamente al database e non contiene logica di dominio. L'identificativo dell'utente viene sempre estratto da `req.user.userId`, popolato dalla `JwtStrategy`, e non viene mai accettato dal body della richiesta.

Upload batch

L'endpoint `POST /network-requests/batch` riceve un oggetto `BatchNetworkRequestDto` contenente una lista di richieste di rete.

Il metodo `createBatch()` nel service utilizza:

```
prisma.networkRequest.createMany(...)
```

L'inserimento multiplo in un'unica query riduce l'overhead rispetto a inserimenti singoli e migliora le prestazioni in presenza di batch numerosi.

L'associazione tra richiesta e utente avviene esclusivamente lato backend tramite il parametro `userId` estratto dal JWT. Eventuali identificativi presenti nel payload del client non vengono considerati.

I campi numerici definiti come `BigInt` nello schema Prisma vengono convertiti esplicitamente tramite `BigInt(...)` prima dell'inserimento, garantendo coerenza con il modello dati.

Recupero dati e serializzazione

L'endpoint `GET /network-requests/me/data` restituisce le richieste associate all'utente autenticato, ordinate per timestamp decrescente.

Poiché i campi `BigInt` non sono serializzabili direttamente in JSON, la conversione in stringa avviene esclusivamente in fase di output. Il database mantiene i valori numerici nel formato originale.

Esportazione CSV

L'endpoint `GET /network-requests/me/data/export` consente il download dei dati in formato CSV.

Il service:

- recupera i record dell'utente;
- costruisce dinamicamente l'intestazione a partire dalle chiavi dell'oggetto;
- genera le righe CSV con escaping dei valori;
- restituisce una stringa formattata.

Il controller imposta gli header:

```
Content-Type: text/csv
Content-Disposition: attachment
```

Questa funzionalità implementa il diritto alla portabilità dei dati.

Cancellazione dei dati

L'endpoint `DELETE /network-requests/me/data` utilizza:

```
prisma.networkRequest.deleteMany(...)
```

La cancellazione è permanente e limitata ai record associati all'utente autenticato.

Endpoint amministrativo

L'endpoint `GET /network-requests/admin/export` consente l'esportazione globale di tutte le richieste di rete.

L'accesso è consentito esclusivamente a utenti con ruolo `ADMIN`. Il controllo di autorizzazione è delegato a `RolesGuard`, che verifica il campo `role` presente nel payload del JWT.

Il service implementa `exportAllCsv()`, che recupera tutti i record e applica la stessa logica di serializzazione utilizzata per l'export utente.

4.2.6 Controllo degli accessi e ruoli (RBAC)

Oltre all'autenticazione tramite JWT, il backend implementa un meccanismo di autorizzazione basato su ruoli (Role-Based Access Control, RBAC).

I ruoli disponibili sono definiti nello schema Prisma tramite l'enum `Role`:

- `USER`
- `ADMIN`

Il ruolo dell'utente viene incluso nel payload del JWT al momento del login e reso disponibile in `req.user.role` dalla `JwtStrategy`.

RolesDecorator

Il file `roles.decorator.ts` definisce il decorator:

```
@Roles('ADMIN')
```

Il decorator utilizza `SetMetadata` per associare al metodo del controller un metadata identificato dalla chiave `ROLES_KEY`. Non contiene logica di autorizzazione, ma si limita a dichiarare quali ruoli sono richiesti per accedere a un determinato endpoint.

RolesGuard

La verifica effettiva dei permessi è implementata nella classe `RolesGuard`, che implementa l'interfaccia `CanActivate`.

Il guard utilizza `Reflector` per leggere i metadata associati al metodo o alla classe tramite:

`this.reflector.getAllAndOverride(...)`

Se non sono definiti ruoli richiesti, l'accesso viene consentito. In caso contrario, il guard confronta il ruolo presente in `req.user.role` con l'elenco dei ruoli richiesti.

L'accesso è autorizzato solo se il ruolo dell'utente è incluso tra quelli specificati nel decorator.

Applicazione agli endpoint

L'endpoint amministrativo:

```
GET /network-requests/admin/export
```

è protetto tramite:

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles('ADMIN')
```

In questo modo:

- `JwtAuthGuard` verifica l'autenticazione;
- `RolesGuard` verifica l'autorizzazione;

separando chiaramente il controllo dell'identità dal controllo dei permessi.

Questo approccio evita di inserire controlli condizionali nei service e mantiene la logica di autorizzazione isolata a livello di infrastruttura HTTP.

4.2.7 Input e gestione degli errori

La gestione degli input in ingresso è realizzata attraverso una combinazione di controlli espliciti nel livello di servizio e definizione strutturale dei DTO.

Nel modulo di autenticazione, la validazione avviene in modo esplicito all'interno di `AuthService`. Durante la registrazione vengono verificati:

- presenza dei campi obbligatori;
- correttezza del formato email tramite espressione regolare;
- lunghezza minima della password;
- unicità dell'email nel database.

In caso di errore vengono sollevate eccezioni HTTP specifiche (`BadRequestException`, `ConflictException`, `UnauthorizedException`), garantendo risposte coerenti con lo standard REST.

Per l'endpoint di upload batch, la struttura del payload è definita tramite i DTO `BatchNetworkRequestDto` e `CreateNetworkRequestDto`. Tali classi specificano vincoli di

tipo sui campi (stringhe, interi, numeri opzionali), separando il modello esterno dell'API dal modello interno di persistenza.

Questa organizzazione consente di mantenere isolata la validazione dei dati rispetto alla logica di dominio e di ridurre il rischio di input malformati.

4.2.8 Deployment del sistema

Per consentire la raccolta dei dati in un contesto reale, il backend del sistema è stato deployato su un server del laboratorio e reso accessibile tramite un dominio pubblico. Questa configurazione ha permesso all'applicazione Android di comunicare direttamente con l'infrastruttura remota durante la fase sperimentale, senza utilizzare ambienti locali o emulatori.

Il backend è stato containerizzato tramite Docker, in modo da garantire un ambiente di esecuzione riproducibile e indipendente dal sistema operativo del server. L'immagine del servizio è costruita a partire da una base `Node 20` e include tutte le dipendenze necessarie per l'esecuzione dell'applicazione NestJS. Durante la fase di build vengono installati i pacchetti del progetto, generato il client Prisma e compilata l'applicazione.

All'avvio del container vengono inoltre eseguite automaticamente le migrazioni del database tramite il comando `prisma migrate deploy`, seguite dall'esecuzione dello script di inizializzazione dei dati. Questa scelta consente di garantire la coerenza dello schema del database anche in caso di deploy su nuovi ambienti, evitando interventi manuali durante l'avvio del sistema.

L'infrastruttura è composta da due servizi principali gestiti tramite Docker: il backend applicativo e il database. Il database utilizza PostgreSQL con estensione PostGIS, scelta che permette di supportare eventuali operazioni di analisi spaziale sui dati raccolti. I dati vengono salvati su un volume persistente, in modo da mantenere lo stato del database anche in caso di riavvio o ricreazione dei container.

Il deploy sul server del laboratorio è stato gestito tramite Portainer, utilizzato per eseguire lo stack Docker e monitorare lo stato dei servizi. Il backend è esposto pubblicamente tramite protocollo HTTPS su un dominio del laboratorio, configurato come endpoint per la comunicazione con l'applicazione mobile durante la fase sperimentale.

Dal punto di vista architetturale, il flusso di comunicazione durante l'esecuzione del sistema può essere sintetizzato in questo modo: l'applicazione mobile invia i dati raccolti al backend tramite connessione HTTPS; il backend NestJS riceve le richieste, gestisce l'autenticazione e la validazione dei dati e infine persiste le informazioni nel database PostgreSQL tramite l'ORM Prisma.

Questa configurazione ha permesso di disporre di un'infrastruttura stabile e facilmente replicabile, adatta alla fase di sperimentazione con dispositivi reali e alla raccolta dei dati analizzati nel resto di questo capitolo.

Capitolo 5

Validazione

5.1 Metodologia della sperimentazione

La validazione del sistema è stata condotta attraverso una sperimentazione su dispositivi reali, finalizzata a verificare il corretto funzionamento dell'infrastruttura sviluppata e a raccogliere un dataset di traffico di rete.

L'applicazione Android è stata distribuita sotto forma di file APK a un gruppo di tester composto complessivamente da 12 utenti. I partecipanti hanno installato l'applicazione sui propri dispositivi personali e l'hanno utilizzata durante le normali attività quotidiane, consentendo al sistema di registrare il traffico di rete generato dalle applicazioni presenti sul dispositivo. Il numero di connessioni osservate per ciascun dispositivo varia significativamente, riflettendo differenze nell'intensità di utilizzo degli smartphone e nella durata delle sessioni di monitoraggio attivate dai tester.

La raccolta dei dati è stata effettuata nell'arco di circa una settimana. Durante questo periodo gli utenti potevano avviare e arrestare manualmente il servizio di monitoraggio tramite l'interfaccia dell'applicazione.

È importante sottolineare che la raccolta del traffico non è stata necessariamente continua per tutta la durata dell'esperimento, poiché l'attivazione del servizio di monitoraggio dipendeva dall'azione diretta degli utenti. Nonostante ciò, il volume complessivo dei dati raccolti risulta significativo e sufficiente per effettuare un'analisi delle principali caratteristiche del traffico di rete osservato.

5.2 Descrizione del dataset raccolto

La fase di sperimentazione ha prodotto un dataset composto complessivamente da 215 324 richieste di rete.

La quasi totalità dei record include anche informazioni di geolocalizzazione: su 215 324 connessioni registrate, 215 319 presentano coordinate valide. Questo livello di completezza,

superiore al 99.99% del dataset, ha reso possibile effettuare un'analisi spaziale del traffico osservato. I pochi record privi di coordinate geografiche sono stati esclusi dalle analisi che richiedono informazioni di localizzazione.

Il dataset rappresenta quindi una raccolta significativa di traffico di rete generato da dispositivi mobili in condizioni di utilizzo reale. I dati ottenuti consentono di analizzare diversi aspetti del comportamento del traffico mobile, tra cui la distribuzione dei protocolli utilizzati, le porte maggiormente impiegate, la durata delle connessioni e l'andamento temporale delle richieste durante la giornata.

Nella sezione seguente vengono analizzate alcune delle principali caratteristiche del traffico osservato, attraverso una serie di visualizzazioni che permettono di evidenziare i pattern più rilevanti presenti nel dataset raccolto.

5.3 Analisi del traffico osservato

5.3.1 Distribuzione dei protocolli

Per analizzare le caratteristiche del traffico di rete osservato durante la fase sperimentale, è stata innanzitutto considerata la distribuzione delle connessioni in base al protocollo utilizzato.

La Figura 5.1 mostra il numero di connessioni registrate per ciascun protocollo identificato nel dataset raccolto. Come prevedibile nel contesto del traffico Internet moderno, la maggior parte delle connessioni risulta associata a protocolli sicuri, in particolare HTTPS e TLS. Questo comportamento riflette la crescente diffusione della cifratura nelle comunicazioni tra applicazioni e servizi web.

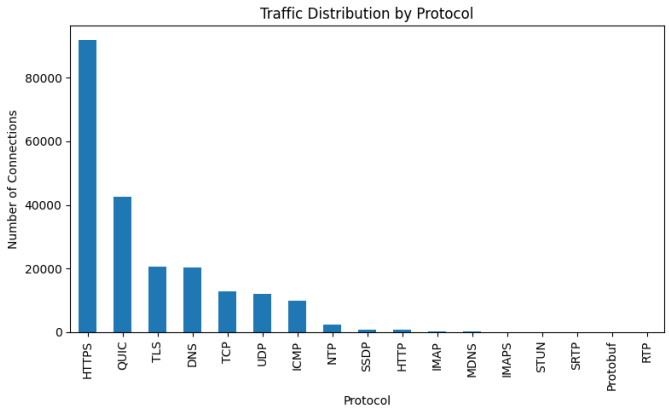


Figura 5.1: Distribuzione delle connessioni in base al protocollo di rete utilizzato.

Nel dataset è inoltre possibile osservare la presenza del protocollo QUIC (Quick UDP Internet Connections), un protocollo di trasporto sviluppato originariamente da Google e utilizzato come base per HTTP/3. QUIC opera sopra UDP e integra meccanismi di

sicurezza e controllo della congestione, consentendo di ridurre la latenza e migliorare le prestazioni delle connessioni web.

Tra le informazioni raccolte compaiono sia protocolli applicativi (come HTTPS, DNS e QUIC) sia protocolli di trasporto come TCP e UDP. Questo è dovuto al fatto che il sistema di monitoraggio osserva le connessioni a diversi livelli dello stack di rete: in alcuni casi è possibile identificare direttamente il protocollo applicativo utilizzato, mentre in altri casi viene registrato soltanto il protocollo di trasporto sottostante.

Nel complesso, la distribuzione osservata mostra una forte predominanza di comunicazioni cifrate e di richieste di breve durata verso servizi remoti, caratteristiche coerenti con il tipo di traffico osservato nel dataset raccolto durante la sperimentazione.

5.3.2 Distribuzione delle porte

Un ulteriore aspetto analizzato riguarda la distribuzione delle connessioni in base alla porta di destinazione utilizzata. L'analisi delle porte consente di identificare i servizi di rete più frequentemente contattati dalle applicazioni presenti sui dispositivi.

La Figura 5.2 mostra le porte di destinazione più frequenti osservate nel dataset. Come previsto, la porta 443 risulta nettamente predominante rispetto alle altre. Questa porta è utilizzata dal protocollo HTTPS ed è oggi lo standard per la maggior parte delle comunicazioni web sicure tra client e server.

Tra le altre porte più presenti nel dataset si osserva la porta 53, associata al protocollo DNS, utilizzato per la risoluzione dei nomi di dominio. Sono inoltre presenti porte come 5222 e 5228, comunemente utilizzate da servizi di messaggistica e notifiche push, ad esempio nell'infrastruttura di Google Play Services.

La distribuzione osservata conferma quindi come gran parte del traffico generato dai dispositivi mobili sia legato all'accesso a servizi web sicuri e alla comunicazione con infrastrutture cloud utilizzate dalle applicazioni installate sul dispositivo.

Nel grafico è inoltre presente la porta 0. In questo contesto il valore 0 non rappresenta una porta di destinazione reale, ma indica connessioni per le quali l'informazione relativa alla porta non è stata identificata dal sistema di monitoraggio oppure non era disponibile nei metadati della connessione osservata.

5.3.3 Distribuzione della durata delle connessioni

Un ulteriore elemento analizzato riguarda la durata delle connessioni di rete registrate durante la fase di sperimentazione. La durata rappresenta il tempo intercorso tra l'apertura e la chiusura di una connessione osservata dal sistema.

La Figura 5.3 mostra la distribuzione delle durate delle connessioni presenti nel dataset. Come si può osservare dal grafico, la maggior parte delle connessioni ha una durata relativamente breve. Questo comportamento è tipico del traffico generato dalle applicazioni

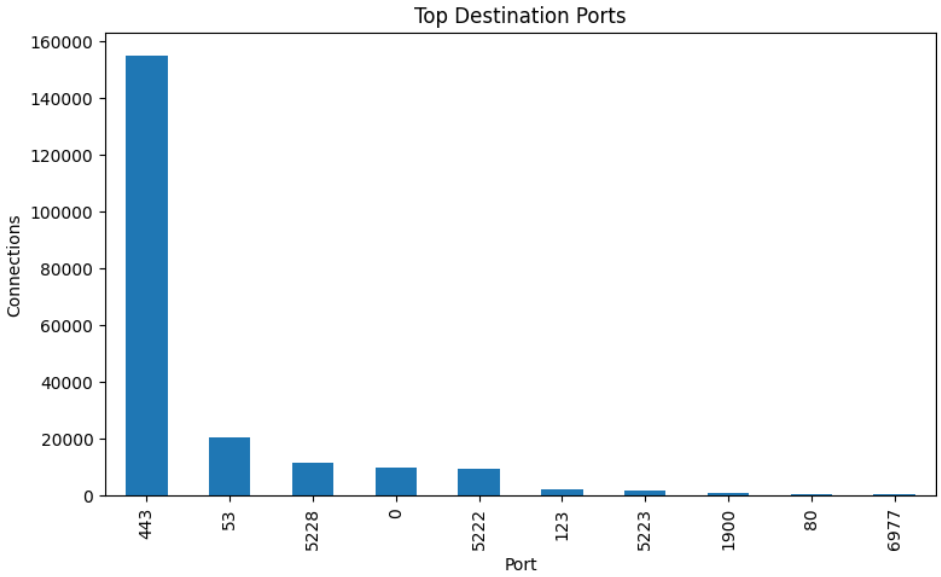


Figura 5.2: Distribuzione delle principali porte di destinazione.

mobili, che spesso stabiliscono connessioni temporanee per effettuare operazioni specifiche come richieste HTTP, sincronizzazioni di dati o aggiornamenti di stato.

Nel grafico è inoltre possibile osservare la presenza di un numero minore di connessioni con durata più elevata. Queste possono essere associate a sessioni più persistenti, ad esempio connessioni mantenute attive per servizi di messaggistica, streaming o comunicazioni in tempo reale.

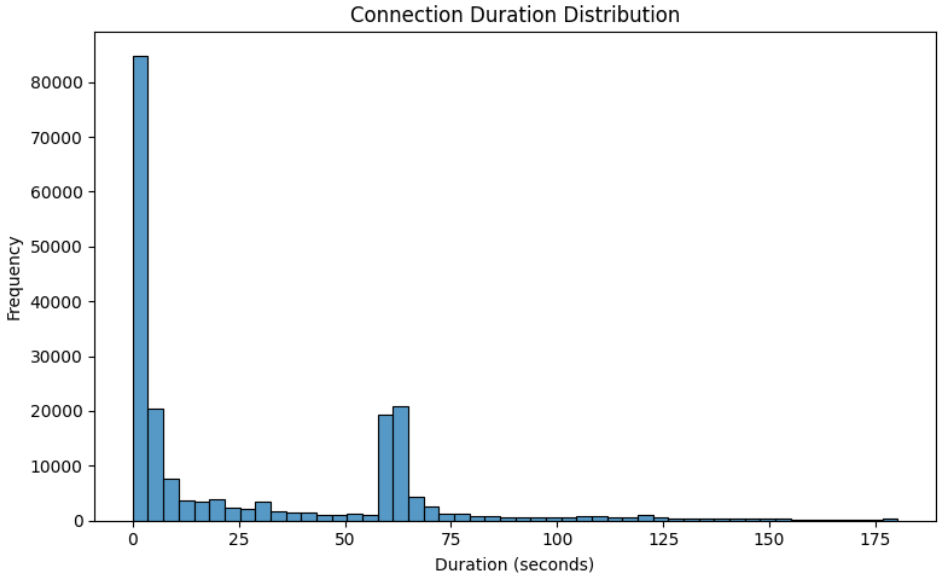


Figura 5.3: Distribuzione della durata delle connessioni.

Nel complesso, la distribuzione osservata è coerente con il comportamento tipico del traffico mobile, caratterizzato da un elevato numero di connessioni brevi e da un numero

più limitato di connessioni di durata maggiore.

5.3.4 Distribuzione temporale del traffico

Per comprendere come il traffico di rete si distribuisca nel corso della giornata, è stata analizzata la frequenza delle connessioni osservate per ciascuna ora del giorno.

La Figura 5.4 mostra il numero di connessioni registrate nel dataset suddivise per fascia oraria. Dal grafico emerge come il traffico non sia distribuito uniformemente durante la giornata. In generale si osserva una minore attività nelle ore notturne, mentre il numero di connessioni tende ad aumentare progressivamente nelle ore diurne e nel tardo pomeriggio.

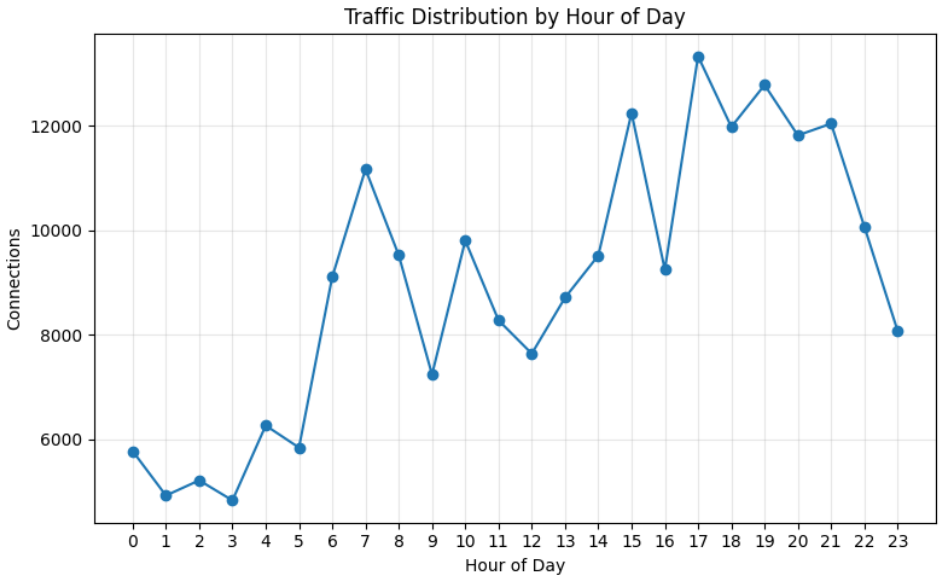


Figura 5.4: Distribuzione del numero di connessioni osservate nel dataset in base all'ora del giorno.

Questo andamento rispecchia il comportamento tipico nell'utilizzo degli smartphone, che vengono impiegati più frequentemente durante le attività quotidiane e nelle ore serali. La distribuzione temporale osservata è quindi coerente con il contesto di utilizzo reale dei dispositivi coinvolti nella sperimentazione.

L'analisi della dimensione temporale rappresenta inoltre un primo passo per comprendere come il traffico mobile possa variare nel tempo e costituisce la base per le successive analisi che integrano anche la dimensione geografica dei dati raccolti.

5.4 Analisi geografica del traffico

5.4.1 Distribuzione spaziale delle connessioni

Oltre alla dimensione temporale, il dataset raccolto consente di analizzare anche la distribuzione geografica delle connessioni di rete registrate durante la sperimentazione. Grazie alla presenza delle coordinate geografiche associate ai record, è possibile osservare come il traffico si distribuisca nello spazio in relazione ai luoghi in cui i dispositivi sono stati utilizzati.

La Figura 5.5 mostra una visualizzazione complessiva della distribuzione geografica delle connessioni registrate. Dal grafico emerge come il traffico non sia uniformemente distribuito sul territorio, ma presenti alcune aree di maggiore concentrazione. In particolare si osservano cluster di traffico nelle principali zone in cui i dispositivi sono stati utilizzati durante la fase sperimentale.

Per analizzare più nel dettaglio la distribuzione del traffico in ambiente urbano, la Figura 5.6 mostra uno zoom dell'area di Bologna. In questo caso è possibile osservare una maggiore concentrazione di connessioni nella zona urbana principale e in alcune aree circostanti. Tra queste si distingue anche la zona dell'aeroporto, dove è stata registrata una presenza significativa di richieste di rete.

Questa distribuzione riflette come il traffico di rete generato dai dispositivi mobili sia strettamente legato ai luoghi frequentati dall'utente durante la giornata. L'analisi geografica dei dati permette quindi di osservare come l'utilizzo delle applicazioni e dei servizi Internet sia influenzato dal contesto spaziale in cui si trova il dispositivo.

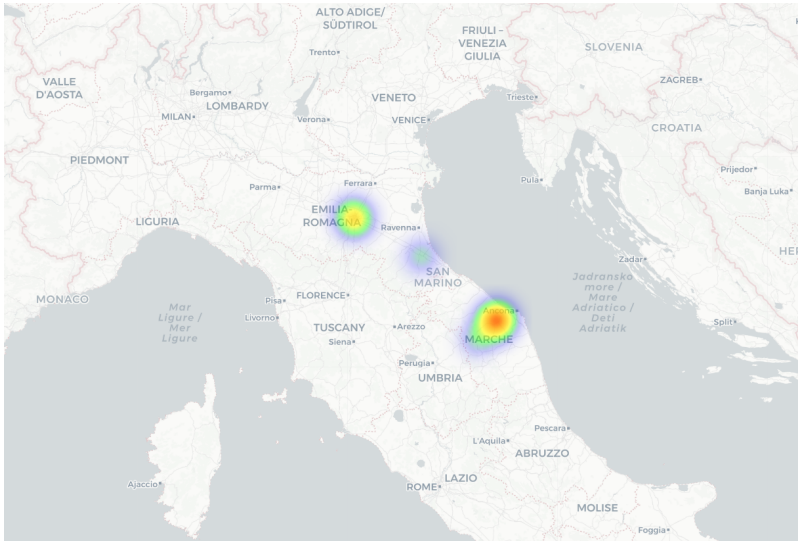


Figura 5.5: Distribuzione geografica complessiva delle connessioni registrate.

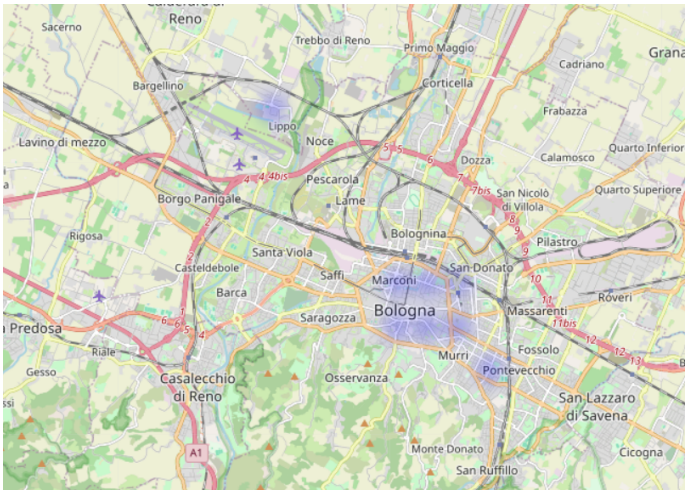


Figura 5.6: Zoom della distribuzione delle connessioni nell'area urbana di Bologna.

È possibile osservare una maggiore concentrazione di traffico nelle principali zone urbane e nelle aree limitrofe.

5.4.2 Analisi spazio-temporale del traffico per utente

Per comprendere meglio il comportamento del traffico mobile, è utile analizzare con maggiore dettaglio il traffico generato da un singolo utente, considerando simultaneamente la dimensione spaziale e quella temporale.

A questo scopo è stato selezionato uno degli utenti con il maggior numero di connessioni registrate nel dataset. Le coordinate geografiche associate alle connessioni sono state raggruppate tramite un algoritmo di clustering spaziale (DBSCAN), in modo da identificare automaticamente alcune aree geografiche principali in cui il dispositivo è stato utilizzato durante il periodo di sperimentazione.

La Figura 5.7 mostra la distribuzione delle connessioni registrate per i diversi cluster geografici individuati per l'utente selezionato.

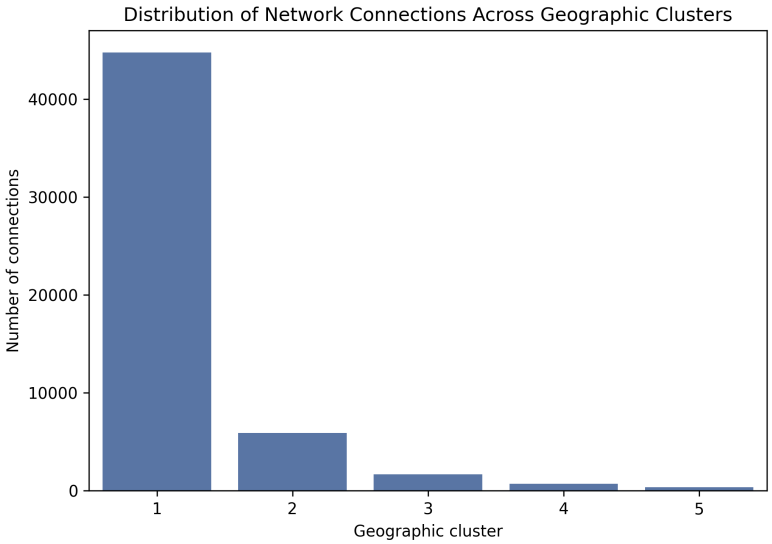


Figura 5.7: Distribuzione delle connessioni per cluster geografico per un utente del dataset.

Dal grafico si può notare come la maggior parte delle connessioni sia concentrata in uno o due cluster principali, mentre gli altri cluster presentano un numero significativamente inferiore di richieste. Questo comportamento suggerisce che gran parte del traffico venga generato in un numero limitato di luoghi abitualmente frequentati dall'utente.

Per analizzare come il traffico vari nel corso della giornata nelle diverse aree geografiche individuate, è stata considerata la distribuzione delle connessioni per ora del giorno all'interno di ciascun cluster.

La Figura 5.8 mostra una heatmap che visualizza la distribuzione temporale delle connessioni nei diversi cluster geografici.

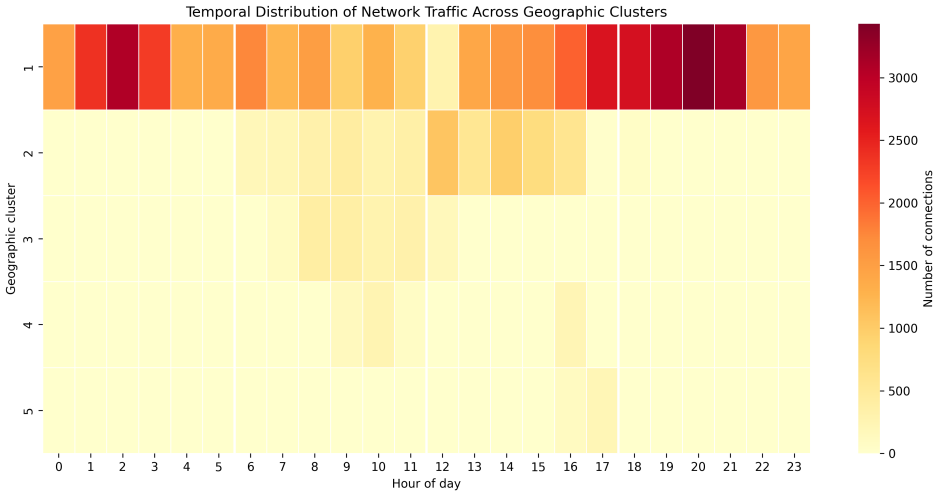


Figura 5.8: Distribuzione temporale delle connessioni per cluster geografico nel corso della giornata.

La heatmap evidenzia come l'intensità del traffico possa variare non solo nel corso della giornata, ma anche in relazione al luogo in cui si trova il dispositivo. Alcuni cluster presentano infatti una maggiore concentrazione di traffico in specifiche fasce orarie, mentre altri mostrano un'attività più distribuita nel tempo.

Questa analisi evidenzia come il traffico di rete generato dai dispositivi mobili sia influenzato contemporaneamente da fattori spaziali e temporali, riflettendo le abitudini di utilizzo dell'utente e i contesti in cui il dispositivo viene impiegato.

5.5 Discussione dei risultati

L'analisi del dataset raccolto durante la fase di sperimentazione consente di osservare alcune caratteristiche tipiche del traffico di rete generato da dispositivi mobili in condizioni di utilizzo reale.

In primo luogo, la distribuzione dei protocolli e delle porte evidenzia una forte predominanza di comunicazioni cifrate tramite HTTPS, confermando la diffusione ormai consolidata di protocolli sicuri nelle applicazioni moderne. La presenza di protocolli come QUIC e DNS riflette inoltre il comportamento tipico delle applicazioni mobili e dei servizi web contemporanei.

L'analisi della durata delle connessioni mostra come la maggior parte delle richieste sia caratterizzata da connessioni brevi, spesso legate a operazioni come interrogazioni di servizi remoti, sincronizzazioni o aggiornamenti di stato. Questo comportamento è coerente con il modello di comunicazione utilizzato da molte applicazioni mobili.

Dal punto di vista temporale, il traffico osservato presenta variazioni nel corso della giornata che riflettono i normali pattern di utilizzo degli smartphone, con una maggiore attività nelle ore diurne e serali e una minore attività durante la notte.

Infine, l'analisi geografica e l'analisi spazio-temporale del traffico evidenziano come le connessioni registrate siano strettamente legate sia ai luoghi frequentati dagli utenti sia ai momenti della giornata in cui i dispositivi vengono utilizzati. Considerando insieme la dimensione spaziale e quella temporale, emerge come il traffico mobile sia fortemente influenzato dalle abitudini di utilizzo degli utenti e dai diversi contesti in cui i dispositivi vengono utilizzati.

È opportuno sottolineare che il dataset raccolto, pur risultando significativo, deriva da un numero limitato di utenti e da un periodo di osservazione relativamente breve. Studi futuri potrebbero estendere la sperimentazione a un numero maggiore di dispositivi e a intervalli temporali più lunghi, al fine di ottenere un quadro ancora più rappresentativo del traffico mobile.

Capitolo 6

Conclusioni

6.1 Sintesi del lavoro svolto

In questa tesi è stato progettato e sviluppato un sistema per la raccolta e l'analisi di metadati relativi al traffico di rete generato da dispositivi mobili Android. L'obiettivo principale del lavoro è stato quello di realizzare un'infrastruttura in grado di osservare e registrare alcune caratteristiche delle connessioni di rete, associandole a informazioni temporali e, previo consenso dell'utente, anche alla posizione geografica del dispositivo.

Il sistema sviluppato è composto da due componenti principali: un'applicazione Android e un server backend. L'applicazione mobile utilizza un servizio VPN locale per intercettare le connessioni di rete generate dal dispositivo e raccoglierne alcuni metadati, tra cui protocollo utilizzato, porta di destinazione, durata della connessione e timestamp. Queste informazioni vengono inizialmente salvate localmente e successivamente sincronizzate con il backend.

Il backend, sviluppato utilizzando il framework NestJS, è responsabile della gestione delle richieste provenienti dai dispositivi e della memorizzazione dei dati in un database PostgreSQL. L'infrastruttura è stata containerizzata tramite Docker e deployata su un server del laboratorio, rendendo il sistema accessibile tramite un endpoint pubblico.

La fase di validazione è stata condotta attraverso una sperimentazione con dispositivi reali, durante la quale è stato raccolto un dataset composto da oltre 215 mila connessioni di rete. L'analisi dei dati ha permesso di osservare diverse caratteristiche del traffico mobile, tra cui la distribuzione dei protocolli utilizzati, le porte maggiormente impiegate, la durata delle connessioni, l'andamento temporale delle richieste nel corso della giornata e alcune differenze nei pattern di traffico osservati tra diversi utenti.

Grazie alla presenza delle coordinate geografiche associate ai record, è stato inoltre possibile analizzare la distribuzione spaziale delle connessioni e individuare alcune aree di maggiore concentrazione del traffico. Questo ha permesso di osservare come il traffico di rete mobile possa essere interpretato come un fenomeno spazio-temporale, influenzato sia

dalla posizione del dispositivo sia dal momento della giornata in cui avviene l'interazione con le applicazioni.

6.2 Contributo rispetto allo stato dell'arte

Uno degli aspetti principali che caratterizzano il lavoro presentato in questa tesi riguarda l'approccio utilizzato per la raccolta dei dati. Molti degli studi presenti in letteratura analizzano il traffico di rete mobile a partire da dataset forniti da operatori di rete o da provider di servizi. In questi casi l'analisi viene spesso effettuata su dati aggregati e non sempre consente di osservare direttamente il comportamento di un singolo dispositivo.

Il sistema sviluppato in questa tesi adotta invece un approccio basato sul dispositivo, permettendo di raccogliere i metadati delle connessioni direttamente dallo smartphone. Questo consente di associare alle richieste di rete non solo informazioni tecniche relative al traffico, ma anche dati temporali e geografici che riflettono il contesto di utilizzo del dispositivo.

Un ulteriore contributo del lavoro è rappresentato dalla realizzazione di un'infrastruttura completa per la raccolta e la gestione dei dati. Il sistema integra infatti un'applicazione mobile, un backend server e un database progettato per supportare anche informazioni geospaziali tramite l'estensione PostGIS. Questa architettura consente di costruire un dataset strutturato potenzialmente utile per future analisi sul comportamento del traffico mobile.

L'integrazione tra la dimensione temporale e quella geografica permette inoltre di osservare come il traffico di rete si distribuisca nello spazio e nel tempo. Ciò risulta in linea con le considerazioni presenti nello stato dell'arte riguardo alla relazione tra mobilità degli utenti e utilizzo dei servizi Internet.

6.3 Sviluppi futuri

Il sistema sviluppato in questa tesi rappresenta una base su cui è possibile costruire ulteriori funzionalità e strumenti di analisi più avanzati.

Un primo possibile sviluppo riguarda la realizzazione di un'interfaccia di amministrazione per la gestione e l'esplorazione dei dati raccolti. Nel sistema attuale è previsto un ruolo amministratore utilizzato per accedere ai dati memorizzati nel backend, ma non è ancora stata sviluppata una dashboard dedicata. In futuro potrebbe essere implementata un'interfaccia web che permetta agli amministratori di visualizzare statistiche, interrogare il database e gestire i dati raccolti tramite operazioni di tipo CRUD.

Un secondo ambito di sviluppo riguarda l'utilizzo più esteso delle funzionalità geospaziali offerte da PostGIS. Il database utilizzato nel sistema supporta infatti operazioni di geoquery che potrebbero essere sfruttate per effettuare analisi più avanzate sulla distribu-

zione geografica del traffico, ad esempio per individuare cluster di attività o confrontare il comportamento del traffico in diverse aree urbane.

Infine, un ulteriore sviluppo potrebbe riguardare l'estensione della fase di raccolta dei dati coinvolgendo un numero maggiore di dispositivi e prolungando il periodo di sperimentazione. Un dataset più ampio permetterebbe di effettuare analisi più approfondite e di studiare in modo più dettagliato i pattern spazio-temporali del traffico mobile.

Nel complesso, il sistema sviluppato dimostra come sia possibile realizzare un'infrastruttura leggera e non invasiva per l'osservazione del traffico mobile direttamente dal dispositivo, aprendo la strada a studi futuri sul comportamento delle applicazioni e sulle dinamiche spazio-temporali dell'utilizzo dei servizi Internet su dispositivi mobili.

Bibliografia

[1] E. Floris, "PCAPdroid: Network traffic capture for Android," 2024. [Online]. Available: https://github.com/emanuele-f/PCAPdroid

[2] Yang, Jie and Qiao, Yuanyuan and Zhang, Xinyu and He, Haiyang and Liu, Fang and Cheng, Gang, "Characterizing User Behavior in Mobile Internet," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 1, pp. 95–106, 2015.

[3] Zhang, Yi and Yang, Lintao and Jiang, Hao and Yi, Shuwen and Hu, Zhiyi and Liu, Xing, "Mining Mobile Internet Lifestyles in Distinct Urban Areas: Tales of Two Cities," *IEEE Access*, vol. 6, pp. 36 208–36 217, 2018.

[4] Z. Jin, J. Qian, Z. Kong, and C. Pan, "A mobility aware network traffic prediction model based on dynamic graph attention spatio-temporal network," *Computer Networks*, vol. 235, p. 109981, 2023.

[5] V. J. Aski, R. S. Chavan, V. S. Dhaka, G. Rani, E. Zumpano, and E. Vocaturo, "Forecasting of mobile network traffic and spatio–temporal analysis using modLSTM," *Machine Learning*, vol. 113, no. 4, pp. 2277–2300, 2024.

[6] Lv, Qiujian and Qiao, Yuanyuan and Zhang, Yi and Abdesslem, Fehmi Ben and Lin, Wenhui and Yang, Jie, "Measuring Geospatial Properties: Relating Online Content Browsing Behaviors to Users' Points of Interest," *Wireless Personal Communications*, vol. 101, no. 3, pp. 1469–1498, 2018. [Online]. Available: https://doi.org/10.1007/s11277-018-5773-7

[7] Tang, Vicente and Painho, Marco, "Content-location relationships: a framework to explore correlations between space-based and place-based user-generated content," *International Journal of Geographical Information Science*, vol. 37, no. 8, pp. 1840–1871, 2023. [Online]. Available: https://doi.org/10.1080/13658816.2023.2213869

[8] I. Zyrianoff, A. Trotta, L. Sciullo, F. Montori, and M. Di Felice, "IoT Edge Caching: Taxonomy, use cases and perspectives," *IEEE Internet of Things Magazine*, vol. 5, no. 3, pp. 12–18, 2022.

[9] I. Zyrianoff, L. Gigli, F. Montori, L. Sciullo, C. Kamienski, and M. Di Felice, "CACHE-IT: A distributed architecture for proactive edge caching in heterogeneous IoT scenarios," *Ad Hoc Networks*, vol. 156, p. 103413, 2024.

[10] Android Developers, "VpnService," 2024. [Online]. Available: https://developer.android.com/develop/connectivity/vpn

[11] ——, "Android Security Overview," 2024. [Online]. Available: https://source.android.com/docs/security

[12] J. Postel, "Transmission Control Protocol," RFC 793, 1981. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc793

[13] Y. Shen, "go-tun2socks," 2024. [Online]. Available: https://github.com/eycorsican/go-tun2socks

[14] OpenVPN Inc., "OpenVPN," 2024. [Online]. Available: https://openvpn.net/

[15] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8446

[16] S. Shukla, O. Bhardwaj, A. A. Abouzeid, T. Salonidis, and T. He, "Proactive retention-aware caching with multi-path routing for wireless edge networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 6, pp. 1286–1299, 2018.

[17] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin, "A first look at traffic on smartphones," ser. IMC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 281–287.

[18] Le, Anh and Varmarken, Janus and Langhoff, Simon and Shuba, Anastasia and Gjoka, Minas and Markopoulou, Athina, "AntMonitor: A System for Monitoring from Mobile Devices," ser. C2B(1)D '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 15—-20. [Online]. Available: https://doi.org/10.1145/2787394.2787396

[19] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 439–454.

[20] M. Bokhorst, "NetGuard - no-root firewall for Android," https://netguard.me/, 2024.

[21] Taosoftware Co., Ltd., "tPacketCapture - Packet Capture for Android," https://www.taosoftware.co.jp/en/android/packetcapture/, 2024.

[22] "Regulation (EU) 2016/679 of the European Parliament and of the Council (General Data Protection Regulation)," Official Journal of the European Union, 2016. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2016/679/oj