ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica — Scienza e Ingegneria
Corso di Laurea triennale in Informatica

Progettazione e sviluppo di un protocollo di routing ibrido MANET-DTN a livello applicativo per reti off-grid su dispositivi mobili commerciali

Relatore:
Chiar.ma Prof.
ANGELO TROTTA

Presentata da:
GIUSEPPE BONDI

Marzo 2026
Anno Accademico 2024/2025

Abstract

Le reti di telecomunicazione basate su infrastrutture centralizzate presentano criticità intrinseche in scenari operativi *off-grid* o post-disastro. In tali contesti, le reti mobili ad-hoc (MANET) e le *Delay-Tolerant Networks* (DTN) offrono un paradigma di comunicazione decentralizzato essenziale. Tuttavia, l'implementazione pratica di protocolli di routing *multi-hop* su dispositivi mobili commerciali è severamente limitata dai vincoli di sicurezza dei sistemi operativi, che inibiscono l'accesso e la modifica dello stack di rete ai livelli inferiori (MAC/IP).

Per superare tali limitazioni architettoniche, questo lavoro di tesi presenta la progettazione, lo sviluppo e la validazione di un protocollo di routing ibrido operante a livello applicativo (*overlay network*). Sfruttando le API di Google Nearby Connections come livello di astrazione hardware, il sistema implementa una soluzione "zero-hardware" che orchestra nativamente le interfacce Wi-Fi e Bluetooth dei comuni smartphone.

L'algoritmo di instradamento si adatta dinamicamente alle variazioni topologiche alternando due fasi operative. All'interno di cluster stabili (Fase MANET), un meccanismo proattivo elegge un nodo coordinatore e costruisce una topologia logica ad albero mediante un algoritmo *Depth-First Search* (DFS). Questo approccio elimina i *routing loops* e sopprime i *broadcast storms*, abilitando un inoltro rapido e diretto dei pacchetti. In presenza di partizioni di rete o disconnessioni (Fase DTN), il sistema adotta il paradigma *store-carry-and-forward* tramite una variante adattiva del protocollo *Spray and Wait*. Il limite massimo di repliche iniettate nella rete è calcolato

dinamicamente in funzione della cardinalità N della topologia locale nota ($L = \lfloor \sqrt{N} \rfloor$), ottimizzando il compromesso tra probabilità di consegna e consumo di risorse.

Infine, l'architettura integra logiche di *buffer management*, quali l'espulsione FIFO e la soppressione passiva delle ridondanze, per contenere il traffico di rete. Il sistema, validato tramite un'applicazione Android per la messaggistica offline, dimostra la fattibilità di trasformare reti frammentate di smartphone standard in un'infrastruttura *mesh* resiliente, idonea a supportare comunicazioni critiche in totale assenza di infrastrutture di supporto o moduli radio dedicati.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Le infrastrutture di telecomunicazione centralizzate (reti cellulari, Wi-Fi, Internet) costituiscono la spina dorsale della società odierna. Tuttavia, esistono numerosi scenari operativi in cui tale connettività standard risulta inaccessibile, severamente congestionata o strutturalmente assente. Queste situazioni spaziano dai grandi eventi di massa con saturazione delle celle telefoniche (come stadi o concerti), alle aree rurali e montane (*off-grid*[1]), fino agli ambienti sotterranei e ai contesti di emergenza post-disastro. In tutti questi casi, garantire uno scambio di informazioni affidabile e indipendente da un'infrastruttura fissa rappresenta una sfida tecnologica interessante.

In assenza di router e stazioni radio, il paradigma di comunicazione si sposta verso reti decentralizzate e auto-configuranti[2], in cui i dispositivi degli utenti fungono essi stessi da nodi di instradamento. In questo ambito spiccano le reti mobili ad-hoc (**MANET** - *Mobile Ad-hoc Networks*[3]) e le re-

[1]Termine che indica aree geografiche o situazioni operative completamente isolate e prive di accesso alle reti elettriche e di telecomunicazione pubbliche tradizionali.

[2]Reti prive di un'autorità centrale o di un'infrastruttura di base preesistente, in cui i nodi partecipanti si scoprono reciprocamente e stabiliscono connessioni in modo autonomo, distribuito e dinamico.

[3]Reti wireless in cui ogni dispositivo mobile funge sia da terminale (*host*) che da *router*, inoltrando i pacchetti per conto di altri nodi per consentire la comunicazione oltre la portata radio diretta.

ti tolleranti ai ritardi (**DTN** - *Delay-Tolerant Networks*[4]). Poiché le MANET tradizionali richiedono percorsi continui (*end-to-end*) — una condizione spesso irrealistica in scenari caratterizzati da alta mobilità e frammentazione — l'approccio DTN risulta essenziale. Esso introduce il paradigma *store-carry-and-forward* (memorizza, trasporta e inoltra)[5], permettendo ai messaggi di superare le partizioni di rete viaggiando fisicamente nei buffer dei dispositivi in movimento.

Tuttavia, l'implementazione pratica di questi protocolli incontra un ostacolo tecnologico significativo: la maggior parte degli smartphone attualmente in commercio (**COTS** - *Commercial Off-The-Shelf*[6]) impone rigidi vincoli di sicurezza a livello di sistema operativo, inibendo l'accesso e la modifica dei protocolli di instradamento ai livelli inferiori dello stack di rete (MAC/IP).

Per superare tale limite, l'obiettivo e fulcro di questa tesi è la progettazione e lo sviluppo di un **protocollo di routing ibrido operante a livello applicativo (*overlay*)**[7]. Implementato all'interno di un'applicazione di messaggistica offline per dispositivi Android, il sistema astrae l'hardware di rete per creare una topologia logica e ottimizza la consegna dei messaggi alternando dinamicamente due fasi operative:

1. **Instradamento Deterministico (Fase MANET):** Quando la densità dei nodi permette la formazione di cluster stabili, il protocollo elegge un nodo *leader* e costruisce una topologia ad albero tramite

[4] Architetture di rete specificamente progettate per operare in ambienti critici soggetti a frequenti disconnessioni, partizioni di rete prolungate, elevata mobilità dei nodi e tassi di errore significativi.

[5] Modello di inoltro in cui un nodo riceve un messaggio, lo salva in memoria (*store*), si sposta fisicamente nell'ambiente (*carry*) e lo trasmette al nodo successivo (*forward*) solo quando entra in contatto con un potenziale destinatario o relè.

[6] Acronimo indicante dispositivi hardware o componenti software standard, commerciali e ampiamente diffusi nel mercato di massa, in contrapposizione a soluzioni proprietarie sviluppate per scopi militari o industriali di nicchia.

[7] Una rete logica sovrapposta (*overlay network*) creata a livello software, che sfrutta la connettività di base per instradare i dati senza necessitare dei privilegi di sistema (*root*) solitamente richiesti per alterare i protocolli di rete nativi.

un algoritmo *Depth-First Search* (DFS)[8]. Se sorgente e destinatario appartengono allo stesso albero, il routing avviene in modo rapido e diretto.

2. **Trasporto Opportunistico (Fase DTN):** Se il destinatario risulta irraggiungibile all'interno della topologia nota, il sistema passa in modalità DTN adottando un approccio basato sul protocollo *Spray and Wait*[9]. A differenza delle implementazioni standard a numero di repliche fisso[10], il sistema proposto calcola dinamicamente il limite di copie L in funzione dei nodi attualmente presenti nell'albero ($L = \lfloor \sqrt{N} \rfloor$), bilanciando le probabilità di consegna con il consumo di risorse.

Questa architettura ibrida mira a risolvere i colli di bottiglia tipici dei protocolli DTN di base (come l'*Epidemic routing*[11]), limitando drasticamente l'*overhead* di rete e le collisioni di *broadcast*. L'applicazione Android sviluppata funge così da *proof-of-concept* per validare la fattibilità di una messaggistica affidabile su smartphone convenzionali in un'ampia gamma di ambienti destrutturati.

[8] Algoritmo di ricerca in profondità per l'esplorazione dei grafi, che procede il più possibile lungo ogni ramo prima di retrocedere (*backtracking*). Nel contesto di questa architettura, facilita la creazione di una gerarchia di rete priva di cicli chiusi (*loop-free*).

[9] Protocollo di routing DTN che limita il consumo di risorse distribuendo un numero massimo predefinito di copie del messaggio (fase di *spray*) per poi attendere passivamente che uno dei portatori incontri il destinatario finale (fase di *wait*).

[10] Nelle versioni classiche del protocollo, il limite massimo di repliche generate è un parametro statico pre-impostato nel codice, risultando insensibile alla reale densità locale dei nodi in un dato istante.

[11] Approccio di routing di base in cui i nodi replicano e trasmettono ogni messaggio a qualsiasi nuovo vicino incontrato. Pur massimizzando teoricamente la probabilità di consegna, satura rapidamente i buffer e la banda disponibile (*flooding*).

Capitolo 2

Related Works

Per affrontare le sfide imposte dalla comunicazione in scenari privi di infrastrutture fisse, è fondamentale esaminare le soluzioni algoritmiche e tecnologiche attualmente proposte in letteratura. Questo capitolo analizza lo stato dell'arte delle reti mobili ad-hoc e tolleranti ai ritardi, delineando i paradigmi teorici e gli strumenti pratici che costituiscono le fondamenta del presente lavoro di tesi. L'analisi è strutturata in tre sezioni: i fondamenti dei protocolli di routing MANET e DTN, l'analisi di sistemi di emergenza ibridi affini, e infine una panoramica sulla tecnologia abilitante scelta per il livello fisico, ovvero Google Nearby Connections.

2.1 Paradigmi di Routing in Reti MANET e DTN

Nelle reti mobili ad-hoc, i nodi comunicano su canali wireless soggetti a interferenze e senza un'infrastruttura di coordinamento. I protocolli MANET tradizionali si dividono principalmente in due categorie [6, 9]: *proattivi* (come

DSDV[1] e OLSR[2]), che mantengono tabelle di routing aggiornate verso tutti i nodi, e *reattivi* (come AODV[3] e DSR[4]), che scoprono i percorsi solo *on-demand*. Sebbene efficaci in reti dense, tali protocolli falliscono quando la rete è sparsa e subisce disconnessioni frequenti [2], poiché non riescono a stabilire e mantenere un percorso *end-to-end* continuo.

Per ovviare a questo limite, l'architettura **DTN (*Delay-Tolerant Networking*)** adotta un approccio asincrono basato sulla custodia dei messaggi, definiti *bundle*[5]. Nel contesto delle reti DTN, in cui si assume l'assenza di percorsi continui tra sorgente e destinazione, l'instradamento si basa sul paradigma *store-carry-and-forward* (memorizza, trasporta e inoltra) [1].

I protocolli progettati per questo ambiente estremo devono operare decisioni di routing opportunistiche a ogni incontro (o "contatto") tra i nodi mobili. L'obiettivo principale di tali algoritmi è bilanciare un delicato compromesso (*trade-off*): massimizzare il tasso di consegna (*delivery rate*) minimizzando al contempo il consumo delle risorse critiche di rete, in particolare lo spazio di memoria nei *buffer*[6] e la larghezza di banda durante le brevi finestre temporali in cui i dispositivi entrano in contatto [4].

[1] *Destination-Sequenced Distance-Vector*: un protocollo *table-driven* in cui ogni nodo mantiene una tabella di instradamento completa e costantemente aggiornata verso tutte le possibili destinazioni della rete.

[2] *Optimized Link State Routing*: protocollo proattivo che ottimizza la diffusione dei messaggi di controllo eleggendo nodi specifici, chiamati *Multipoint Relays*, per la ritrasmissione.

[3] *Ad hoc On-Demand Distance Vector*: protocollo che crea percorsi solo quando un nodo sorgente ha effettivamente bisogno di trasmettere dati, riducendo l'overhead di controllo.

[4] *Dynamic Source Routing*: simile ad AODV, ma utilizza il *source routing*, un meccanismo per cui l'intero percorso da seguire è specificato nell'intestazione del pacchetto dati.

[5] Il *bundle* è l'unità di protocollo a livello applicativo introdotta dall'architettura DTN. Raggruppa i dati in blocchi di dimensioni eterogenee per facilitarne la memorizzazione prolungata sui nodi intermedi in caso di assenza di link [3].

[6] Poiché i nodi DTN devono custodire i *bundle* anche per lunghi periodi di tempo durante i loro spostamenti fisici (*carry*), la saturazione e il conseguente svuotamento forzato del *buffer* rappresentano una causa importante di perdita dei messaggi.

In base alla strategia adottata per replicare o inoltrare i messaggi (definiti *bundle* nel livello overlay DTN), i protocolli si classificano principalmente in tre macro-categorie [4, 1]:

- **Naive Replication (Replicazione Ingenua):** Questa categoria include i protocolli che generano copie multiple del messaggio e le distribuiscono ai nodi incontrati senza effettuare alcuna valutazione preventiva sulla bontà o affidabilità del nodo ricevente. L'obiettivo principale è massimizzare la probabilità di consegna attraverso la pura ridondanza.

  Un esempio classico di questa categoria è il protocollo **Epidemic**. Il suo funzionamento si basa su una strategia di *flooding*[7] non controllato: quando due nodi entrano in contatto, si scambiano un vettore riassuntivo contenente la lista dei bundle in loro possesso e si trasferiscono reciprocamente i messaggi mancanti. Sebbene in scenari con risorse illimitate questo approccio garantisca il tasso di consegna più elevato e il ritardo minimo, in reti reali si rivela altamente inefficiente. La proliferazione incontrollata delle copie genera un sovraccarico insostenibile (*broadcast storm*[8]), che satura rapidamente la memoria (*buffer*) dei dispositivi intermedi e causa un crollo delle prestazioni, rendendo il protocollo poco scalabile [4, 1].

- **Replicazione Controllata:** Per ovviare ai limiti dell'Epidemic routing, i protocolli a replicazione controllata impongono un limite superiore rigido al numero massimo di copie di un singolo messaggio che possono circolare nella rete.

[7] Tecnica di instradamento in cui un nodo invia il pacchetto dati a tutti i suoi vicini, i quali a loro volta lo ritrasmettono a tutti i propri vicini, allagando di fatto l'intera rete fino a raggiungere la destinazione.

[8] Fenomeno critico in cui le innumerevoli ritrasmissioni simultanee di pacchetti broadcast saturano il mezzo trasmissivo wireless, causando collisioni a catena e paralizzando le comunicazioni della rete.

Il protocollo più noto di questa famiglia è lo **Spray and Wait**. Questo algoritmo mitiga il sovraccarico di rete dividendo il processo di instradamento in due fasi distinte [4]:

1. *Fase Spray:* Il nodo sorgente definisce un numero massimo di copie consentite (L). Negli approcci di tipo "binario", quando il nodo sorgente (o un *relay*[9]) che possiede $L > 1$ copie incontra un vicino sprovvisto del messaggio, cede a quest'ultimo la metà delle copie ($\lfloor L/2 \rfloor$) e ne trattiene per sé l'altra metà ($\lceil L/2 \rceil$). Questo processo si ripete fino a quando un nodo non rimane con una singola copia ($L = 1$) [1].
2. *Fase Wait:* Quando un nodo possiede una sola copia del messaggio ($L = 1$), cessa ogni ulteriore propagazione. Da questo momento in poi, manterrà il pacchetto nel buffer adottando un approccio di instradamento diretto (*Direct Delivery*), ovvero attenderà di incontrare fisicamente il destinatario finale per consegnare il messaggio [4].

Il protocollo ibrido proposto in questa tesi estende e migliora proprio questo concetto: invece di utilizzare un valore L statico, il sistema rende il numero di copie L dinamicamente dipendente dalla cardinalità della topologia locale scoperta in tempo reale, ottimizzando ulteriormente il consumo di risorse.

- **Utility Forwarding (Inoltro basato su Utilità):** A differenza degli approcci basati sulla pura replicazione, questi protocolli mantengono tipicamente un numero molto ridotto di copie (spesso una sola) e decidono a quale nodo inoltrare il messaggio basandosi su una metrica di "utilità". Il messaggio viene delegato a un nodo adiacente solo se quest'ultimo presenta un valore di utilità maggiore verso la destinazione finale [4].

[9] Un nodo intermedio della rete che fa da "staffetta", ricevendo i dati da un nodo e inoltrandoli verso il nodo successivo lungo il percorso verso la destinazione.

Un esempio di spicco che sfrutta metriche storiche è **PROPHET** (*Probabilistic Routing Protocol using History of Encounters and Transitivity*). Questo protocollo parte dall'assunto che i movimenti dei nodi (ad esempio persone o veicoli) non siano puramente casuali, ma seguano schemi parzialmente prevedibili. PROPHET calcola una metrica probabilistica definita *delivery predictability* $P(a, b)$, che stima la probabilità che il nodo a riesca a consegnare un messaggio al nodo b. La metrica si aggiorna secondo tre principi [4, 1]:

1. gli incontri frequenti tra due nodi aumentano il loro valore di prevedibilità reciproca;
2. il valore è soggetto ad "invecchiamento" (*aging*) e decresce col passare del tempo in assenza di contatti;
3. si applica la proprietà transitiva, per cui se il nodo a incontra frequentemente il nodo b, e b incontra frequentemente c, allora b viene considerato un eccellente nodo intermedio per inoltrare messaggi da a verso c.

In questo modo, PROPHET propaga i messaggi prediligendo i nodi che hanno uno storico di incontri più solido con il destinatario.

Come ampiamente dimostrato in letteratura, approcci ibridi che uniscono l'efficienza della fase di scoperta proattiva o reattiva tipica delle MANET con la resilienza del *forwarding* opportunistico DTN, offrono i risultati migliori per garantire la connettività in scenari destrutturati o in situazioni critiche caratterizzate da partizioni di rete [8, 10].

2.2 Sistemi di Emergenza Off-Grid e Lavori Affini

La progettazione e la valutazione di protocolli di instradamento per scenari di emergenza procedono tipicamente attraverso fasi incrementali che

spaziano dalle simulazioni software, all'emulazione, fino ai *testbed* [10] fisici, affrontando sfide pratiche spesso trascurate nei modelli teorici.

2.2.1 Metodologie di Valutazione e Sfide Pratiche

Nella pratica, gran parte del lavoro iniziale sui protocolli MANET e DTN viene svolto tramite simulatori a eventi discreti come NS-2, OMNeT++ o GloMoSim. Ad esempio, studi di valutazione delle performance hanno dimostrato tramite simulazione che le reti puramente MANET (come AODV[11] o OLSR[12]) eccellono nelle comunicazioni intra-gruppo in scenari densi, ma falliscono in reti sparse [1]. In tali contesti avversi, approcci DTN (come *Spray and Wait* o *PRoPHET*) garantiscono invece la consegna dei messaggi, seppur a costo di un maggiore *overhead* e latenza [1, 4]. Tuttavia, la ricerca ha ampiamente evidenziato che le sole simulazioni non sono sufficienti per catturare le criticità imprevedibili delle reti wireless.

Il passaggio ai *testbed* fisici ha infatti rivelato ostacoli significativi. Lavori pionieristici sull'implementazione reale di protocolli reattivi (come AODV) e proattivi (come DSDV) hanno dimostrato che, sul campo, la creazione di rotte *multi-hop* stabili fallisce frequentemente a causa dei cosiddetti *transient links*, ovvero connessioni effimere e canali radio soggetti a *fading* (fluttuazioni repentine del segnale) [2]. Un traguardo fondamentale per mitigare questo problema è stata l'introduzione di filtri basati sulla potenza del segnale o sul rapporto segnale/rumore (SNR) a livello di interfaccia MAC (come nel

[10] Un ambiente di sviluppo e sperimentazione rigoroso e controllato, utilizzato per testare e validare modelli teorici, prototipi hardware o software prima della loro distribuzione ufficiale sul campo.

[11] *Ad hoc On-Demand Distance Vector* (AODV): un protocollo di routing di tipo reattivo che instaura i percorsi solo quando un nodo necessita di trasmettere dati, riducendo l'overhead di controllo ma introducendo una potenziale latenza di scoperta iniziale.

[12] *Optimized Link State Routing* (OLSR): un protocollo di tipo proattivo che mantiene costantemente aggiornate le rotte verso tutti i nodi della rete. Ottimizza la diffusione dei messaggi di topologia delegando l'inoltro a un sottoinsieme ristretto di nodi chiamati *Multipoint Relays* (MPR).

caso del driver modificato *powerwave* per schede wireless commerciali). Tali meccanismi permettono di scartare i "falsi vicini" — nodi con cui la comunicazione risulta asimmetrica o troppo debole — stabilizzando la percezione della topologia di rete prima che questa venga elaborata dal protocollo di routing sovrastante [2].

Per colmare il divario tra simulazioni irrealistiche e testbed fisici eccessivamente costosi da gestire (poiché richiedono lo spostamento fisico di persone o veicoli), la ricerca ha raggiunto un ulteriore traguardo sviluppando emulatori di mobilità basati su griglie statiche. Un esempio è l'emulatore sviluppato sul testbed ORBIT, che modella la mobilità migrando virtualmente lo stato dell'applicazione in esecuzione (tramite snapshot e operazioni di accensione/spegnimento) da un nodo fisico all'altro all'interno di una griglia, permettendo di testare applicazioni DTN reali in modo ripetibile.

2.2.2 L'Evoluzione verso il Routing Ibrido

La consapevolezza che né il paradigma MANET puro né quello DTN puro siano sufficienti in scenari critici ha guidato la ricerca verso soluzioni ibride avanzate. Un primo approccio consiste nell'estendere i messaggi del protocollo AODV affinché, durante la fase di scoperta delle rotte, la rete possa mappare la presenza di router DTN disponibili nell'ambiente circostante. In questo modo, l'applicazione può decidere dinamicamente se stabilire una connessione IP *end-to-end* (se disponibile) o affidare il messaggio ai nodi DTN intermedi per il trasporto asincrono [8].

Altri lavori propongono protocolli ibridi che prendono decisioni di inoltro basate sull'intelligenza computazionale[13]. In situazioni critiche, algoritmi avanzati come il *Simulated Annealing*[14] vengono impiegati per calcolare un

[13]Branca dell'informatica che utilizza algoritmi euristici e modelli ispirati alla natura (come reti neurali, logica *fuzzy* o algoritmi evolutivi) per risolvere problemi complessi di ottimizzazione e tollerare dati imprecisi o incerti, risultando ideale per reti wireless instabili.

[14]Algoritmo di ottimizzazione meta-euristica ispirato al processo termodinamico di ricottura (*annealing*) dei metalli. Esplora lo spazio delle soluzioni accettando temporanea-

"Fattore di Affidabilità" (*Reliability Factor*) per ogni nodo candidato a fungere da *relay* [10]. Questo fattore ottimizza la scelta del *next-hop* valutando parametri in tempo reale quali: lo spazio libero nel buffer del nodo, la velocità media, la direzione del movimento e lo storico dei ritardi. Questi traguardi algoritmici mirano a minimizzare il ritardo di consegna pur operando in un regime di disconnessione estrema [10].

2.2.3 Sistemi di Emergenza basati su Smartphone (Stato dell'Arte)

La necessità di comunicare in assenza di rete cellulare, applicando i concetti teorici sopra descritti, ha portato allo sviluppo di diverse soluzioni ibride recenti per il mercato *consumer*.

Un esempio rilevante è il sistema **LOCATE**, un'applicazione per smartphone Android collegata via Bluetooth Low Energy (BLE)[15] a un modulo radio hardware aggiuntivo basato su tecnologia LoRa (*Long Range*)[16]. LOCATE implementa un algoritmo DTN probabilistico di disseminazione *multi-hop* e sfrutta le eccezionali proprietà di propagazione dell'hardware LoRa per coprire distanze fino a 1 chilometro in ambito urbano e rurale, bilanciando il tempo di risoluzione dell'emergenza con l'*overhead* di rete [11].

In modo simile, altri lavori [7] presentano sistemi che combinano un'app mobile (per interfaccia e sicurezza crittografica) con nodi radio esterni dotati

mente anche percorsi sub-ottimali per evitare di rimanere intrappolato in minimi locali, riducendo gradualmente questa "tolleranza" nel tempo.

[15]Tecnologia di rete personale senza fili (WPAN) progettata per fornire un consumo energetico drasticamente ridotto rispetto al Bluetooth classico, mantenendo un raggio di comunicazione simile. È lo standard di fatto per accoppiare periferiche a basso consumo agli smartphone.

[16]Tecnologia di modulazione a radiofrequenza di tipo LPWAN (*Low Power Wide Area Network*) che permette trasmissioni a lunghissimo raggio con consumi energetici irrisori, a fronte però di una larghezza di banda (*bitrate*) estremamente limitata, idonea solo per messaggi di testo o telemetria.

di firmware open-source *Meshtastic*[17]. Tali sistemi dimostrano eccellenti tassi di consegna, raggiungendo un *Packet Delivery Ratio* (PDR) del 92% fino a 1.2 km in test urbani reali. Oltre alle performance radio, questo sistema affronta il problema della fiducia nelle reti di emergenza implementando una PKI (*Public Key Infrastructure*) leggera per la verifica dell'identità tramite documenti reali e l'assegnazione di ruoli di moderazione, risultando ideale per prevenire la disinformazione in aree colpite da disastri.

Tuttavia, l'uso di tecnologie hardware dedicate come LoRa richiede che ogni utente coinvolto nell'emergenza disponga (o riceva in dotazione) un dispositivo hardware aggiuntivo (dongle) da accoppiare allo smartphone. A differenza di questi sistemi, la soluzione proposta in questa tesi punta a un approccio *zero-hardware*, collocandosi in una rete overlay a livello applicativo e sfruttando esclusivamente le interfacce native presenti sui dispositivi COTS (Wi-Fi e Bluetooth) tramite Google Nearby Connections. Pur sacrificando la portata chilometrica del singolo salto radio, questo approccio garantisce un'accessibilità immediata e pervasiva per gli utenti, abbattendo le barriere d'ingresso in caso di emergenza spontanea semplicemente installando un'applicazione.

2.3 Tecnologia Abilitante: Google Nearby Connections

Google Nearby Connections è un'API sviluppata per facilitare la comunicazione *cross-device* di prossimità in modalità peer-to-peer (P2P)[18] [5].

[17]Progetto open-source che implementa una rete *mesh* decentralizzata su hardware LoRa. A differenza dei moduli LoRa standard usati tipicamente in topologie a stella (come LoRaWAN, che richiede *gateway* centrali connessi a Internet), Meshtastic dota le radio di intelligenza di *routing*, permettendo ai nodi di operare come *relay* paritetici per inoltrare i messaggi l'uno dell'altro (*multi-hop*), creando una vera rete *ad-hoc off-grid*.

[18]Modello di architettura di rete paritetica in cui i nodi comunicano e condividono risorse direttamente tra loro, senza la necessità di appoggiarsi a un server o a un'infrastruttura di coordinamento centrale.

Questa piattaforma offre un livello di astrazione elevato che permette ai dispositivi mobili di rilevarsi e stabilire canali diretti ad alta larghezza di banda e bassa latenza, il tutto senza la necessità di un collegamento a Internet o di un'infrastruttura di rete preesistente. Tali caratteristiche la rendono una soluzione ideale per abilitare interazioni fluide in scenari *off-grid*, come la collaborazione in tempo reale, la condivisione di contenuti e la formazione di gruppi spontanei per emergenze.

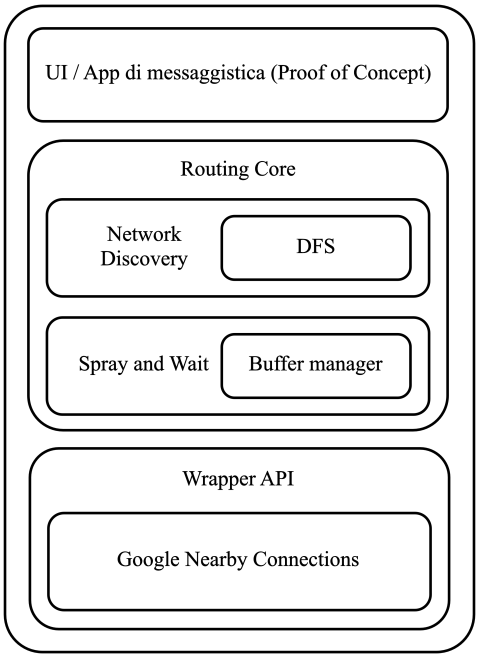

Figura 2.1: struttura a strati del programma

Sotto il cofano, il *framework* unifica e gestisce in modo trasparente diverse tecnologie radio native degli smartphone. Nello specifico, sfrutta protocolli a basso consumo come il Bluetooth (Classic e Low Energy) per le fasi di *Discovery* e *Advertising*[19]. Successivamente, per il trasferimento effettivo dei

[19]Le due fasi iniziali di instaurazione di una connessione *wireless*: l'*advertising* è il processo con cui un nodo trasmette periodicamente pacchetti per annunciare la propria presenza, mentre la *discovery* è l'ascolto passivo o attivo del mezzo per rilevare i dispositivi

dati (*payload*[20]), l'API è in grado di eseguire un *handover*[21] automatico verso tecnologie più performanti come il Wi-Fi Direct o l'attivazione di *hotspot* locali. Un ulteriore vantaggio della piattaforma è la sua natura multipiattaforma, essendo disponibile sia per Android che per iOS, consentendo così la comunicazione nativa tra ecosistemi diversi.

Per adattarsi a vari casi d'uso, Nearby Connections espone diverse strategie di connessione. Per la costruzione di reti decentralizzate, la strategia `P2P_CLUSTER` risulta particolarmente rilevante: essa è progettata per supportare piccole reti *mesh* parziali (topologie M-a-N), consentendo a un singolo nodo di accettare e mantenere contemporaneamente connessioni multiple in entrata e in uscita [5]. Questo si distingue da altre modalità più restrittive offerte dall'API, come `P2P_STAR`, che si limitano a rigide topologie a stella con un singolo *hub* centrale (1-a-N).

Sebbene Google Nearby Connections risolva efficacemente le problematiche legate alla connettività fisica, al risparmio energetico e alla sicurezza del singolo link crittografato, il framework presenta un limite architetturale fondamentale per gli scenari che ci intere: la libreria gestisce in autonomia esclusivamente i socket[22] a singolo salto (1-hop)[23] tra dispositivi fisicamente adiacenti. L'API non fornisce alcun meccanismo nativo di instradamento *multi-hop* o di tolleranza ai ritardi (DTN).

L'analisi congiunta dei protocolli teorici, dei sistemi affini e delle tecnologie abilitanti come Nearby Connections evidenzia dunque un vuoto nelle soluzioni attuali: i *framework* commerciali offrono ottimi collegamenti P2P

adiacenti in fase di *advertising*.

[20]La porzione utile dei dati trasportata all'interno del pacchetto di rete, esclusa l'intestazione (*header*) necessaria per il controllo e l'instradamento del protocollo.

[21]Processo automatico e trasparente all'utente in cui il sistema operativo "passa le consegne" della connessione da una tecnologia radio all'altra per ottimizzare le prestazioni senza far cadere la comunicazione.

[22]Interfaccia software (*endpoint*) che consente a un'applicazione di inviare o ricevere flussi di dati attraverso una rete locale o geografica.

[23]Comunicazione diretta tra due nodi che si trovano fisicamente all'interno della reciproca area di copertura radio, senza l'ausilio di nodi intermedi.

locali ma mancano di logiche di instradamento estese, mentre le soluzioni DTN accademiche o basate su hardware dedicato risultano spesso inaccessibili al grande pubblico. Questa consapevolezza motiva la necessità di sviluppare un protocollo logico e puramente software (*software-only*), capace di posizionarsi al di sopra del livello di connessione fisica fornito dall'API di Google, per trasformare cluster isolati di smartphone in una vera e propria rete *mesh* opportunistica. Le specifiche e il design di tale architettura verranno dettagliate nel Capitolo 3.

Capitolo 3

Architettura del Sistema

Il presente capitolo illustra nel dettaglio l'architettura dell'applicazione sviluppata e le logiche operative del protocollo di instradamento ibrido implementato. L'obiettivo del sistema è fornire una piattaforma di messaggistica decentralizzata, capace di adattarsi dinamicamente alle variazioni topologiche della rete, operando in modo trasparente rispetto all'infrastruttura sottostante.

3.1 Presentazione del sistema

L'architettura software del sistema è stata sviluppata seguendo un rigoroso approccio a strati (*layered architecture*)[1]. Questa scelta progettuale risponde all'esigenza fondamentale di disaccoppiare la logica algoritmica di instradamento (*routing core*) dai dettagli fisici dell'hardware radio, dalla persistenza dei dati e dall'interfaccia utente.

È fondamentale sottolineare, infatti, che l'obiettivo primario di questo lavoro di tesi non risiede nell'applicazione di messaggistica in sé — la quale funge principalmente da *proof-of-concept* e ambiente di validazione — bensì

[1]Modello di progettazione software che organizza il sistema in livelli gerarchici, in cui ogni strato fornisce servizi allo strato superiore e nasconde i dettagli implementativi di quello inferiore, garantendo il principio della separazione delle competenze (*Separation of Concerns*).

nel protocollo di instradamento stesso. Tale modularità strutturale offre un duplice vantaggio: da un lato, garantisce che la tecnologia trasmissiva di base (i livelli inferiori) possa essere aggiornata o sostituita in futuro senza intaccare il motore della rete; dall'altro, rende estremamente agevole l'isolamento e l'esportazione del *routing core*. In questo modo, l'algoritmo sviluppato si configura come un modulo *stand-alone* e riutilizzabile (dopo una breve rielaborazione), sul quale futuri sviluppatori potranno costruire applicazioni software completamente nuove e diversificate, delegando interamente al livello sottostante la complessa gestione delle comunicazioni *multi-hop* in scenari *off-grid*.

3.1.1 Wrapper API e Livello di Trasporto

Per gestire la complessità dei collegamenti diretti a livello fisico sui comuni dispositivi Android, eludendo i vincoli imposti dal sistema operativo sulla creazione di reti *ad-hoc*, il sistema si appoggia alle API di Google Nearby Connections [5]. Tuttavia, per isolare la logica applicativa dalle specificità di tale libreria commerciale, è stato sviluppato un livello di astrazione hardware dedicato (*Wrapper API*)[2].

Questo livello agisce come un insieme di primitive di rete essenziali, occupandosi esclusivamente di tre macro-fasi:

- **Discovery e Advertising:** Astrazione dei meccanismi di rilevamento. Il *wrapper* gestisce l'accensione simultanea e trasparente delle antenne (sfruttando in *background* Bluetooth e Wi-Fi Direct) e l'emissione di *beacon*[3] contenenti identificativi grezzi.

[2]Un *wrapper* (dall'inglese "involucro") è un costrutto software che incapsula funzionalità complesse o interfacce di terze parti, esponendo al resto del sistema un'interfaccia standardizzata e su misura per le specifiche esigenze dell'applicazione.

[3]Brevi pacchetti di controllo trasmessi periodicamente da un nodo per annunciare la propria presenza, il proprio identificativo e il proprio stato operativo ai dispositivi circostanti.

- **Gestione della Topologia Fisica:** Il *wrapper* impone all'hardware l'utilizzo di una strategia di rete a cluster parziale (molti-a-molti). Ciò permette a un singolo smartphone di accettare e mantenere contemporaneamente connessioni multiple sia in entrata che in uscita, un requisito imprescindibile per la costruzione fisica di una rete *mesh*.

- **Trasferimento Payload:** Fornisce primitive asincrone per la ricezione e la trasmissione di flussi di byte. Si occupa inoltre di convertire i dati grezzi provenienti dai *socket* crittografati in strutture dati serializzate[4] (ad esempio in formato JSON[5]) comprensibili dai livelli architetturali superiori.

In sintesi, il *wrapper* fornisce al protocollo ibrido un'interfaccia pulita per inviare pacchetti a un nodo fisico adiacente, mascherando del tutto quale tecnologia radio o frequenza sia effettivamente in uso in quel preciso istante.

3.1.2 Inizializzazione del Nodo

All'avvio dell'applicazione, il sistema configura l'identità del nodo generando un identificativo logico univoco a 8 byte. Questo ID, insieme al nome scelto dall'utente, viene archiviato nella memoria locale del dispositivo per garantire la persistenza dello stato tra le diverse sessioni di utilizzo.

Successivamente, il nodo avvia i servizi di rete in *background* attivando in modo simultaneo le operazioni di *Advertising* (trasmissione della propria presenza tramite *beacon*) e di *Discovery* (scansione dei canali per rilevare altri dispositivi). Per garantire la rapida formazione della rete *mesh*, il protocollo implementa una politica di auto-accettazione: il sistema accetta automaticamente le richieste di connessione provenienti in via esclusiva da dispositivi

[4]La serializzazione è il processo di conversione di un oggetto o di una struttura dati complessa residente in memoria in un formato lineare (come una stringa JSON o un array di byte), idoneo per essere trasmesso su un canale di rete o salvato su disco.

[5]*JavaScript Object Notation*: un formato standard leggero e leggibile sia dalle macchine che dagli umani, ampiamente utilizzato per lo scambio di dati strutturati tra sistemi eterogenei.

che trasmettono il medesimo *Service ID*[6]. Questo approccio permette di instaurare i collegamenti fisici iniziali tra i nodi in modo del tutto trasparente, senza richiedere alcun intervento manuale o di *pairing* da parte dell'utente.

3.1.3 Connessione e Handshake

Non appena il livello hardware notifica l'avvenuta creazione di un canale di comunicazione fisico tra due nodi (collegamento a singolo salto o 1-*hop*), i dispositivi avviano la procedura logica di *Handshake*[7].

L'astrazione hardware, infatti, gestisce le connessioni assegnando identificativi fisici effimeri e casuali[8]. Per tradurre la rete fisica in una rete logica immutabile, il nodo appena connesso invia immediatamente un pacchetto di controllo di tipo *HELLO*[9]. Questo pacchetto contiene l'identificativo logico univoco del nodo e i suoi metadati (come il nome utente e lo stato del *buffer*).

Il nodo ricevente analizza il pacchetto ed esegue un *binding*[10] all'interno delle proprie tabelle di routing interne, associando l'ID del *socket* fisico all'ID logico del nodo. Questa mappatura è un passaggio architetturale cruciale: permette ai livelli superiori del protocollo di ignorare gli indirizzi fisici

[6]Un identificativo univoco (solitamente una stringa o un UUID) utilizzato a livello radio per filtrare i dispositivi circostanti. Permette all'applicazione di ignorare *smartwatch*, auricolari o altri dispositivi Bluetooth irrilevanti, tentando l'accoppiamento solo con gli smartphone che eseguono il medesimo protocollo.

[7]Fase di negoziazione preliminare tipica dei protocolli di rete, in cui due nodi, appena stabilito un canale fisico, si scambiano i parametri operativi di base e verificano le reciproche identità prima di iniziare il trasferimento dei dati utili (*payload*).

[8]I moderni sistemi operativi mobili (come Android e iOS) implementano tecniche di *MAC randomization* per prevenire il tracciamento degli utenti, alterando costantemente l'indirizzo hardware (MAC address) dell'interfaccia radio. Pertanto, i protocolli *overlay* non possono fare affidamento su di esso per l'instradamento nel lungo periodo.

[9]I messaggi di *HELLO* (o *keep-alive*) sono un costrutto standard nei protocolli MANET [9]. Vengono utilizzati per notificare la propria presenza a un nuovo vicino o per confermare periodicamente che un collegamento radio esistente è ancora attivo.

[10]Processo di associazione logica (mappatura) in memoria. In questo contesto, lega il canale di comunicazione temporaneo fornito dall'API di Google all'identità persistente del nodo all'interno della topologia *mesh*.

temporanei e di instradare i pacchetti basandosi esclusivamente sulle identità logiche stabili. L'inserimento di un nuovo vicino logico invalida la conoscenza della topologia di rete corrente, innescando un ricalcolo globale dei percorsi e l'avvio della fase di scoperta (*Discovery Phase* avanzata).

3.1.4 Disconnessione e Fallback

Le reti mobili *ad-hoc* sono intrinsecamente volatili e soggette a continui mutamenti topologici. Quando un nodo esce dal raggio di copertura radio o disattiva la propria interfaccia di rete, il livello hardware intercetta l'evento di disconnessione (caduta del *link*). Il sistema reagisce espungendo istantaneamente il nodo e l'intero sotto-albero dei suoi discendenti dalle tabelle di instradamento locali, annullando il relativo *binding* fisico-logico.

La rottura di un collegamento critico può provocare la segmentazione della rete in partizioni isolate (fenomeno noto come *network partitioning*[11]) o la creazione di nodi "orfani". Tale evento innesca una procedura sincrona di ri-elezione del nodo coordinatore (*Leader*) e il ricalcolo immediato della topologia per i dispositivi rimasti all'interno del *cluster* connesso.

Dal punto di vista del protocollo, l'impossibilità di raggiungere la destinazione tramite un percorso *multi-hop* continuo innesca il meccanismo di *fallback* verso la logica DTN [8]. Invece di scartare (*drop*) i messaggi destinati a un nodo non più raggiungibile, il sistema li deposita all'interno di un *buffer* locale adottando il paradigma *store-carry-and-forward*. Tuttavia, questo inoltro opportunistico non avviene in modo casuale o epidemico, bensì è rigorosamente governato da una variante adattiva del protocollo *Spray and Wait*.

Questo approccio ibrido e adattivo garantisce la resilienza della comunicazione anche in scenari di frammentazione estrema, arginando l'*overhead*

[11]Condizione in cui una singola rete si divide in due o più sottomanglie (o *cluster*) incapaci di comunicare tra loro a causa dell'assenza di nodi intermedi che facciano da ponte. Nelle MANET tradizionali, questo evento interrompe irrimediabilmente i flussi di dati tra le partizioni.

tipico delle reti puramente opportunistiche.

3.2 Funzionamento del Protocollo di Routing

Il cuore dell'architettura risiede nel protocollo ibrido che opera al di sopra dei collegamenti diretti, combinando un approccio deterministico per le reti connesse e un approccio opportunistico per le partizioni [8, 10].

3.2.1 Fase Proattiva: Elezione del Leader

Per poter instradare i messaggi in modo efficiente ed evitare cicli (*routing loops*)[12] infiniti, i nodi fisicamente connessi si auto-organizzano in una struttura gerarchica ad albero. Il primo passo è l'elezione di un nodo *Root* (Leader), necessario per coordinare la mappatura della rete senza causare collisioni o esplorazioni ridondanti [6].

Il meccanismo si basa su un timer di inattività (*heartbeat*[13]). Se un nodo non riceve aggiornamenti topologici per un lasso di tempo prestabilito, si auto-elegge Leader: crea una struttura dati rappresentante la radice dell'albero, genera un *Token*[14] di esplorazione associandovi il proprio *timestamp*[15] locale e lo inoltra in *broadcast* ai vicini fisici.

Nel caso in cui due o più nodi facciano scadere il timer simultaneamente e propaghino contemporaneamente un *Token*, la rete risolve la collisione in

[12]Condizione d'errore in cui un pacchetto viene inoltrato continuamente tra un gruppo di nodi senza mai raggiungere la destinazione, consumando inutilmente larghezza di banda e risorse dei dispositivi.

[13]Un segnale periodico trasmesso tra i nodi per confermare la reciproca presenza e lo stato di attività. In assenza di *heartbeat* per un tempo di *timeout* predefinito, il sistema deduce che il collegamento o il nodo coordinatore non sono più disponibili.

[14]Un pacchetto di controllo speciale che circola nella rete per concedere il diritto di eseguire determinate operazioni o per trasportare informazioni di stato globali, garantendo che l'esplorazione della topologia avvenga in modo ordinato.

[15]Una sequenza di caratteri o informazioni codificate che indicano l'ora e la data esatta in cui si è verificato un determinato evento, utilizzata nei sistemi distribuiti per ordinare gli eventi in modo cronologico.

modo deterministico: i nodi confrontano i *timestamp* dei pacchetti ricevuti. Il pacchetto con il *timestamp* inferiore (ovvero l'elezione iniziata temporalmente prima) ha la priorità, invalidando l'elezione concorrente. A parità di *timestamp*, prevale il nodo con l'identificativo alfanumerico maggiore. Questo rigore garantisce che, indipendentemente dal numero di conflitti iniziali, la rete converga sempre verso un unico coordinatore stabile.

3.2.2 Scoperta della Topologia tramite Algoritmo DFS

Una volta stabilito il Leader, si innesca la fase di esplorazione del grafo di rete (fase di *Seek* o *Explore*). L'obiettivo di questa fase è mappare le connessioni fisiche a singolo salto (1-*hop*) — che possono formare maglie ridondanti e causare cicli di instradamento (*routing loops*)[16] — in una struttura logica gerarchica ad albero, intrinsecamente priva di cicli (*loop-free*) e ottimizzata per l'instradamento deterministico [9].

Il veicolo di questa esplorazione è un *Token*[17] generato dal Leader, il quale viaggia di nodo in nodo trasportando un oggetto serializzato in formato JSON. Tale oggetto rappresenta e traccia in tempo reale lo stato dell'albero in costruzione (es. `NODO_A[NODO_B[?], NODO_C[?]]`).

L'algoritmo distribuito si basa su una Ricerca in Profondità (*Depth-First Search* - DFS)[18] e la sua esecuzione pratica si divide in diverse casistiche, valutate autonomamente da ciascun nodo alla ricezione del Token:

[16]Come già accennato, i cicli rappresentano un problema critico nelle reti *ad-hoc*. Un pacchetto instradato all'interno di un ciclo continuerebbe a transitare all'infinito tra gli stessi nodi, sprecando la banda e le risorse energetiche dei dispositivi mobili.

[17]In questo contesto, il *Token* funge da messaggio di stato globale. A differenza di un pacchetto *broadcast* che si espande in tutte le direzioni (*flooding*), il Token è univoco e viene passato in modo sequenziale (unicast) da un nodo all'altro, garantendo un'esplorazione ordinata e serializzata del grafo di rete.

[18]Algoritmo di esplorazione non informata dei grafi che procede il più possibile in profondità lungo un singolo ramo prima di retrocedere per esplorare i rami alternativi. Risulta particolarmente efficiente per esplorare reti *wireless* mantenendo un basso livello di traffico di controllo (*overhead*).

1. **Inizializzazione (Azione del Leader):** Il Leader individua i propri vicini fisici, li inserisce nel Token contrassegnandoli come nodi da esplorare (identificati da uno stato pendente, es. "?"), e invia una copia del Token al primo vicino disponibile.

2. **Ricezione e Validazione (Definizione del Parent):** Quando un nodo generico riceve il Token per la prima volta, analizza la struttura JSON. Il nodo riconosce l'identificativo fisico del mittente da cui ha ricevuto il pacchetto e lo registra in modo permanente all'interno delle proprie tabelle locali come proprio nodo "Padre" (*Parent Node*). Questo passaggio stabilisce un vincolo gerarchico univoco verso la radice dell'albero, prevenendo la formazione di percorsi circolari.

3. **Valutazione dello Stato ed Espansione (Caso A - Propagazione):** Il nodo calcola la differenza insiemistica tra i propri vicini fisici attualmente rilevati a livello radio e l'insieme dei nodi già visitati (*visited_nodes*) trascritti nell'albero globale trasportato dal Token.

    - Se l'insieme risultante **non è vuoto**, significa che esistono vicini non ancora esplorati dalla rete. Il nodo ne seleziona uno, lo registra nella propria tabella locale come "Figlio" (*Child Node*), aggiorna la struttura JSON del Token aggiungendo questo nuovo ramo e trasmette il Token *esclusivamente* a questo figlio, delegandogli il proseguimento dell'esplorazione in profondità.

4. **Chiusura del Ramo e Backtracking (Caso B - Ritorno):**

    - Se l'insieme dei vicini non visitati **è vuoto** (situazione che si verifica se il nodo è una "foglia" periferica della rete, oppure se tutti i suoi vicini fisici sono già stati mappati in altri rami dell'albero), il nodo non ha più direzioni utili in cui espandere la ricerca.

- A questo punto scatta il meccanismo di *backtracking*[19]: il nodo modifica lo stato del proprio sotto-albero nel Token indicandolo come completato (es. assegnando lo stato `READY`) e restituisce il Token al proprio nodo Padre.

5. **Compilazione Tabelle di Routing (Risalita):** Durante la fase di *backtracking*, quando un nodo Padre riceve indietro il Token da un proprio Figlio, estrapola dalla struttura JSON tutti i discendenti presenti in quel ramo specifico appena chiuso. Il Padre aggiorna la propria tabella di routing verso il basso (*downstream routing table*), associando tutti quegli ID logici (nipoti e pronipoti) all'ID fisico di quel preciso Figlio, creando così le regole di inoltro definitive.

6. **Terminazione dell'Algoritmo:** Il processo di andata e ritorno continua ricorsivamente finché il Token non viene restituito al Leader originario. Quando il Leader riceve il Token e constata di non avere a sua volta ulteriori vicini fisici inesplorati, l'esplorazione DFS è ufficialmente conclusa e l'albero di instradamento è pienamente operativo.

Utilità della Struttura ad Albero e Risoluzione del Routing

Il risultato finale di questa fase proattiva è la transizione della rete da un grafo fisico caotico a una mappa logica rigorosa. L'albero di instradamento così costruito svolge due funzioni architetturali fondamentali:

- **Eliminazione dei Broadcast Storms:** Avendo trasformato un grafo fisico a maglie (*mesh*) in un albero, per definizione esiste sempre ed esclusivamente un solo percorso valido tra due nodi qualsiasi della re-

[19]Tecnica algoritmica di "ritorno sui propri passi". Quando l'algoritmo raggiunge un vicolo cieco (un nodo senza ulteriori vicini inesplorati), retrocede al nodo precedente (il Padre) per verificare la presenza di altri percorsi alternativi.

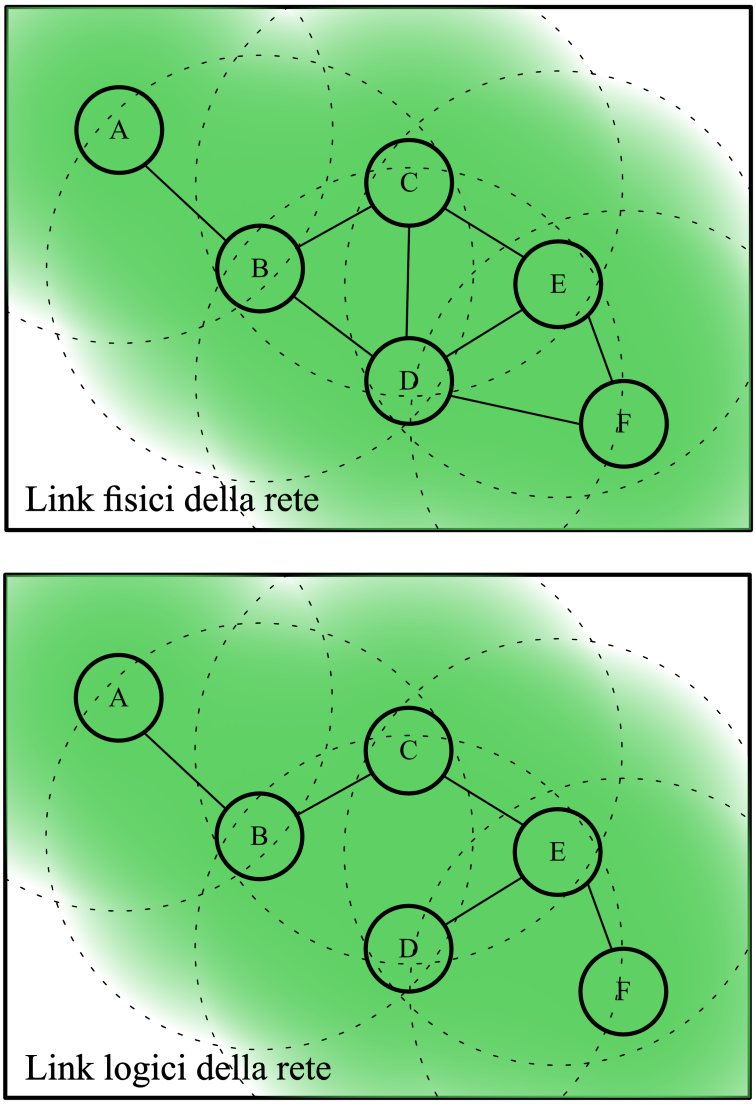

Figura 3.1: collegamenti fisici e collegamenti logici della rete spontanea

te[20]. Ciò elimina del tutto la necessità di inondare la rete (*flooding*) per recapitare i messaggi, minimizzando le collisioni radio e abbattendo drasticamente il consumo di banda.

- **Instradamento Immediato:** Ogni nodo possiede ora una consapevolezza topologica completa. Quando un utente deve inviare un messaggio, il sistema verifica se il destinatario appartiene all'albero noto. In caso affermativo, il nodo sa esattamente quale *next-hop*[21] utilizzare: se la destinazione figura tra i propri discendenti, il pacchetto viene inoltrato al Figlio appropriato lungo quel ramo; se invece non è un discendente, il pacchetto viene delegato verso l'alto al Padre, che a sua volta reitererà la medesima logica.

Esempio Operativo di Instradamento Per chiarire la fluidità di questo meccanismo, si consideri una topologia in cui il nodo **A** (Leader/Radice) ha due figli, **B** e **C**. A sua volta, il nodo **B** ha un figlio, **D**.

- **Routing verso il basso (*Downstream*):** Se il nodo **A** vuole comunicare con **D**, consulta la propria tabella e vede che **D** si trova nel ramo gestito da **B**. Il *next-hop* scelto sarà quindi **B**. Alla ricezione, **B** consulterà la propria tabella, vedrà che **D** è un suo figlio diretto e gli consegnerà il pacchetto.

- **Routing trasversale/verso l'alto (*Upstream*):** Se il nodo **D** vuole inviare un messaggio a **C**, consulterà la sua tabella. Non avendo figli (essendo una "foglia"), delegherà il pacchetto al proprio Padre, **B**. Il nodo **B** controllerà se **C** è un suo discendente; non essendolo, inoltrerà

[20]Nella teoria dei grafi, un albero è un grafo non orientato, connesso e aciclico. Una proprietà fondamentale di questa struttura è che un albero con N nodi possiede esattamente $N-1$ archi: ciò garantisce l'assenza di percorsi ridondanti e cicli chiusi.

[21]Il "prossimo salto": in telecomunicazioni indica il nodo router immediatamente adiacente a cui un pacchetto deve essere fisicamente inoltrato affinché possa proseguire il suo viaggio verso la destinazione finale.

il pacchetto al proprio Padre, **A**. Infine, **A** vedrà che **C** è un suo figlio diretto e concluderà l'instradamento inviandogli il pacchetto.

Solo quando questa precisa fase deterministica fallisce, ovvero nel caso in cui il destinatario ricercato non figuri in alcun ramo dell'albero esplorato, sintomo di una partizione di rete, il sistema delegherà la gestione e la consegna del messaggio al protocollo opportunistico e asincrono *Delay-Tolerant* (*Spray and Wait* adattivo).

3.2.3 Fase Opportunistica: DTN (Spray and Wait Adattivo)

Il sistema transita verso il paradigma *store-carry-and-forward*, tipico delle reti *Delay-Tolerant Network* (DTN), in due scenari specifici: quando il nodo destinatario non figura all'interno dell'ultimo albero topologico validato (esito negativo della fase di ricerca DFS), oppure quando l'instradamento deterministico *multi-hop* fallisce a causa della rottura improvvisa di un *link* fisico. In queste circostanze, i pacchetti vengono contrassegnati per l'elaborazione opportunistica.

La scelta del protocollo di instradamento per questa fase è scaturita da un'attenta analisi della letteratura [4], pesata sulle peculiarità degli scenari di emergenza e sul vantaggio tattico offerto dalla topologia parziale già nota. Nello specifico, approcci basati sulla *Naive Replication* (come l'*Epidemic Routing*) massimizzano teoricamente la probabilità di consegna, ma saturano rapidamente la banda e la memoria dei dispositivi intermedi, innescando fenomeni di *broadcast storm* che fanno collassare l'intera rete [1].

D'altro canto, i protocolli basati sull'*Utility Forwarding* (come *PROPHET* o *MaxProp*) operano scelte di instradamento calcolando metriche probabilistiche basate sullo storico degli incontri passati tra i nodi [4]. Tuttavia, in scenari di emergenza spontanei, la mobilità delle persone è caotica e la rete si forma *ex novo*: i dispositivi sono privi di uno storico pregresso degli incontri, rendendo di fatto inefficaci e inapplicabili tali metriche probabilistiche [10].

Fase dell'Algoritmo	Stato JSON del Token	Nodo Attivo	Azione Eseguita (Logica e Instradamento)
1. **Inizializzazione**	`A[ B[?], C[?] ]`	**Nodo A**	Il Leader individua i vicini fisici (B e C), li imposta come pendenti (`?`) e cede il Token al primo nodo disponibile (B).
2. **Espansione DFS**	`A[ B[ D[?] ], C[?] ]`	**Nodo B**	Il nodo B riceve il Token, riconosce A come Padre. Rileva un nuovo vicino inesplorato (D), aggiorna il JSON e inoltra il Token a D.
3. **Raggiungimento Foglia**	`A[ B[ D[READY] ], C[?] ]`	**Nodo D**	Il nodo D riconosce B come Padre. Non avendo altri vicini inesplorati (è una foglia), marca il suo stato come `READY` e innesca il *backtracking* rimandando il Token a B.
4. **Backtracking (Ramo 1)**	`A[ B[READY], C[?] ]`	**Nodo B**	B riceve il Token di ritorno. Associa l'ID di D (suo discendente) al collegamento fisico. Non avendo altri vicini, si marca come `READY` e rimanda ad A.
5. **Esplorazione (Ramo 2)**	`A[ B[READY], C[?] ]`	**Nodo A**	A riceve il Token di ritorno da B. Compila la tabella di routing per il ramo B. Notando che C è ancora pendente, inoltra il Token a C.
6. **Completamento**	`A[ B[READY], C[READY] ]`	**Nodo A**	Dopo il ritorno del Token anche da C, A constata l'assenza di ulteriori rami (`?`). L'esplorazione termina e l'albero è validato.

Tabella 3.1: Evoluzione dello stato del Token JSON e relative azioni dei nodi durante l'esplorazione della topologia tramite algoritmo DFS.

Per questo motivo, si è scelto di implementare una variante adattiva del protocollo **Spray and Wait**. Tale approccio bilancia i due estremi: limita rigorosamente il traffico definendo un tetto massimo di copie circolanti (Fase di *Spray*) e demanda il recapito finale all'incontro diretto con il destinatario (Fase di *Wait*).

La vera innovazione dell'approccio proposto risiede nel superamento del maggiore limite dello *Spray and Wait* classico: l'impiego di un limite statico di copie L [1]. Il sistema progettato sfrutta la *topology awareness*[22] acquisita a costo zero durante la fase proattiva. Il nodo sorgente, conoscendo la dimensione esatta della partizione di rete (il *cluster*) in cui si trova, calcola dinamicamente il numero di copie L al momento dell'iniezione del pacchetto nella rete, utilizzando la formula:

$$L = \lfloor \sqrt{N} \rfloor$$

dove N rappresenta la cardinalità (il numero totale di nodi) dell'albero connesso.

Questa euristica garantisce un'allocazione delle risorse auto-scalabile: in un *cluster* molto vasto (es. $N = 100$), il protocollo inietterà 10 copie, aumentando la pervasività della ricerca per massimizzare le probabilità di successo; al contrario, in reti estremamente sparse o frammentate (es. $N = 4$), genererà appena 2 copie, adottando un comportamento conservativo che previene l'esaurimento istantaneo dei *buffer* locali.

L'innesco della fase opportunistica si differenzia operativamente in base al momento in cui viene rilevato il fallimento dell'instradamento deterministico. Questa distinzione dà luogo a due scenari di iniezione dei pacchetti profondamente diversi in termini di efficienza:

- **Scenario 1: Assenza topologica del destinatario.** Se il nodo destinatario non figura nell'ultimo albero logico validato, l'algoritmo *Spray*

[22] La "consapevolezza topologica": la capacità di un nodo o di un protocollo di conoscere in tempo reale la struttura, l'estensione e lo stato dei collegamenti della porzione di rete in cui è attualmente inserito.

and Wait viene innescato direttamente dal nodo sorgente al momento della creazione del messaggio. Sebbene questa situazione risulti intrinsecamente meno ottimizzata — in quanto l'esplorazione opportunistica parte in modo agnostico rispetto alla reale posizione del target — essa garantisce comunque una concreta possibilità di consegna del *bundle*, laddove i protocolli *ad-hoc* tradizionali si limiterebbero a notificare un errore e scartare il pacchetto alla fonte.

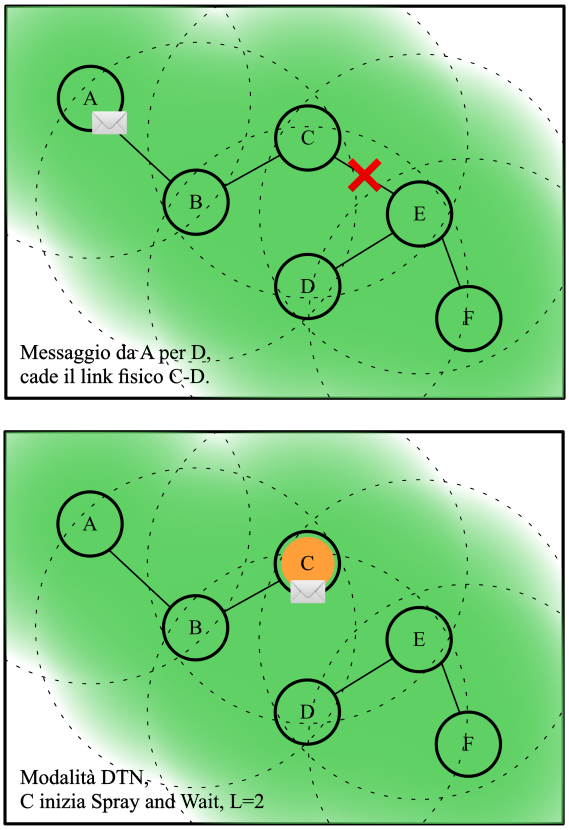

Figura 3.2: posizione di avvio della fase DTN in caso di rottura di un link fisico

- **Scenario 2: Rottura di un *link* durante l'inoltro.** Se un percorso *multi-hop* valido si interrompe improvvisamente a causa della mobilità dei nodi mentre il pacchetto è già in transito, il sistema adotta una

strategia conservativa. L'ultimo nodo che rileva l'interruzione (ovvero il dispositivo che constata l'irreperibilità del proprio *next-hop* designato) non scarta il *bundle* né lo rispedisce al mittente, bensì lo trattiene nel proprio *buffer* assumendosi l'onere di avviare la fase DTN. Questa dinamica offre un immenso vantaggio tattico: nell'ultima topologia nota, questo nodo intermedio si trovava, per definizione dell'algoritmo DFS, sullo stesso ramo logico del destinatario o ne era addirittura un vicino diretto (Padre o Figlio). Di conseguenza, la fase di *Spray* ha inizio da un "punto di caduta" fisicamente o logicamente molto più prossimo alla destinazione finale. Questo parziale avvicinamento spaziale aumenta drasticamente la probabilità di un incontro utile, rendendo la consegna opportunistica notevolmente più rapida ed efficiente.

Gestione del Buffer e Prevenzione della Congestione

Nelle reti DTN, i nodi intermedi agiscono da "custodi" (*custodians*) del *payload*. Tuttavia, operando su dispositivi commerciali (COTS) caratterizzati da risorse condivise e limitate, l'allocazione di memoria per i messaggi in sospeso è rigorosamente contingentata tramite una coda dedicata (*DTN Queue*), dimensionata a una capienza massima prefissata. Per mitigare i fenomeni di congestione e massimizzare il tasso di consegna (*delivery rate*), il sistema implementa tre logiche avanzate per la gestione del ciclo di vita dei pacchetti in memoria:

- **Espulsione con Delega (Eviction FIFO):** In caso di saturazione del *buffer* locale, l'architettura non si limita a scartare il messaggio in coda da più tempo (approccio *drop-tail*[23]). Prima di deallocare la memoria, il nodo applica una politica di "staffetta": delega il pacchetto più vecchio a un vicino fisico scelto casualmente (ad esempio tra i nodi Figlio o il Padre nell'albero). Questa euristica garantisce la sopravvivenza della

[23]Politica di gestione delle code tipica dei *router* tradizionali, che consiste nello scartare irrimediabilmente i nuovi pacchetti in arrivo, o i più vecchi in giacenza, quando il *buffer* raggiunge la sua capacità massima.

singola copia del messaggio all'interno della rete globale, liberando al contempo lo spazio locale necessario per accogliere pacchetti più recenti.

- **Re-iniezione (Transizione DTN-MANET):** Rappresenta il punto di convergenza architetturale tra l'approccio opportunistico e quello deterministico. Al termine di ogni ciclo di scoperta DFS, il nodo esegue un'iterazione sull'intera coda DTN locale. Effettua un confronto tra l'identificativo del destinatario di ciascun messaggio in latenza con i nodi presenti nella nuova mappa topologica appena validata. Se un destinatario precedentemente *offline* risulta nuovamente raggiungibile all'interno dell'albero logico, il pacchetto viene estratto dal *buffer*, privato del contrassegno DTN e re-iniettato nel normale flusso di *routing* deterministico per l'immediato recapito.

- **Soppressione Passiva (Implicit ACK e Prevenzione delle Ridondanze):** L'implementazione di pacchetti di riscontro (*Acknowledgment* - ACK) *end-to-end* espliciti in reti DTN genera un traffico di controllo (*overhead*) insostenibile. Il controllo delle ridondanze avviene pertanto in modo passivo, sfruttando la rigorosa gerarchia dell'albero DFS e delegando la soppressione ai nodi intermedi.

  Si consideri la seguente casistica fondamentale: a seguito di una scoperta DFS, il nodo A rileva che il destinatario di un pacchetto in giacenza nella propria coda DTN è tornato *online*. Il nodo A avvia la re-iniezione inoltrando il messaggio verso il *target*. Successivamente, un nodo B (Figlio di A nell'albero topologico), che possiede nel proprio *buffer* una copia del medesimo messaggio (generata in precedenza durante la fase di *Spray*), si accorge a sua volta della presenza del destinatario e tenta di instradarvi il pacchetto. Poiché il *routing* deterministico impone che i pacchetti originati da B transitino obbligatoriamente per il Padre A per raggiungere quel ramo della rete, A intercetterà la richiesta di inoltro. Esaminando l'intestazione, A verifica che l'identificativo univoco

(UUID[24]) del pacchetto in transito coincide con quello del messaggio appena estratto dalla propria coda locale. Di conseguenza, il nodo A sopprime in modo silente la copia proveniente da B, bloccandone l'inoltro.

Analizzando la dinamica dal punto di vista del nodo B, esso ha semplicemente delegato l'inoltro al nodo genitore, ignorando che quest'ultimo abbia bloccato la propagazione per ridondanza (comportandosi di fatto come un *Implicit ACK*). La conseguenza architetturale di questa logica è che **l'unica copia del messaggio che raggiungerà il destinatario proverrà dal nodo "custode" topologicamente più vicino alla destinazione** al momento della riconnessione. Tutte le altre repliche, nel tentativo di convergere verso il medesimo *target* risalendo i rami dell'albero, verranno sistematicamente intercettate ed eliminate dai nodi intermedi che hanno già provveduto all'inoltro. Questo meccanismo recide sul nascere qualsiasi rischio di inondazione del canale (*flooding*), garantendo al contempo l'assoluta affidabilità della consegna.

[24] *Universally Unique Identifier*: un identificatore standardizzato a 128 bit utilizzato nello sviluppo software per identificare in modo univoco le informazioni all'interno di sistemi distribuiti, senza la necessità di un coordinamento centrale.

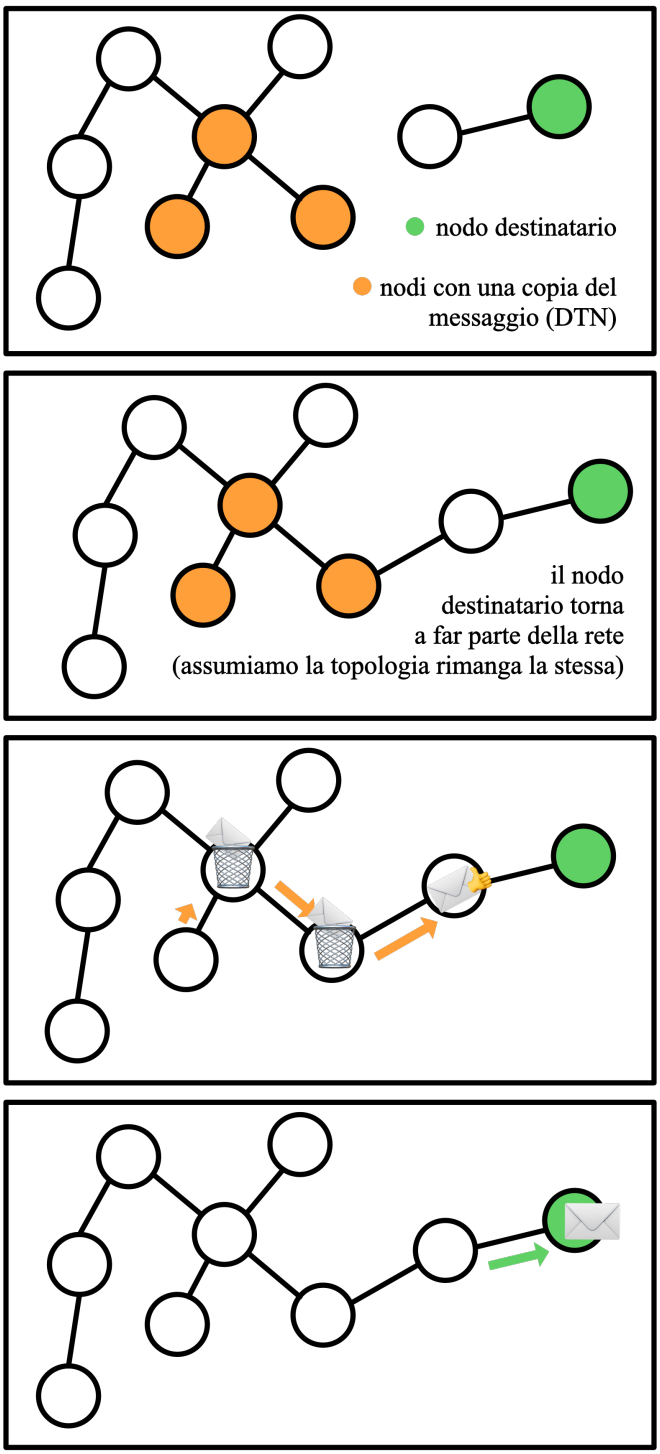

Figura 3.3: fase di Wait e successiva consegna del messaggio

Capitolo 4

Sperimentazione e Implementazione

Questo capitolo descrive i dettagli implementativi dell'applicazione Android sviluppata come *proof-of-concept* per validare il protocollo di routing ibrido presentato nel capitolo precedente. L'obiettivo dell'implementazione è stato quello di creare un sistema robusto, modulare e in grado di operare interamente *off-grid*, fungendo da *testbed* reale per valutare le prestazioni degli algoritmi su dispositivi COTS. Verranno illustrate le tecnologie adottate, l'architettura software a livelli, il motore di instradamento DTN, la gestione della concorrenza e le scelte relative all'interfaccia utente.

4.1 Tecnologie Utilizzate

L'applicazione è stata sviluppata nativamente per il sistema operativo Android utilizzando **Kotlin** come linguaggio di programmazione principale. La scelta di Kotlin, rispetto al tradizionale Java, è motivata dal supporto nativo alle *Coroutines*[1]. In un ambiente di rete distribuito e decentralizzato, le comunicazioni di rete presentano una natura fortemente asincrona: ricezione

[1]Un costrutto introdotto in Kotlin che consente di eseguire codice asincrono in modo non bloccante. Rispetto ai *thread* tradizionali, le *coroutines* sono estremamente più leggere ed efficienti in termini di allocazione di memoria, permettendo l'esecuzione di molte

dei *payload*, ricalcolo della topologia e operazioni di I/O su disco devono avvenire concorrentemente senza bloccare il *Main Thread* dell'interfaccia grafica. L'uso dei `Dispatchers`[2] (in particolare `Dispatchers.IO` per il disco/rete e `Dispatchers.Default` per i calcoli piu pesanti da eseguire velocemente) ha permesso di giostrare questi task correttamente.

Per la costruzione dell'interfaccia utente (UI) è stato adottato **Jetpack Compose**, il moderno toolkit dichiarativo di Google[3]. La sua natura reattiva si sposa perfettamente con l'architettura basata sugli stati della rete: la UI si ri-disegna automaticamente non appena l'albero di routing subisce una mutazione. Infine, per la persistenza dei dati e la serializzazione degli oggetti di rete è stata impiegata la libreria **Gson** per gestire gli oggetti strutturati e destrutturati nel migliore dei modi, convertendo le complesse strutture dati Kotlin in stringhe JSON standardizzate e viceversa.

4.2 Architettura Software a Livelli

Per garantire il principio di Separation of Concerns, l'architettura dell'applicazione è stata strutturata in quattro livelli principali. Questa modularità disaccoppia il motore del protocollo algoritmico dalle specificità dell'interfaccia utente e dai driver dell'hardware radio, rendendo il codice astratto e potenzialmente portabile su altri sistemi operativi o su applicazioni diverse che necessitano lo stesso sistema di comunicazione.

operazioni concorrenti senza esaurire le risorse di sistema.

[2]Componenti dell'ecosistema delle Coroutines che determinano su quale *thread* o *pool* di *thread* verrà eseguita una specifica operazione asincrona.

[3]A differenza del precedente sistema Android per la UI basato su file XML, in cui l'interfaccia grafica veniva definita imperativamente e separata dalla logica applicativa, Jetpack Compose adotta un approccio dichiarativo. L'interfaccia viene descritta definendo il suo stato in un unico linguaggio (Kotlin), e il framework si occupa di ri-disegnarla in modo selettivo (*recomposition*) solo quando tale stato subisce una variazione.

4.2.1 Livello Interfaccia Utente (UI Layer)

Il livello di presentazione è stato sviluppato adottando il *pattern* architetturale MVVM [4] tramite la classe `MainViewModel`. Il livello UI non contiene alcuna logica di business, ma si limita a sottoscrivere e osservare gli stati emessi dai livelli inferiori tramite flussi reattivi, implementati con `StateFlow` per gli stati persistenti (come la topologia attuale) e `SharedFlow` per gli eventi *one-shot* (come le notifiche di ricezione messaggi).

L'interfaccia mantiene liste reattive che classificano i nodi scoperti in tre categorie logiche:

- **Nodi online con chat recenti:** Dispositivi attualmente presenti nell'albero di routing con cui l'utente ha già scambiato messaggi.
- **Nuovi nodi online:** Dispositivi appena scoperti e validati dal protocollo DFS, pronti per la comunicazione.
- **Storico offline:** Nodi con cui si è comunicato in passato, ma che al momento risultano disconnessi o irraggiungibili.

Contestualmente all'inizializzazione del layer grafico, il ViewModel invoca l'avvio dei servizi di rete in *background*, garantendo che il processo di *Discovery* sopravviva ai cambi di configurazione del dispositivo [5].

[4] Acronimo di *Model-View-ViewModel*. È un pattern architetturale che separa nettamente l'interfaccia grafica (*View*) dalla logica di business e dai dati (*Model*). Il *ViewModel* funge da intermediario vitale, esponendo i dati alla *View* in modo reattivo e incapsulando la logica di presentazione.

[5] Nel sistema operativo Android, le *Activity* (i componenti che ospitano le schermate dell'app) vengono periodicamente distrutte e ricreate dal sistema in risposta a determinati eventi o cambi di configurazione, come la rotazione dello schermo. L'utilizzo del *ViewModel* permette di trattenere i dati e mantenere aperti i socket di rete, impedendone la perdita e garantendo la continuità operativa del nodo.

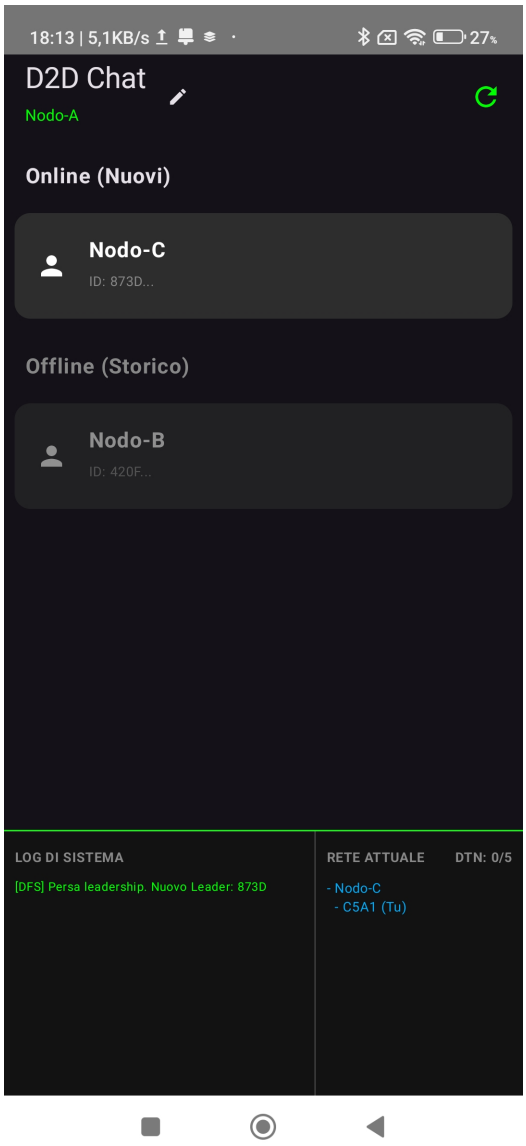

Figura 4.1: Schermata principale dell'applicazione che mostra la classificazione dei nodi scoperti.

4.2.2 Livello di Persistenza (Storage Layer)

Questo livello gestisce l'input/output sul *file system* del dispositivo per garantire la persistenza dello stato tra le diverse sessioni dell'applicazione. Tutte le operazioni di lettura e scrittura avvengono in modo non bloccante. I componenti principali sono:

- `ChatStorage`: Modulo incaricato di serializzare su file (`chat_history.json`) la cronologia dei messaggi. Svolge un ruolo cruciale anche per il protocollo DTN, effettuando il salvataggio periodico della coda dei messaggi in attesa di recapito (`dtn_queue.json`). Questo garantisce che, in caso di crash o chiusura forzata dell'app, il nodo non perda i *bundle* di cui si è fatto custode.

- `PrefsManager`: Modulo interfacciato con le *SharedPreferences*[6] di Android per l'archiviazione di parametri statici. Si occupa di generare e salvare al primo avvio un identificativo logico univoco a 8 byte (derivato da un UUID) che accompagnerà il nodo in modo permanente, il *Display Name* scelto dall'utente e una rubrica locale che associa gli ID logici ai nomi dei *peer* incontrati.

4.2.3 Hardware Wrapper Layer (Livello di Collegamento)

Il modulo `MeshManager` agisce da livello di astrazione hardware, interfacciandosi con le API di Google Nearby Connections. Il suo compito è mascherare la complessità radio ai livelli superiori. Gestisce il ciclo di vita della scansione simultanea sfruttando *Bluetooth Classic*, *Bluetooth Low Energy* (BLE) per l'individuazione e *Wi-Fi Direct* per il trasferimento dati ad alta velocità. Configura esplicitamente il *framework* per l'utilizzo della strategia

[6]Un'API nativa di Android che fornisce un meccanismo leggero per memorizzare e recuperare piccoli insiemi di dati primitivi sotto forma di coppie chiave-valore (*key-value*), ideale per configurazioni e impostazioni persistenti.

`Strategy.P2P_CLUSTER`, l'unica in grado di supportare l'apertura simultanea di *socket* multipli in entrata e in uscita, requisito imprescindibile per la creazione di una topologia *mesh* (M-to-N). Il modulo implementa inoltre i *listener* asincroni (`ConnectionLifecycleCallback` e `PayloadCallback`), accettando automaticamente le connessioni entranti e convertendo bidirezionalmente i flussi di byte grezzi (UTF-8) ricevuti nei *socket* in stringhe JSON formattate per il parser superiore.

4.2.4 Protocol Layer (Livello di Rete e Routing)

Il `MeshProtocolManager` rappresenta il "cervello" della rete. Riceve dal livello inferiore eventi di connessione fisica e messaggi JSON, operando le logiche di instradamento. Per superare i limiti della *MAC randomization* dei sistemi operativi mobili, gestisce la fase di *Handshake* scambiando pacchetti `HELLO`. Mantiene aggiornati dizionari critici in RAM: la `endpointMap` e la `reverseEndpointMap`, che mappano dinamicamente gli ID logici e persistenti dei nodi agli ID effimeri generati per i *socket* fisici. Il modulo orchestra l'elezione del *Leader* tramite un *timeout* di inattività di 20 secondi (impostato manualmente come esempio) e risolve le collisioni confrontando i *timestamp* temporali. Una volta eletto il Root, gestisce in tempo reale lo stato computazionale della *Depth-First Search* (DFS), elaborando e popolando la `downstreamRoutingTable` essenziale per l'instradamento *multi-hop*.

4.3 Implementazione del Routing Ibrido e DTN

Il cuore innovativo dell'applicazione risiede nella funzione di inoltro dei pacchetti applicativi (`MESSAGE`), che discrimina dinamicamente quale paradigma adottare in base allo stato in tempo reale della rete. La logica si divide in due scenari operativi principali:

1. **Routing Diretto (Fase MANET):** Se il destinatario cercato è presente nell'ultimo albero topologico validato (`lastKnownTree`), il pac-

chetto viene instradato lungo i rami in modo deterministico. L'algoritmo verifica prima la raggiungibilità diretta nei vicini fisici (1-hop); in caso negativo, interroga la `downstreamRoutingTable` per inoltrarlo a un nodo figlio (instradamento verso il basso); se anche questa fallisce, il pacchetto viene delegato d'ufficio al nodo padre, `currentDfsParentNodeId` (instradamento verso l'alto).

2. **Routing Opportunistico (Fase DTN):** Se il destinatario risulta inesistente nella topologia attuale o se un *link* fisico si interrompe durante la trasmissione, il nodo corrente assume il ruolo di custode e innesca la variante adattiva dello *Spray and Wait*. Il sistema calcola dinamicamente il limite di copie L in funzione della densità del *cluster* noto al momento della disconnessione, secondo la formula:

$$L = \lfloor \sqrt{N} \rfloor$$

dove N è la cardinalità dell'albero. Il nodo trattiene una copia locale nel *buffer* asincrono (`dtnQueue`) e inietta le restanti $L - 1$ copie a un sottoinsieme casuale di vicini fisici connessi, modificando il *flag* interno del pacchetto in `isDtn = true`.

4.3.1 Gestione della Congestione e Soppressione Passiva

Sviluppare per smartphone commerciali impone limiti severi sull'utilizzo della memoria RAM. Per questo motivo, la coda dei messaggi ritardati (`dtnQueue`) è contingentata da una costante stringente, ad esempio `MAX_DTN_QUEUE_SIZE = 5`. Per evitare il fenomeno del *drop-tail* (scarto definitivo del pacchetto), in caso di saturazione del *buffer* l'algoritmo applica un'**espulsione FIFO delegata**: il messaggio in giacenza da più tempo all'indice 0 viene rimosso dalla memoria locale, ma prima di essere distrutto viene trasmesso forzatamente a un vicino fisico casuale, delegandogli l'onere della conservazione e assicurandone la sopravvivenza nella rete globale.

Al fine di scongiurare *broadcast storms*, è stato implementato un meccanismo di **soppressione passiva**. Piuttosto che intasare il canale con messaggi

di *Acknowledgment* (ACK), ogni volta che un nodo instrada un normale pacchetto dati analizza il suo UUID (identificatore univoco del messaggio). Se l'UUID in transito coincide con quello di un messaggio già presente nella propria `dtnQueue`, il nodo sopprime silenziosamente la propria replica, assumendo che un percorso preferenziale o più rapido sia già stato stabilito da un altro nodo verso la destinazione. Questo approccio di ACK in un certo senso implicito, abbassa drasticamente l'*overhead* del canale radio.

4.4 Interfaccia Utente

Poiché l'applicazione funge da piattaforma di sperimentazione sul campo, l'integrazione di strumenti di visibilità interna è risultata importante. Oltre alle normali *view* di messaggistica istantanea, è stato implementato un pannello di Debug persistente nella parte inferiore dello schermo. Esso fornisce ai ricercatori e agli utenti tecnici un *feedback* visivo istantaneo delle complesse dinamiche *under-the-hood*, esponendo:

- **Console di Sistema:** Un flusso di log continuo a scorrimento che traccia eventi critici, come le disconnessioni fisiche, l'avvio delle elezioni del *Leader*, gli scarti FIFO e gli allarmi di rete.

- **Rappresentazione Seriale DFS:** Una stampa indentata in tempo reale dell'oggetto `TopologyNode`, che permette di verificare visivamente l'architettura logica dell'albero calcolato (identificando immediatamente le relazioni gerarchiche Padre-Figlio-Foglia).

- **Allocazione Buffer DTN:** Un indicatore numerico (es. "DTN: 2/5") che monitora lo stato di saturazione della coda locale, essenziale per valutare i tassi di iniezione e la bontà del meccanismo di re-immissione deterministica quando un nodo *target* torna online.

4.5 Validazione Funzionale del Sistema

Essendo l'applicazione concepita come un *proof-of-concept*, la fase di sperimentazione pratica si è concentrata sulla validazione qualitativa dei singoli moduli dell'architettura e delle transizioni di stato del protocollo, piuttosto che su un'analisi quantitativa delle performance.

Sono stati condotti test mirati su piccola scala (utilizzando un pool ristretto di dispositivi commerciali) per verificare empiricamente le seguenti dinamiche:

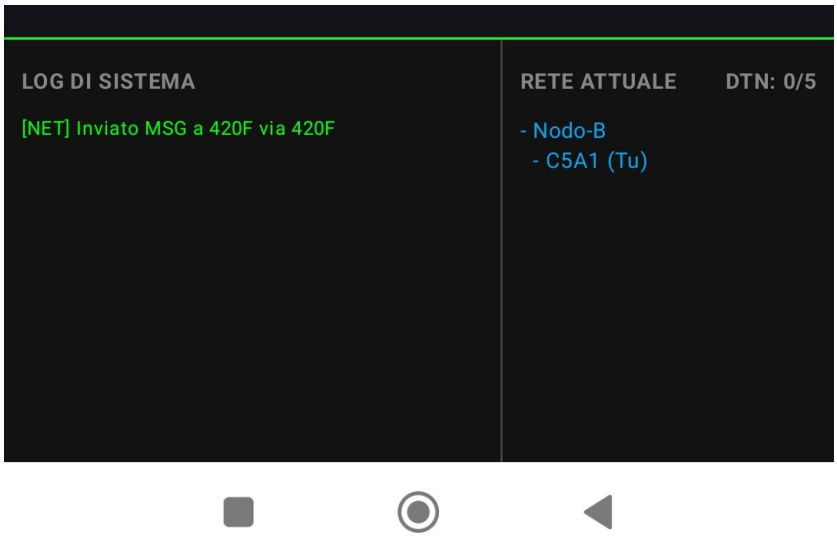

Figura 4.2: Output della console di debug che conferma il corretto instradamento deterministico di un messaggio.

- **Formazione e stabilità della topologia:** Corretta elezione del *Leader* ed elaborazione priva di errori dell'albero DFS in risposta agli eventi di connessione fisica.
- **Transizione ibrida MANET-DTN:** Efficacia del meccanismo di *fallback*, simulando disconnessioni improvvise per innescare correttamente l'assunzione del ruolo di custode dei messaggi e il ricalcolo del limite di copie L (*Spray and Wait*).

Figura 4.3: Dettaglio dei log di sistema durante la caduta di un link e il conseguente innesco della fase opportunistica DTN.

- **Resilienza e soppressione:** Persistenza della coda DTN a seguito di riavvii forzati dell'applicativo e verifica visiva, tramite console di debug, del corretto filtraggio dei messaggi ridondanti (soppressione passiva).

Una validazione empirica sistematica e su larga scala per estrapolare dati statistici complessi esula dagli scopi di questa implementazione preliminare ed è stata esplicitamente demandata agli sviluppi futuri.

Conclusioni

Questo lavoro di tesi ha dimostrato con successo la fattibilità e l'efficacia di una rete *mesh* opportunistica decentralizzata, operante interamente a livello applicativo (*overlay network*). Il risultato principale è aver superato i rigidi vincoli architetturali e di sicurezza imposti dai moderni sistemi operativi mobili commerciali (COTS), abilitando comunicazioni *multi-hop* senza la necessità di privilegi di sistema (*root*) o di hardware radio dedicato. In un panorama in cui le infrastrutture centralizzate si rivelano fragili di fronte a disastri naturali o congestioni estreme, questo approccio "zero-hardware" abbatte le barriere d'ingresso per la creazione di reti di emergenza spontanee.

Attraverso la progettazione e lo sviluppo di un'applicazione Android *proof-of-concept*, il sistema ha dimostrato la capacità di astrarre la complessità dei livelli radio sottostanti. Sfruttando le API di Google Nearby Connections, il protocollo ibrido orchestra in modo trasparente l'uso del Bluetooth per la fase di scoperta a basso consumo e del Wi-Fi Direct per il trasferimento dati. Questa orchestrazione ha permesso di forzare le interfacce native a supportare topologie fisiche non strutturate (molti-a-molti), sulle quali è stata costruita una topologia logica stabile e auto-configurante.

L'implementazione ha convalidato la solidità di un paradigma di instradamento ibrido. Da un lato, l'uso di un algoritmo *Depth-First Search* (DFS) all'interno di *cluster* connessi ha garantito un *routing* deterministico, privo di cicli (*loop-free*) e capace di azzerare i *broadcast storms* tipici degli approcci epidemici. Dall'altro, in presenza di partizioni di rete o alta mobilità, la transizione verso il paradigma *Delay-Tolerant Networking* (DTN) ha assicu-

rato la consegna dei messaggi.Un approccio in un certo senso innovativo di questa fase è consistita nell'adozione di una variante adattiva dello *Spray and Wait*: sfruttando la consapevolezza topologica (*topology awareness*) acquisita proattivamente, il sistema calcola dinamicamente il limite di repliche da iniettare, superando l'inefficienza delle soglie statiche proposte dalla letteratura classica.

In scenari reali, i dispositivi mobili presentano risorse di memoria e batteria rigorosamente contingentate. A tal fine, le strategie di mitigazione della congestione implementate si sono rivelate essenziali per mantenere il sistema operativo e responsivo. L'adozione dell'espulsione FIFO delegata garantisce che un pacchetto non venga mai semplicemente scartato per mancanza di spazio (*drop-tail*), ma venga "salvato" trasferendolo a un nodo vicino. Congiuntamente, il meccanismo di soppressione passiva delle ridondanze ha permesso di bloccare la proliferazione delle copie senza gravare sul canale radio con messaggi di riscontro espliciti (*Implicit ACK*), preservando così la larghezza di banda.

Infine, l'isolamento modulare garantito dall'architettura software a strati assicura che il motore algoritmico del protocollo (*routing core*) possa essere facilmente estratto e riadattato. Non si limita alla sola messaggistica testuale, ma pone le basi per supportare nuove tipologie di servizi *off-grid*, come la condivisione di coordinate GPS o la distribuzione decentralizzata di file critici.

Sebbene i test condotti abbiano validato il corretto funzionamento logico del sistema e la fattibilità dell'architettura proposta, si è consapevoli dell'assenza di una validazione quantitativa in toto che misuri le reali prestazioni del protocollo in codizioni particolari reali. Proprio per colmare questo limite e per la maturazione del progetto, tre direttrici risultano prioritarie tra gli sviluppi futuri:

1. **Sperimentazione su Larga Scala:** Una validazione empirica massiva in contesti urbani, coinvolgendo decine di dispositivi con schemi di mobilità eterogenei, al fine di misurare con precisione metriche chiave

come il *Packet Delivery Ratio* (PDR) e il consumo energetico a lungo termine.

2. **Interoperabilità Cross-Platform:** L'estensione del *wrapper* hardware all'ecosistema iOS. Poiché Google Nearby Connections è multipiattaforma, il porting del motore logico permetterebbe la formazione di reti ibride Android/Apple, un requisito essenziale per massimizzare la pervasività in scenari reali.

3. **Integrazione Hardware Ibrida:** La possibilità di interfacciare i nodi *Leader* con moduli radio a lungo raggio (es. LoRa), creando un'architettura a due livelli in cui la *mesh* su smartphone copre la distribuzione locale e i ponti radio superano distanze chilometriche tra cluster isolati.

Questi passi rappresentano le tappe fondamentali per trasformare questo *proof-of-concept* da un progetto accademico a uno standard di comunicazione resiliente, pronto per l'impiego operativo sul campo. **4.5 Validazione Funzionale del Sistema**

49

Appendice A

Codice Sorgente Significativo

Il presente capitolo in appendice raccoglie i frammenti di codice sorgente più rilevanti ai fini della validazione del protocollo di instradamento ibrido descritto nei capitoli precedenti. I listati presentati si concentrano sulle logiche decisionali (*core routing*) e sugli algoritmi di esplorazione, tralasciando le implementazioni di interfaccia grafica e persistenza dei dati.

A.1 Logica di Instradamento Ibrido e Fallback DTN

La funzione `routeMessage` rappresenta il fulcro dell'architettura ibrida.

- Verifica inizialmente se il destinatario è presente nell'ultimo albero topologico DFS validato.
- Tenta un instradamento deterministico verso il basso (tramite la `downstreamRoutingTable`) o verso l'alto (delegando al nodo Padre).
- Se l'instradamento deterministico fallisce (a causa di una topologia frammentata o di un link interrotto), il sistema intercetta l'errore e innesca in modo trasparente il paradigma *Delay-Tolerant*.

```
private fun routeMessage(msg: MessageData) {
    if (msg.destinationId == myId) {
        onMessageReceived(msg)
        return
    }

    val removed = dtnQueue.removeAll { it.id == msg.id }
    if (removed) onDtnQueueUpdated(dtnQueue.toList())

    val destInTree = lastKnownTree?.findNode(msg.
       destinationId) != null
    var routedSuccessfully = false

    if (destInTree) {
        val nextHop = if (endpointMap.containsKey(msg.
           destinationId)) {
            msg.destinationId
        } else if (downstreamRoutingTable.containsKey(msg.
           destinationId)) {
            downstreamRoutingTable[msg.destinationId]
        } else {
            currentDfsParentNodeId
        }

        if (nextHop != null && endpointMap.containsKey(
           nextHop)) {
            val packet = MeshPacket(
                type = PacketType.MESSAGE, sourceId = myId,
                   senderId = myId, senderName = myName,
                timestamp = System.currentTimeMillis(),
                   messageData = msg.copy(isDtn = false)
            )
            sendToNode(nextHop, packet)
            onLog("[NET] Inviato MSG a ${msg.destinationId.
               take(4)} via ${nextHop.take(4)}")
            routedSuccessfully = true
        }
```

```
31      }
32
33      if (!routedSuccessfully) {
34          onLog("[DTN] Rotta non disponibile per ${msg.
                destinationId.take(4)}. Passo a DTN.")
35          if (msg.senderId == myId && !msg.isDtn) {
36              initiateSprayAndWait(msg)
37          } else {
38              enqueueDtnMessage(msg)
39          }
40      }
41  }
```

Listing A.1: Logica di instradamento e fallback verso DTN

A.2 Protocollo Spray and Wait Adattivo

Questo frammento mostra l'implementazione della variante adattiva dello *Spray and Wait*.

- A differenza degli approcci classici con limite di copie statico, la funzione `initiateSprayAndWait` calcola dinamicamente il limite di spruzzatura L in funzione del numero di nodi N presenti nel cluster noto al momento della disconnessione.
- Il calcolo $L = \lfloor \sqrt{N} \rfloor$ ottimizza il bilanciamento tra probabilità di consegna e saturazione delle risorse di rete.

```
1  private fun initiateSprayAndWait(msg: MessageData) {
2      val n = lastKnownTree?.getAllIds()?.size ?: 1
3      var L = sqrt(n.toDouble()).toInt()
4      if (L < 1) L = 1
5
6      onLog("[DTN] Dest. Offline. Rete N=$n, genero L=$L copie.
           ")
7
```

```
8       enqueueDtnMessage(msg)
9
10      val sprayCount = L - 1
11      if (sprayCount > 0) {
12          val neighborsToSpray = physicalNeighbors.shuffled().
                take(sprayCount)
13
14          neighborsToSpray.forEach { neighborId ->
15              val packet = MeshPacket(
16                  type = PacketType.MESSAGE,
17                  sourceId = myId, senderId = myId, senderName
                        = myName,
18                  timestamp = System.currentTimeMillis(),
19                  messageData = msg.copy(isDtn = true)
20              )
21              sendToNode(neighborId, packet)
22              onLog("[DTN] Copia spruzzata a ${neighborId.take
                    (4)}")
23          }
24      }
25  }
```

Listing A.2: Iniezione adattiva dei pacchetti nella rete DTN

A.3 Gestione del Buffer e Delegazione FIFO

Per mitigare la congestione della memoria sui dispositivi mobili, l'accodamento DTN implementa una strategia di espulsione attiva.

- La funzione `enqueueDtnMessage` verifica se la coda ha raggiunto la capienza massima stabilita.
- In caso di saturazione, invoca `delegateDtnBundle` per deallocare il messaggio più vecchio, trasferendone la responsabilità di custodia a un nodo fisico adiacente scelto casualmente.

```
private fun enqueueDtnMessage(msg: MessageData) {
    if (dtnQueue.any { it.id == msg.id }) return

    if (dtnQueue.size >= MAX_DTN_QUEUE_SIZE) {
        val oldest = dtnQueue.removeAt(0)
        delegateDtnBundle(oldest)
    }

    dtnQueue.add(msg.copy(isDtn = true))
    onDtnQueueUpdated(dtnQueue.toList())
}

private fun delegateDtnBundle(msg: MessageData) {
    val neighbors = physicalNeighbors.toList()
    if (neighbors.isNotEmpty()) {
        val chosenPeer = neighbors.random()
        onLog("[DTN] Buffer Pieno! Delego bundle a ${
            chosenPeer.take(4)}")
        val packet = MeshPacket(
            type = PacketType.MESSAGE, sourceId = myId,
                senderId = myId, senderName = myName,
            timestamp = System.currentTimeMillis(),
                messageData = msg.copy(isDtn = true)
        )
        sendToNode(chosenPeer, packet)
    }
}
```

Listing A.3: Accodamento e delegazione in caso di buffer saturo

A.4 Costruzione dell'Albero DFS (Fase Proattiva)

La funzione `processDfsToken` elabora i *Token* di stato globale generati dal nodo coordinatore (Leader) per mappare in modo asincrono la topologia

fisica.

- L'algoritmo distribuisce la conoscenza della topologia iterando sui nodi adiacenti non ancora visitati (fase di espansione in profondità).

- Se non vi sono più nodi disponibili, marca il sotto-albero come `READY` e innesca il meccanismo di retrocessione (*backtracking*) restituendo il Token al nodo Padre, permettendogli di compilare le tabelle di routing.

```
1  private fun processDfsToken(packet: MeshPacket) {
2      val tree = packet.rootTopology ?: return
3      val sender = packet.senderId
4
5      updateUiPeers(tree)
6
7      val myNodeInTree = tree.findNode(myId) ?: return
8      if (myNodeInTree.isReady) { if (packet.sourceId == myId)
           { } else return }
9
10     val isSenderMyChild = myNodeInTree.children.any { it.id
           == sender }
11     if (!isSenderMyChild && sender != myId)
           currentDfsParentNodeId = sender
12
13     if (isSenderMyChild) {
14         tree.findNode(sender)?.getAllIds()?.forEach {
               downstreamRoutingTable[it] = sender }
15     }
16
17     val visitedIds = tree.getAllIds()
18     val unvisitedNeighbors = physicalNeighbors.filter { !
           visitedIds.contains(it) }
19
20     if (unvisitedNeighbors.isNotEmpty()) {
21         val nextChildId = unvisitedNeighbors[0]
22         myNodeInTree.children.add(TopologyNode(id =
               nextChildId, parentId = myId))
```

```
23          val newPacket = packet.copy(senderId = myId,
                rootTopology = tree)
24          sendToNode(nextChildId, newPacket)
25      } else {
26          myNodeInTree.isReady = true
27          if (currentDfsParentNodeId != null) {
28              val newPacket = packet.copy(senderId = myId,
                    rootTopology = tree)
29              sendToNode(currentDfsParentNodeId!!, newPacket)
30          } else if (packet.sourceId == myId) {
31              onBossElected(true)
32              onLog("[DFS] Elezione completata. Sono il LEADER.
                    ")
33              scope.launch { delay(15000); startNewDfsRound(
                    true) }
34          }
35      }
36
37      checkDtnDelivery()
38  }
```

Listing A.4: Elaborazione del Token e costruzione iterativa dell'albero DFS

Repository GitHub del Progetto

Tutto il codice sorgente sviluppato per questo lavoro di tesi è liberamente consultabile.

`https://github.com/GiuseppeBondii/DTN_chat_GNC`

Bibliografia

[1] Virgil Del Duca Almeida, André B. Oliveira, Daniel F. Macedo, and José Marcos S. Nogueira. Performance evaluation of manet and dtn routing protocols. In *Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, 2011.

[2] Kwan-Wu Chin, John Judge, Aidan Williams, and Roger Kermode. Implementation experience with manet routing protocols. *ACM SIGCOMM Computer Communication Review*, 32(5), 2002.

[3] Kevin Fall and Stephen Farrell. DTN: An architectural retrospective. *IEEE Journal on Selected Areas in Communications*, 26(5):828–837, 2008.

[4] João Gonçalves Filho, Ahmed Patel, Bruno Lopes Alcantara Batista, and Joaquim Celestino Júnior. A systematic technical survey of dtn and vdtn routing protocols. *Computer Standards & Interfaces*, 48:139–159, 2016.

[5] Google Developers. Nearby connections api overview, 2024.

[6] Alex Hinds, Michael Ngulube, Shaoying Zhu, and Hussain Al-Aqrabi. A review of routing protocols for mobile ad-hoc networks (manet). *International Journal of Information and Education Technology*, 3(1):1–5, 2013.

[7] Karim Khamaisi, Oliver Kamer, Bruno Rodrigues, Jan von der Assen, and Burkhard Stiller. Bridging technical capability and user accessi-

bility: Off-grid civilian emergency communication. In *Proceedings of the 15th International Conference on the Internet of Things (IOT '25)*. ACM, 2025.

[8] Jörg Ott, Dirk Kutscher, and Christoph Dwertmann. Integrating dtn and manet routing. In *Proceedings of the 2006 SIGCOMM workshop on Challenged networks (CHANTS '06)*. ACM, 2006.

[9] Sachin Patil, Vahida Attar, Eesha Kurode, and Naishadh Vora. Manet routing protocols with emphasis on zone routing protocol an overview. In *2021 IEEE Region 10 Symposium (TENSYMP)*. IEEE, 2021.

[10] Somaye Pirzadi, Mohammad Ali Pourmina, and Seyed Mostafa Safavi-Hemami. A novel routing method in hybrid dtn-manet networks in the critical situations. *Computing*, 104:2137–2156, 2022.

[11] Luca Sciullo, Angelo Trotta, and Marco Di Felice. Design and performance evaluation of a lora-based mobile emergency management system (locate). *Ad Hoc Networks*, 96:101993, 2020.

Ringraziamenti

Ringrazio tutte le persone che mi sono state vicine durante questo percorso e l'hanno reso possibile.

Ringrazio il prof. Montori e il prof. Trotta che mi hanno dato la possibilità di concluderlo in questo modo.