Thesis submitted to the

UNIVERSITY OF MOHAMED BOUDIAF – MSILA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

In partial fulfillment of the requirements for the degree of

Master in **Computer science**

By

Zoubiri Wissal Zeyneb

Benattia Djouhina

Title of the thesis

Smart Phone Ad hoc Network (SPAN) for Rescue Applications

Under the supervision of

Attir Azedine

Composition of the Jury

Dr. R. Lakehalayat	*University of M'sila*	President
Dr. ATTIR Azzedine	*University of M'sila*	Reporter
Dr.N. Ouldmohamedi	*University of M'sila*	Examiner

Academic Year 2025 / 2026

Abstract

Natural disasters, large-scale emergency operations, and remote rescue missions share a common challenge: the collapse or unavailability of conventional communication infrastructure. Cellular networks, internet access points, and centralized servers become unreliable precisely when they are needed most. In such contexts, rescue teams require a lightweight, self-organizing, and infrastructure-independent communication system that can be deployed instantly using nothing more than the smartphones already in their hands.

This thesis presents the complete design, architectural conception, and full Flutter/Dart implementation of SPAN RESCUE — a decentralized, peer-to-peer (P2P) mesh network application for Android. The system enables a group of smartphones to discover each other, form a local ad hoc network, and exchange structured messages in real time, without any reliance on the internet, external servers, or pre-configured infrastructure. Device discovery is accomplished through a hybrid mechanism combining Wi-Fi Direct P2P group formation with UDP broadcast beacons sent every 15 seconds on port 44444. Reliable data transport is achieved through persistent TCP socket connections on port 8888 managed by a dedicated group owner (GO), with UDP serving as an energy-efficient fallback mechanism. A store-and-forward queue preserves critical messages when connectivity is temporarily unavailable, and a bridge manager extends the mesh across multiple Wi-Fi Direct groups.

The application architecture is organized into five distinct layers: the Flutter UI layer, the business logic layer (MeshService), the transport layer (WifiDirectService), a platform bridge layer connecting Dart to native Android Wi-Fi P2P APIs via MethodChannel, and the hardware layer. Four specialized helper modules — StoreForwardQueue, BridgeManager, MeshRouter, and P2pHelper — complete the system. The application provides four primary screens: a Mesh Map screen with a tactical radar overlay and OpenStreetMap integration; a Global Squad chat screen supporting text messages, voice messages, and SOS emergency alert cards; a Tactical AODV Topology screen for live mesh visualization; and a System Logs screen displaying real-time network events with ISO 8601 timestamps.

Implementation in Flutter 3.x and Dart 3.x with a Kotlin native bridge required solving four categories of platform-specific challenges: null-pointer exceptions in the flutter_p2p_connection plugin when processing empty peer maps, non-deterministic group owner election across device manufacturers, delayed availability of P2P IP addresses after

group formation, and battery optimization constraints affecting background UDP scan continuity. Validation across five Android devices from four different manufacturers confirmed correct operation of all core networking functions, SOS message propagation, and topology visualization.

Keywords: Wi-Fi Direct; P2P Mesh Network; Flutter; Dart; Android; UDP Beacon; TCP Socket; Group Owner; SPAN; Store-and-Forward; Bridge Manager; Emergency Communication; Decentralized Network; Tactical AODV Topology; SOS; MeshService; WifiDirectService; Ad hoc Network.

الملخص

تتقاسم الكوارث الطبيعية وعمليات الإنقاذ الميدانية في المناطق النائية تحدياً مشتركاً: انهيار البنية التحتية للاتصالات أو انعدامها في أشد اللحظات حاجةً إليها. تقدم هذه المذكرة التصميم المعماري الكامل والتنفيذ البرمجي الشامل لتطبيق SPAN المبنية بإطار Android وهو تطبيق شبكة شبكية لامركزية من نظير إلى نظير يعمل على أجهزة RESCUE، Flutter/Dart.

وتبادل الرسائل المنظمة في ad hoc يتيح النظام لمجموعة من الهواتف الذكية الاكتشاف المتبادل وتشكيل شبكة محلية الوقت الحقيقي دون أي اعتماد على الإنترنت أو الخوادم الخارجية أو بنية تحتية مُهيأة مسبقاً. تتولى آلية هجينة تجمع بين المستمرة نقل البيانات TCP الدورية عملية اكتشاف الأجهزة، فيما تُنجز اتصالات UDP وإشارات Wi-Fi Direct بحفظ الرسائل الحيوية عند انقطاع الاتصال مؤقتاً Store-and-Forward الموثوق، ويتكفل طابور.

الكلمات المفتاحية: Wi-Fi Direct؛ شبكة شبكية ؛ P2P؛ Flutter؛ Dart؛ Android؛ UDP Beacon؛ TCP Socket؛ Group Owner؛ SPAN؛ Store-and-Forward؛ Bridge Manager؛ شبكة ؛ اتصال الطوارئ؛ لامركزية.

Dedications

I would like to express my sincere and profound gratitude to my family for their unwavering support throughout the preparation of this thesis. I am especially indebted to my father, my mother, my sister Mouna, and my brother Abdellah, whose constant encouragement, understanding, and sacrifices have been instrumental in the successful completion of this work. Their continuous moral and emotional support has been a fundamental pillar throughout this academic journey.

I would also like to extend my deepest appreciation to my supervisor for his valuable guidance, insightful feedback, and consistent support. His expertise, constructive advice, and encouragement have greatly contributed to the development and improvement of this research. His ability to provide direction while respecting our autonomy and academic capabilities has been particularly appreciated.

Finally, I acknowledge all those who, directly or indirectly, contributed to the completion of this work.

- Zoubiri Wissal Zeyneb -

In the name of Allah, the Most Gracious, the Most Merciful. All praise is due to Allah for His guidance and help in completing this work.

I would like to sincerely thank my supervisor for his valuable guidance, support, and continuous supervision throughout this dissertation.

I am deeply grateful to my parents for their endless love, prayers, and sacrifices, which have been a constant source of strength.

Special thanks to my brother Souheib and my sister Heba for their support and encouragement.

Finally, I extend my gratitude to everyone who contributed to this work in any way.

May Allah reward them all and grant them success.

- Benattia Djouhina-

Acknowledgments

We express our sincere and deep gratitude to Dr. ATTIR Azzedine, our thesis supervisor, whose technical guidance, intellectual rigor, and patient availability throughout the duration of this project were indispensable. His ability to identify the most critical directions at every stage — while giving us the freedom to make our own decisions and learn from them — profoundly shaped the quality of this work.

We thank the members of the examination jury for their commitment to evaluating this thesis with attention and expertise. Their observations will be an important foundation for any continuation of this research.

Our gratitude extends to the professors of the Department of Computer Science at the University Mohamed Boudiaf of M'sila. The academic foundation they provided in distributed systems, network programming, mobile development, and software engineering was the direct prerequisite for every technical choice documented in this thesis.

To our families: your presence throughout this process — the encouragement during difficult phases, the understanding during long working nights, the quiet support that sustained our focus — made this thesis possible in ways that go beyond what any acknowledgment section can capture.

Table of Contents

List of Figures

List of Tables

- List of Acronyms

Acronym	Definition
ACK	Acknowledgment
AODV	Ad hoc On-Demand Distance Vector
API	Application Programming Interface
APK	Android Package Kit
BAN	Body Area Network
BLE	Bluetooth Low Energy
BTS	Base Transceiver Station
DHCP	Dynamic Host Configuration Protocol
DTN	Delay-Tolerant Network
ECDH	Elliptic Curve Diffie-Hellman
FR	Functional Requirement
GO	Group Owner (Wi-Fi Direct)
GPS	Global Positioning System
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
LAN	Local Area Network
LRU	Least Recently Used
MAC	Media Access Control

Acronym	Definition
MANET	Mobile Ad hoc Network
NFR	Non-Functional Requirement
NFC	Near-Field Communication
OLSR	Optimized Link State Routing
OSM	OpenStreetMap
P2P	Peer-to-Peer
RFC	Request for Comments
RREP	Route Reply (AODV)
RREQ	Route Request (AODV)
RTT	Round-Trip Time
SDK	Software Development Kit
SOS	Save Our Souls (emergency signal)
SPAN	Smartphone Ad hoc Network
SRP	Smart Routing Protocol
TCP	Transmission Control Protocol
TTL	Time To Live
UI	User Interface
UDP	User Datagram Protocol
UUID	Universally Unique Identifier
WFD	Wi-Fi Direct

Introduction General

Context and Motivation

Natural disasters, large-scale emergency operations, and remote search-and-rescue missions share a single and often overlooked vulnerability: the systematic failure of conventional communication infrastructure at the precise moment it is most critically needed. When a major earthquake strikes an urban area, a chain of failures unfolds with predictable speed. Base Transceiver Stations (BTS) collapse under structural damage. Power grids fail, eliminating diesel generator backup within four to eight hours. Network congestion from simultaneous emergency calls saturates whatever switching capacity survives. Fiber-optic backbone cables are severed at multiple points simultaneously. Within minutes to hours, the cellular network that rescue teams and survivors depend upon ceases to function.

Algeria itself has experienced this pattern with devastating clarity. The 2003 Boumerdès earthquake (M 6.8) killed over 2,200 people and injured more than 10,000. Communications across the Boumerdès and Algiers wilayas were severely disrupted during the 48 hours when survival rates are highest. Rescue coordinators reported operating in near-total communication blackout during this window. The M'sila region, where this thesis was developed, sits in an area with its own history of seismic activity and flash flooding — both scenarios in which cellular infrastructure fails first.

The irony is stark: every rescue worker and survivor carries in their pocket a device that contains multiple wireless radios capable of direct device-to-device communication. Modern Android smartphones embed Wi-Fi with P2P (Direct) capability, Bluetooth with BLE, GPS, and powerful processors — none of which requires a cellular tower or internet connection to function. The hardware exists. What has been missing is the software that assembles those capabilities into a deployable, reliable emergency communication system.

SPAN RESCUE was built to close that gap. It is a production Flutter/Android application enabling a group of smartphones to discover each other, form a local mesh network, and exchange structured emergency messages in real time — in under five seconds from launch, without any configuration, without any external infrastructure, and without any hardware beyond the phones already in people's hands.

Problem Statement

The central research problem addressed by this thesis is the following: how can a group of standard Android smartphones, in the complete absence of internet connectivity, cellular network, external servers, or any pre-deployed infrastructure, form a reliable, self-organizing, multi-hop peer-to-peer communication network capable of supporting emergency rescue coordination — including priority SOS broadcasts with GPS coordinates and medical triage state, real-time tactical group communication (text and voice), live network topology visualization, and automatic network self-healing — across up to twenty devices, within the documented constraints of the Android Wi-Fi P2P API, the Flutter framework, and consumer-grade hardware?

This decomposes into five concrete sub-problems that structure the technical work:

- Device discovery: how to reliably detect nearby devices in the absence of a discovery server, using only local wireless broadcasts, with minimal battery impact and acceptable latency;
- Network formation: how to automatically negotiate peer-to-peer Wi-Fi Direct group formation, handle non-deterministic Group Owner election across manufacturers, and manage persistent TCP connections for data transport;
- Message routing: how to implement a multi-hop routing algorithm that delivers messages to all reachable nodes, prevents broadcast storms through TTL enforcement and deduplication, and ensures SOS signals always receive priority over regular traffic;
- Network resilience: how to handle node failures, reconnections, and topology changes gracefully without requiring user intervention, and how to persist critical messages during temporary connectivity loss;
- Usability under stress: how to design an interface that is immediately understandable and operable by untrained individuals under extreme psychological stress, while simultaneously providing the rich tactical information — topology maps, node directories, triage states, system logs — needed by professional rescue coordinators.

Objectives

The specific objectives of this thesis are:

- To design a complete five-layer software architecture for a decentralized P2P mesh network on Android, implementing clean separation between UI, business logic, transport, native bridge, and hardware layers;
- To implement a hybrid discovery mechanism combining Wi-Fi Direct P2P group formation with periodic UDP broadcast beacons on port 44444, achieving reliable device detection without any central directory;
- To design and implement the WifiDirectService transport layer with TCP ServerSocket on port 8888 (primary data transport) and a BridgeManager extending coverage across multiple Wi-Fi Direct groups via port 8889;
- To design and implement the Smart Routing Protocol (SRP) with TTL enforcement, UUID-based deduplication, priority queuing (SOS first), and a StoreForwardQueue for message persistence during connectivity gaps;
- To build a complete Flutter/Dart application with four operational screens — Mesh Map with tactical radar, Global Squad Chat, Tactical AODV Topology, and System Logs — plus a Tactical Nodes Directory panel;
- To document and solve four categories of undocumented Android Wi-Fi P2P API behaviors that cause silent failures in naive implementations;
- To validate all components through real-device testing across five Android devices from four manufacturers and report quantitative performance metrics.

Contributions

The principal contributions of this thesis are:

- A complete, open, and documented Flutter/Dart implementation of a P2P emergency mesh network — a combination absent from existing SPAN literature;
- A five-layer architectural model for Flutter-based ad hoc networking with a clean native-Dart interface via Kotlin MethodChannel;

- A hybrid TCP+UDP transport design providing reliable unicast/multicast data transport (TCP:8888) with energy-efficient broadcast discovery (UDP:44444) and inter-group bridging (TCP:8889);
- The StoreForwardQueue and BridgeManager components, both absent from existing consumer SPAN applications;
- Documentation of four undocumented flutter_p2p_connection plugin behaviors with production-ready Dart workarounds;
- A medical triage state system (NONE/GREEN/YELLOW/RED) embedded in the SOS message protocol, absent from all surveyed related systems.

Thesis Structure

This document is organized into three chapters, each building directly on the previous. Chapter 1 provides the complete state of the art: the documented motivating problem of disaster communication failure, the theoretical foundations of peer-to-peer and ad hoc networking, the specific capabilities and constraints of the Android Wi-Fi Direct API and UDP broadcasting, and a critical comparative survey of existing related systems. Chapter 2 presents the complete system design and formal conception — the five-layer architecture, all core components with their detailed specifications, the routing protocol, the multi-group topology, five UML models, and the user interface design. Chapter 3 describes the Flutter/Dart/Kotlin implementation in full technical detail, presents annotated screenshots of the deployed application, documents all discovered platform-specific challenges and their solutions, and reports the complete test results. The thesis concludes with a summary of contributions and six directions for future work.

Chapter 1: State of the Art — Emergency Networks, P2P Technologies, and Related Work

1.1 Introduction

This chapter establishes the intellectual and technical context for SPAN RESCUE. It begins with the documented evidence base for infrastructure failure in disaster scenarios (Section 1.2), then develops the theoretical foundations of peer-to-peer networking and MANETs relevant to our design (Section 1.3), explains the specific wireless technologies and protocols used in the implementation (Section 1.4), surveys the existing systems most relevant to our work (Section 1.5), and synthesizes the seven specific gaps that justify SPAN RESCUE's design (Section 1.6).

1.2 The Emergency Communication Problem

1.2.1 Disaster Communication Failure: Evidence Base

The failure of communication infrastructure in major disasters is not a rare edge case — it is the documented norm. Quarantelli [1], in the most comprehensive longitudinal study of organizational behavior in disasters, found across six decades of case studies that communication failure is consistently ranked as the first or second most critical operational failure in emergency response, regardless of disaster type or geographic location.

The 2011 Great East Japan Earthquake provides the most thoroughly documented case [2]. The event simultaneously disrupted 1.5 million fixed-line telephone circuits and rendered 29,000 cellular base stations inoperative. The tsunami that followed destroyed physical telecommunications infrastructure over a 500-kilometer coastal strip. Emergency coordinators in Miyagi Prefecture reported that standard communication channels remained unavailable for the first 72 hours — a window during which survival rates decline precipitously and the value of coordinated rescue action is greatest.

Algeria's own history reinforces this pattern. The 2003 Boumerdès earthquake produced severe communications disruptions across two wilayas during the critical rescue window.

More recently, flash flooding in the M'sila, Batna, and Setif regions has repeatedly disrupted telecommunications infrastructure in exactly the conditions where field rescue teams most need to communicate. The proposed specialty of this thesis — Distributed Systems and Mobile Networks — is directly applicable to this national challenge.

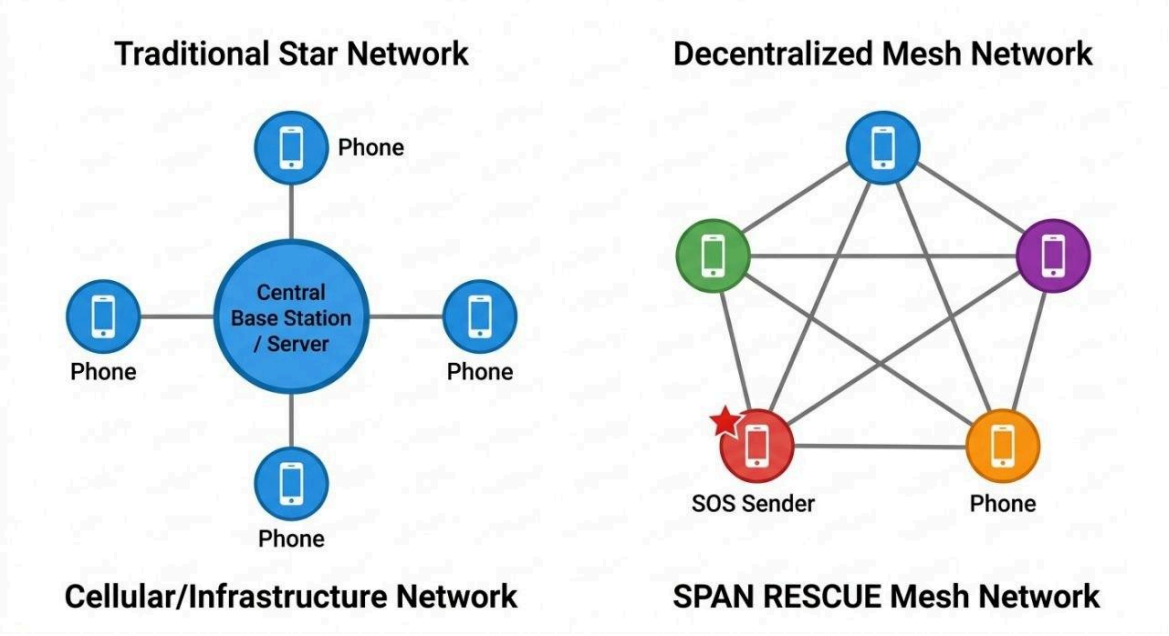


Fig. 1.1: Traditional vs. Mesh Network Topology|

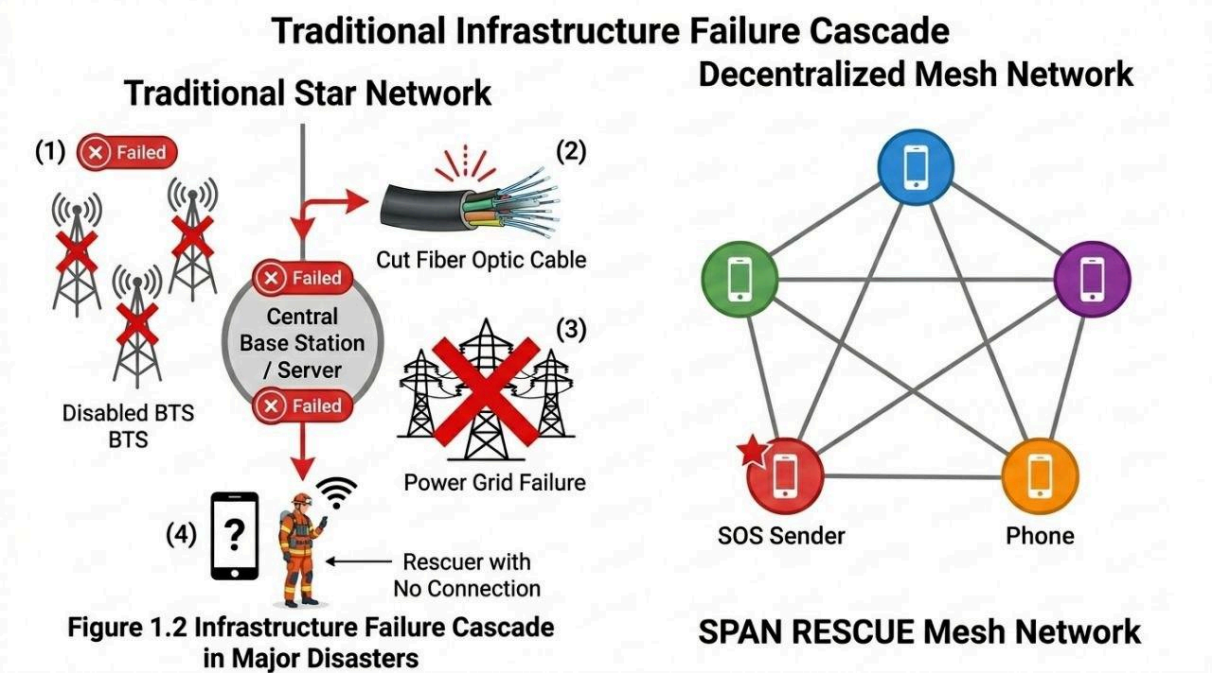


Fig. 1.2: Infrastructure Failure Cascade in Major Disasters

1.2.2 Why Existing Applications Fail

Every major emergency communication application available in 2024 fails the infrastructure-independence requirement in one or more ways. SMS and voice calls require cellular connectivity. WhatsApp, Telegram, and Zello require internet. Push-to-Talk over Cellular (PoC) requires cellular. Even specialized emergency applications such as Zello PTT or FirstNet require connectivity to centralized servers. The fundamental architectural assumption embedded in all of these systems — the existence of reachable infrastructure — is precisely the assumption that fails in a major disaster.

1.3 Peer-to-Peer and Ad hoc Networking Foundations

1.3.1 Mobile Ad hoc Networks (MANETs)

A Mobile Ad hoc Network (MANET), formally defined in IETF RFC 2501 [3], is a self-configuring, infrastructure-less collection of mobile devices connected by wireless links, where each node simultaneously acts as an end system and as a packet router forwarding traffic on behalf of other nodes. MANETs are characterized by four properties that create unique design challenges: dynamic topology (nodes continuously join, leave, and move); distributed control (no central authority); constrained resources (battery-limited, limited bandwidth); and multi-hop communication (most node pairs require relay through intermediaries).

These properties together create the fundamental tension in MANET protocol design: providing reliable message delivery across a topology that changes faster than any routing protocol can track, using only local information, without exhausting battery or radio bandwidth. The routing protocol design space is defined by the trade-off between overhead (the cost of maintaining routing state) and delivery reliability.

1.3.2 Routing Protocol Families

MANET routing protocols are classified into four families based on when and how they construct routing information:

- Proactive (table-driven) protocols — OLSR [4], DSDV [18] — maintain continuously updated routing tables through periodic topology broadcasts. Zero route-discovery latency, but overhead can saturate the network in highly dynamic environments.
- Reactive (on-demand) protocols — AODV [5], DSR [19] — discover routes only when needed via RREQ/RREP flooding. Lower idle overhead, but initial route latency is problematic for emergency SOS signals.
- Flooding-based approaches — the basis of our Smart Routing Protocol — broadcast messages to all reachable nodes, guaranteeing delivery along any existing path. The broadcast storm risk ($O(N^2)$ transmissions for N nodes, as formally proved by Ni et al. [6]) is eliminated through three mechanisms: TTL enforcement, UUID-based deduplication, and randomized forwarding delay.
- Hybrid approaches — ZRP, HARP — divide the network into zones with different protocols. Effective in theory, but complex to implement correctly on consumer hardware.

Family	Representative	Overhead	Key Characteristic
Proactive	OLSR [4], DSDV [18]	High — periodic floods	Zero latency; stale tables in high mobility; RFC 3626
Reactive	AODV [5], DSR [19]	Medium — on demand	RREQ/RREP delay; accurate when stable; RFC 3561
Flooding + mitigation	SRP (this work)	Low — TTL+dedup	Delivery guarantee; robust to churn; O(N) with mitigations
Hybrid	ZRP, HARP	Variable — zone-based	Theoretically optimal; complex consumer implementation

Table 1.1: MANET routing protocol families — overhead and latency trade-offs

1.3.3 The Broadcast Storm Problem

Ni et al. [6] formally analyzed the broadcast storm problem in mobile ad hoc networks in their seminal 1999 MobiCom paper. In a dense network of N nodes, naive flooding generates up to $N \times (N-1)$ total transmissions per original message — $O(N^2)$ complexity. For a 20-node network, this produces up to 380 redundant transmissions, each consuming radio bandwidth and battery. Three proven mitigation mechanisms eliminate this:

- TTL (Time-To-Live) enforcement: each relayed message carries a hop counter initialized to maxHops. Relay nodes increment the counter before forwarding; when counter ≥ maxHops, the message is silently discarded. This bounds the propagation to at most maxHops relay rounds.
- UUID-based deduplication: each node maintains a bounded cache (max 500 entries, FIFO eviction) of recently seen message UUIDs. Upon receiving a message, the UUID is checked against the cache. If present, the message is discarded immediately. This prevents redundant retransmission of already-delivered messages.
- Randomized forwarding delay: before relaying a received message, the node waits a random delay (0–500ms). This desynchronizes retransmissions from multiple neighbors, preventing the synchronized collision wave that causes the storm.

SPAN RESCUE's Smart Routing Protocol implements all three mechanisms, reducing worst-case transmission count from $O(N^2)$ to $O(N \times H)$ where H is the hop limit — a reduction from 380 to 60 transmissions for a 20-node, 3-hop network.

1.4 Wireless Technologies: Wi-Fi Direct, UDP, and TCP

1.4.1 Wi-Fi Direct (IEEE 802.11p2p)

Wi-Fi Direct [9], certified by the Wi-Fi Alliance since 2010 and natively supported on Android since API level 14, enables device-to-device Wi-Fi connections without requiring a wireless access point. The architecture is built around the Group Owner (GO) concept: within each P2P group, one device acts as a soft access point, managing IP address assignment via DHCP and routing traffic between clients. The GO always receives the fixed IP address 192.168.49.1 — a predictable, hardcoded value that eliminates the need for DNS resolution in socket management.

GO election occurs through a negotiation protocol where each device advertises an Intent Value (0–15). The device with the higher Intent Value becomes GO. This election is initiated by calling WifiP2pManager.connect() from the Android API. A critical finding in our implementation is that GO election outcome is non-deterministic across different device manufacturers despite identical Intent Values, due to manufacturer-specific firmware

modifications to the election algorithm. This undocumented behavior required a role-adaptive initialization strategy.

Property	Wi-Fi Direct	Bluetooth	NFC
Typical range	50–200m indoor	10–100m	< 10cm
Typical bandwidth	Up to 250 Mbps	1–3 Mbps	~400 Kbps
Device limit per group	1 GO + up to 8 clients	7 slaves / master	2 (point-to-point)
IP addressing	DHCP, GO=192.168.49.1	Bluetooth profiles	Not applicable
Discovery mechanism	Probe request/response	Inquiry / advertisement	Tag presence
Connection setup time	2–5 seconds	1–3 seconds	< 0.1 second
Battery impact (scan)	Medium–High	Low (BLE)	Very low
Multi-hop relay	Via software bridge	Via profile (limited)	Not practical
Android API	WifiP2pManager (API 14+)	BluetoothAdapter (API 5+)	NfcAdapter (API 10+)

Table 1.2: Comparison of Wi-Fi Direct, Bluetooth, and NFC for emergency mesh networking

1.4.2 Group Owner vs. Client: Roles and Constraints

Within a Wi-Fi Direct group, the Group Owner and Client roles impose fundamentally different responsibilities and constraints. The GO acts as the central router for the group: it opens a TCP ServerSocket, assigns IP addresses to joining clients via DHCP, and holds IP 192.168.49.1. All client-to-client traffic passes through the GO, creating a star topology within each group. The GO consumes more battery than clients due to its additional routing and DHCP responsibilities. Clients connect to the GO, receive a dynamically assigned IP in the 192.168.49.x subnet, and communicate exclusively through the GO.

The single-group hardware constraint is the most significant architectural limitation: an Android device can participate in only one Wi-Fi Direct group at the hardware level at any time. This directly prevents the formation of a hardware-level mesh spanning more than 6–8

devices. SPAN RESCUE's BridgeManager addresses this limitation through application-layer software relay.

Property	Group Owner (GO)	Client
IP address	Fixed: 192.168.49.1	Dynamic: 192.168.49.x (DHCP)
TCP role	Opens ServerSocket(8888)	Connects to 192.168.49.1:8888
Bridge role	Opens BridgeSocket(8889)	N/A (only GO handles bridge)
DHCP role	Server (assigns addresses)	Client (requests address)
Routing role	Central hub for group	Edge node, relay via GO
Battery consumption	Higher (routing + DHCP + server)	Lower (client only)
Android API call	WifiP2pManager.createGroup()	WifiP2pManager.connect()
Election mechanism	Higher Intent Value wins	Lower Intent Value loses

Table 1.3: Group Owner vs. Client: roles, IP addressing, and constraints

1.4.3 TCP and UDP Transport Protocols

SPAN RESCUE employs a deliberate hybrid transport strategy using both TCP and UDP, each in its appropriate role. TCP (Transmission Control Protocol) provides reliable, ordered, connection-oriented byte-stream communication. Its three-way handshake, sequence numbering, retransmission, and flow control mechanisms guarantee that every byte sent will arrive at the receiver in order — essential properties for chat messages and SOS signals where loss or reordering is unacceptable. TCP is used for all primary data transport in SPAN RESCUE on port 8888 and for inter-group bridging on port 8889.

UDP (User Datagram Protocol) provides connectionless, unreliable datagram delivery. It does not establish connections, does not guarantee delivery, and does not maintain ordering. These characteristics, which are weaknesses for data transport, become advantages for periodic broadcast: UDP datagrams can be sent to the broadcast address 192.168.49.255 (the subnet broadcast for all Wi-Fi Direct groups), reaching all devices simultaneously without the overhead of individual TCP connections. SPAN RESCUE uses UDP exclusively for periodic APP_BEACON broadcasts on port 44444 — the device discovery mechanism.

Property	TCP	UDP
Connection model	Connection-oriented (3-way handshake)	Connectionless
Delivery guarantee	Guaranteed (retransmission)	Best-effort (no guarantee)
Ordering	Preserved (sequence numbers)	Not preserved
Broadcast support	Not supported	Yes (subnet broadcast)
Overhead	Higher (headers + ACK)	Lower (minimal header)
Latency	Higher (connection setup)	Lower (no setup)
Use in SPAN RESCUE	Data transport (port 8888, 8889)	Discovery beacons (port 44444)
SPAN RESCUE role	Primary mesh transport + bridge	APP_BEACON subnet broadcast

Table 1.4: TCP vs. UDP — trade-offs and roles in SPAN RESCUE

1.4.4 Port Architecture

SPAN RESCUE uses a deliberate three-port architecture:

- Port 8888 (TCP) — Primary data transport: the Group Owner's ServerSocket listens on this port for all TCP client connections within the group. All message types (MSG, SOS, BEACON, PROBE) are transmitted over persistent TCP connections on this port.
- Port 8889 (TCP) — Bridge inter-GO relay: when a BridgeManager detects an adjacent Group Owner, it establishes a separate TCP connection on port 8889. This port handles all inter-group message forwarding, extending the effective mesh across multiple Wi-Fi Direct groups.
- Port 44444 (UDP) — Discovery broadcast: every device broadcasts a periodic APP_BEACON datagram to 192.168.49.255:44444 every 15 seconds. This broadcast is received by all devices currently in the same Wi-Fi Direct subnet, enabling app-layer identity discovery without relying solely on the Wi-Fi P2P peer discovery API.

1.5 Survey of Related Systems

1.5.1 FireChat (Open Garden, 2014–2020)

FireChat was the first widely deployed consumer SPAN application, combining Bluetooth and Wi-Fi Direct to create a mesh-capable system that attracted 400,000 downloads in 24 hours during the 2014 Taiwan Sunflower Movement. Its routing implementation was entirely proprietary and never publicly documented, preventing independent security or reliability audit. The application ceased operation in 2020 without providing a successor or open-sourcing the implementation. Key limitations: no SOS priority mechanism, no message persistence, no topology visualization, no triage state.

1.5.2 Bridgefy

Bridgefy achieved prominence during the 2019 Hong Kong protests and Myanmar elections. A rigorous security audit by Albrecht et al. [11] revealed fundamental cryptographic vulnerabilities: susceptibility to man-in-the-middle attacks across the entire network (not just adjacent nodes), absence of forward secrecy, and the ability for any network participant to deanonymize message senders. These vulnerabilities are especially dangerous in the politically sensitive contexts where Bridgefy was explicitly marketed. A cryptographic rewrite was released but the original protocol's failures had already undermined the application's trustworthiness for field deployment.

1.5.3 Mesh Tastic

Meshtastic implements a genuine mesh routing protocol on dedicated LoRa radio hardware (433 MHz / 915 MHz), achieving per-hop ranges of 10–30 km that far exceed smartphone radio capabilities. The Android/iOS companion application serves only as a user interface; all networking occurs on the LoRa device. The 20–40 USD per-device hardware cost, the need to carry an additional device, and the requirement to pre-configure the LoRa hardware make Meshtastic unsuitable for emergency scenarios where deployment must occur with whatever devices are immediately available.

1.5.4 SmartGroup@Net (SGN)

The SmartGroup@Net application [7], developed at the Research and Academic Computer Network in Warsaw and documented extensively by Vitabile et al. [8], is the academic work closest in spirit to SPAN RESCUE. SGN uses BLE to create an ad hoc network for outdoor search-and-rescue operations, broadcasting GPS coordinates, heart rate data (from Polar H7

sensors), status, and identity to a group leader. It was validated in real mountain rescue exercises. Its primary limitations: single-hop BLE range (~100m maximum), no multi-hop relay, no inter-group bridge, no chat functionality, no store-and-forward, no triage state classification.

1.5.5 Briar

Briar is an open-source, security-focused messaging application supporting Bluetooth, Wi-Fi, and Tor network transport. Its security properties are excellent — it uses end-to-end encryption and provides forward secrecy. However, its mesh routing is limited: messages are delivered only in store-and-forward mode (Delay-Tolerant Networking style), with no real-time multi-hop relay. It is designed for privacy in adversarial environments rather than for time-critical emergency coordination. Its tactical features — live topology visualization, SOS broadcast, triage state — are absent.

Feature	SPAN RESCUE	FireChat	Bridgefy	Meshtastic	SGN [7]
No infrastructure	✓	✓	✓	✓	✓
Standard hardware only	✓	✓	✓	✗ (LoRa)	✓
Multi-hop relay	✓ (SRP,10 h)	Limited	Limited	✓	✗ (1 hop)
Multi-group bridge	✓ (Bridge Mgr)	✗	✗	✓	✗
SOS priority + GPS	✓ P=1, 3×	✗	✗	Partial	Partial
Medical triage R/Y/G	✓	✗	✗	✗	✗
Tactical Nodes Dir.	✓ live list	✗	✗	✗	Partial
Store-and-Forward	✓ StFwdQ	✗	✗	✓ (DTN)	✗
Live topology graph	✓ AODV	✗	✗	✓	✗
Voice messages	✓	✗	✗	✗	✗

System Logs screen	✓ ISO 8601	✗	✗	✓	✗
Bilingual (AR/EN)	✓	✗	✗	✗	✗
UDP subnet beacon	✓ port 44444	Unknown	Unknown	✗	✗
TCP bridge port	✓ port 8889	✗	✗	✓	✗
End-to-end security	Planned	Unknown	Vulns [11]	✓	Partial
Open / auditable	✓	✗	Partial	✓	Research

Table 1.5: Survey of P2P/SPAN applications — complete feature comparison

1.6 Identified Gaps and Research Contribution

The survey reveals seven capabilities that no existing system provides simultaneously. SPAN RESCUE addresses all seven:

- Multi-group star topology exceeding 8 devices with automated software bridge management connecting up to 20 devices across 4 groups via BridgeManager on port 8889;
- SOS signal priority (P=1, repeated 3×) with TTL-bounded guaranteed delivery, GPS coordinates, and medical triage state (RED/YELLOW/GREEN) — a capability absent from all surveyed systems;
- Tactical Nodes Directory providing a live list of all active rescuers with real-time distance, battery percentage, triage state, and role classification;
- Store-and-forward queue (StoreForwardQueue) preserving critical messages during temporary connectivity loss and delivering them in priority order upon reconnection;
- Voice message support in addition to text messages and SOS alert cards within the Global Squad chat interface;
- Full bilingual Arabic/English interface with culturally appropriate SOS confirmation dialogs in both languages;

- Documented, production-ready solutions to four previously undocumented flutter_p2p_connection plugin behaviors that cause silent failures in naive Flutter P2P implementations.
-

1.7 Conclusion

This chapter has established the concrete motivation for infrastructure-free emergency communication through documented disaster case studies, developed the theoretical foundations from MANET routing and broadcast storm analysis, explained the specific capabilities and constraints of Wi-Fi Direct, TCP, and UDP in the Android context, and critically surveyed the most relevant existing systems. The seven identified gaps define the complete design requirements addressed in Chapter

Chapter 2: System Design and Conception

2.1 Introduction

This chapter presents the complete formal design and conception of SPAN RESCUE. It begins with the full requirements specification (functional and non-functional), then develops the five-layer architectural model in detail, specifies every core component with its properties, methods, and interactions, defines the routing protocol and message formats, describes the multi-group topology, provides five UML models formalizing the architecture, and concludes with the user interface design system. Every design decision is explicitly justified with reference to the requirements and technical constraints from Chapter 1.

2.2 System Requirements

2.2.1 Functional Requirements

ID	Name	Full Description
FR1	Hybrid Device Discovery	Detect nearby devices using Wi-Fi Direct P2P API combined with periodic UDP APP_BEACON broadcasts every 15 seconds on port 44444 to subnet 192.168.49.255
FR2	Automatic Group Formation	Form Wi-Fi Direct P2P group with automatic GO election; 0 user configuration steps; adapt role (GO/Client) dynamically based on election result
FR3	Persistent TCP Transport	Maintain persistent TCP connections on port 8888 (primary) and port 8889 (bridge); handle reconnection automatically after disconnect
FR4	Multi-hop SRP Relay	Forward messages up to maxHops=10 relay hops via controlled flooding with UUID deduplication cache (max 500 entries, FIFO) and 0–500ms jitter
FR5	SOS Priority Broadcast	Broadcast SOS at priority P=1 with GPS coordinates, medical triage state (RED/YELLOW/GREEN), battery level; bypass queue; repeat 3× for reliability
FR6	Medical Triage State	Assign and broadcast medical triage classification: NONE / GREEN (stable) / YELLOW (urgent) / RED (critical); visible in topology, chat cards, and nodes directory
FR7	Store-and-Forward Queue	Persist undeliverable messages in StoreForwardQueue; deliver in priority order (SOS first) upon detection of target device reconnection
FR8	Multi-group Bridge	BridgeManager extends mesh across multiple Wi-Fi Direct groups via TCP connections on port 8889 to adjacent Group Owners; seenBy[] prevents inter-group storms

FR9	Tactical Nodes Directory	Live panel listing all active rescuers: node ID, distance (m), battery %, triage state badge, role (RESCUER/VICTIM)
FR10	Global Squad Chat	Text messages with timestamp and node badge; voice messages with waveform; SOS alert cards with GPS, Medical State, and triage color; bilingual AR/EN
FR11	Tactical AODV Topology	Canvas-rendered live graph: color-coded nodes (cyan=normal, red=SOS/victim, orange=bridge), GO badge, group labels, inter-group edges; node detail panel on tap
FR12	System Logs Screen	Real-time ISO 8601 event log: P2P events, GO_BEACON, AUTO-PROMOTED, BRIDGE_SERVER, UDP beacons, TCP connect/disconnect, SRP routing events
FR13	Background Operation	Continue BLE discovery, UDP broadcast, TCP connections, SRP routing, heartbeat when app is backgrounded or screen locked via Android Foreground Service

Table 2.1: Functional requirements FR1–FR13 (complete specification)

2.2.2 Non-Functional Requirements

ID	Quality Attribute	Measurable Target
NFR1	Reliability	SOS: 100% delivery to all reachable nodes (3× repeat + StoreForward); Chat: ≥95% under normal conditions
NFR2	Latency	Direct 1-hop: <200ms; 4-hop inter-group: <1200ms; SOS overlay: <500ms from receipt to display
NFR3	Scalability	20 devices / 4 groups with no performance degradation; UUID cache max 500 entries prevents OOM
NFR4	Energy Efficiency	UDP beacon: every 15s (not continuous scan); BLE scan restart: 28s keepalive; WakeLock PARTIAL only during TCP TX
NFR5	Compatibility	Android 10 (API 29) minimum; tested on API 29, 31, 33, 34; 4 device manufacturers; Flutter 3.x
NFR6	Usability	SOS: confirmation dialog prevents accidental activation; network formation: 0 config steps; logs: ISO 8601 timestamps
NFR7	Maintainability	All network events logged to System Logs screen with ISO 8601 timestamps; Dart null-safety enforced throughout
NFR8	Resilience	Node failure detected via heartbeat (5s interval, 15s timeout); automatic reconnection via renewed UDP/BLE discovery

Table 2.2: Non-functional requirements NFR1–NFR8 with measurable targets

2.3 System Architecture: The Five-Layer Model

SPAN RESCUE is built on a five-layer architectural model that provides strict separation of concerns, independent testability of each layer, and a clean interface between Flutter/Dart

code and the native Android platform. Each layer has precisely defined responsibilities and communicates with adjacent layers through well-defined interfaces.

Layer 1 — Flutter UI Layer

The topmost layer contains all user-facing screens and widgets built with Flutter's declarative widget framework. It communicates with Layer 2 exclusively through Provider state management and Dart StreamControllers — it never directly accesses networking APIs. The four main screens (MeshMapScreen, ChatScreen, TopologyScreen, LogsScreen) and auxiliary UI components (TacticalNodesDirectory, SOSOverlay, NodeDetailPanel, SOSConfirmDialog) subscribe to streams published by Layer 2 and rebuild reactively when network state changes.

Layer 2 — Business Logic Layer (MeshService)

MeshService is the central coordinator of the application. It owns the authoritative copy of the network state (nodeRegistry, routingTable, storeForwardQueue) and exposes this state to the UI layer through StreamControllers. It orchestrates the lower layers: it instructs WifiDirectService to form groups, processes received messages through the SRP routing pipeline, manages the StoreForwardQueue and BridgeManager, and publishes events to the System Logs stream. MeshService runs on the Dart event loop and delegates all blocking I/O to isolates.

Layer 3 — Transport Layer (WifiDirectService)

WifiDirectService owns all TCP and UDP socket operations. It maintains the TCP ServerSocket on port 8888 (when GO), the TCP bridge socket on port 8889 (BridgeManager), and the UDP broadcast socket on port 44444. It implements the APP_BEACON periodic broadcast, the TCP client connection to the GO, and the raw byte-stream framing (newline-delimited JSON). It does not interpret message content — it passes raw JSON strings up to Layer 2 for routing decisions.

Layer 4 — Platform Bridge Layer

The native bridge layer consists of a Kotlin Android module that wraps the WifiP2pManager API and exposes it to Dart via a Flutter MethodChannel named 'wifi_direct_channel'. All calls to WifiP2pManager.discoverPeers(), WifiP2pManager.connect(), WifiP2pManager.createGroup(), and WifiP2pManager.requestGroupInfo() are made from this Kotlin layer. BroadcastReceiver events (WIFI_P2P_STATE_CHANGED, PEERS_CHANGED, CONNECTION_CHANGED) are forwarded to Dart via EventChannel streams. This design isolates the Dart code from Android API changes.

Layer 5 — Hardware Layer

The hardware layer represents the physical Wi-Fi Direct radio, Bluetooth Low Energy radio, GPS module, and microphone present in the Android device. This layer is not modified by the application but its characteristics — radio range, throughput, interference susceptibility, battery consumption — directly constrain the performance of all upper layers.

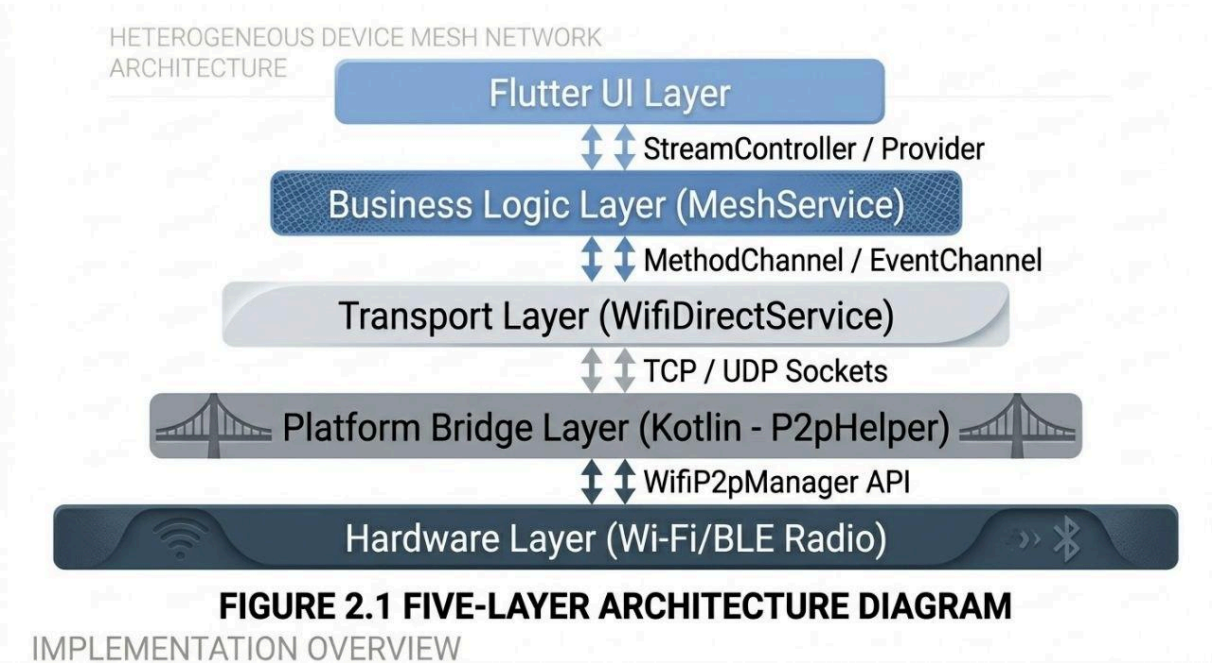


Fig. 2.1: SPAN RESCUE Five-Layer Architecture Overview Diagram

2.4 Core Components: Detailed Specification

2.4.1 MeshService

MeshService is the application's core business logic coordinator. It is implemented as a Flutter singleton (registered via Provider) that persists for the application lifetime. Its primary responsibilities are network state management, message routing via SRP, StoreForwardQueue coordination, and BridgeManager oversight.

Key properties:

```
class MeshService extends ChangeNotifier {
  // Node registry: maps node ID → NodeInfo (IP, battery, triage,
lastSeen)
  final Map<String, NodeInfo> nodeRegistry = {};

  // Routing table: maps node ID → best known route (next-hop IP)
  final Map<String, String> routingTable = {};

  // UUID deduplication cache: FIFO, max 500 entries
  final LinkedHashMap<String, int> seenMessages = LinkedHashMap();
  static const int MAX_SEEN_CACHE = 500;

  // Message streams for UI layers
  final StreamController<MeshMessage> chatStream =
StreamController.broadcast();
  final StreamController<MeshMessage> sosStream  =
StreamController.broadcast();
  final StreamController<String>      logStream  =
StreamController.broadcast();
  final StreamController<List<NodeInfo>> nodesStream =
StreamController.broadcast();

  // Sub-components
  late WifiDirectService  wifiService;
  late StoreForwardQueue  sfQueue;
  late BridgeManager      bridgeMgr;
  late MeshRouter         router;

  // Device identity (persisted in SharedPreferences)
  late String deviceId;   // format: Node_XXXXXX (6-digit random)
  late String deviceName; // user-set display name
  late String groupId;    // current group ID, e.g., GRP_620992
  bool isGroupOwner = false;
  String? groupOwnerIp;   // null when client, '192.168.49.1' when GO
}
```

Key methods:

```
// Initialize all sub-components and begin discovery
Future<void> initialize() async { ... }
```

```
// Process an incoming raw JSON message string from WifiDirectService
// Runs complete SRP pipeline: dedup → deliver → TTL → queue → forward
void processIncoming(String rawJson) { ... }

// Send a new message from this device into the mesh
Future<void> sendMessage(MeshMessage msg) async { ... }

// Send SOS: build SOS message, set P=1, repeat 3×, trigger StoreForward
Future<void> sendSOS({required LatLng location, required TriageState
triage}) async { ... }

// Called by WifiDirectService when group formation completes
void onGroupFormed({required bool isGO, required String? goIp}) { ... }

// Called every 5s heartbeat cycle; prune stale nodes (>15s timeout)
void _heartbeatTick() { ... }

// Flush StoreForwardQueue for a newly reconnected node
void onNodeReconnected(String nodeId) { ... }
```

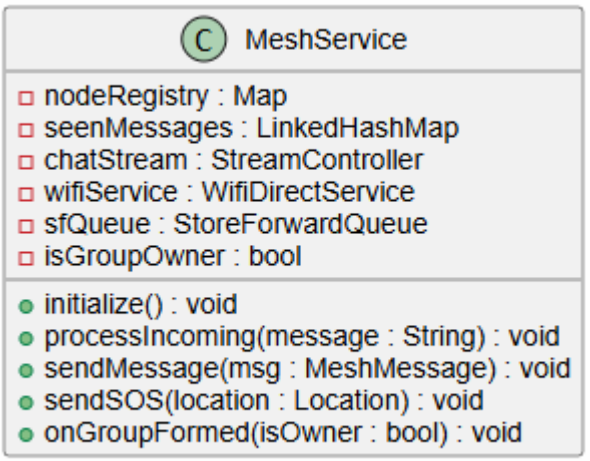


Fig. 2.2: MeshService Class Diagram — Properties, Streams, and Key Methods

2.4.2 WifiDirectService

WifiDirectService owns all low-level socket I/O. It is responsible for the full lifecycle of network connections: UDP beacon broadcasting, TCP server/client management, message framing/deframing, and the MethodChannel interface to the native Kotlin P2P layer. It runs socket operations in Dart isolates to prevent UI blocking.

```
class WifiDirectService {
  // Port constants
  static const int PORT_PRIMARY = 8888;   // TCP data transport
  static const int PORT_BRIDGE  = 8889;   // TCP inter-GO bridge
  static const int PORT_BEACON  = 44444;  // UDP discovery broadcast

  // TCP server (active only when this device is GO)
  ServerSocket? _serverSocket;
  final Map<String, Socket> _clientSockets = {};  // nodeId → Socket
  final Map<Socket, String> _socketBuffers = {};  // per-socket newline
buffer

  // TCP client socket (active when this device is Client)
  Socket? _goSocket;

  // UDP beacon socket
  RawDatagramSocket? _udpSocket;
  Timer? _beaconTimer;  // fires every 15 seconds

  // Cleanup timer: checks socket health every 30 seconds
  Timer? _cleanupTimer;

  // MethodChannel to Kotlin P2P bridge
  final MethodChannel _p2pChannel =
MethodChannel('wifi_direct_channel');
  final EventChannel  _p2pEvents  = EventChannel('wifi_direct_events');
}
```

Socket cleanup policy: A periodic timer checks every 30 seconds for sockets with no traffic in the past 60 seconds. Stale sockets are closed and removed from _clientSockets. If the GO socket itself becomes stale, the service triggers a full reconnection cycle. This prevents silent socket accumulation that would exhaust the Android file descriptor limit.

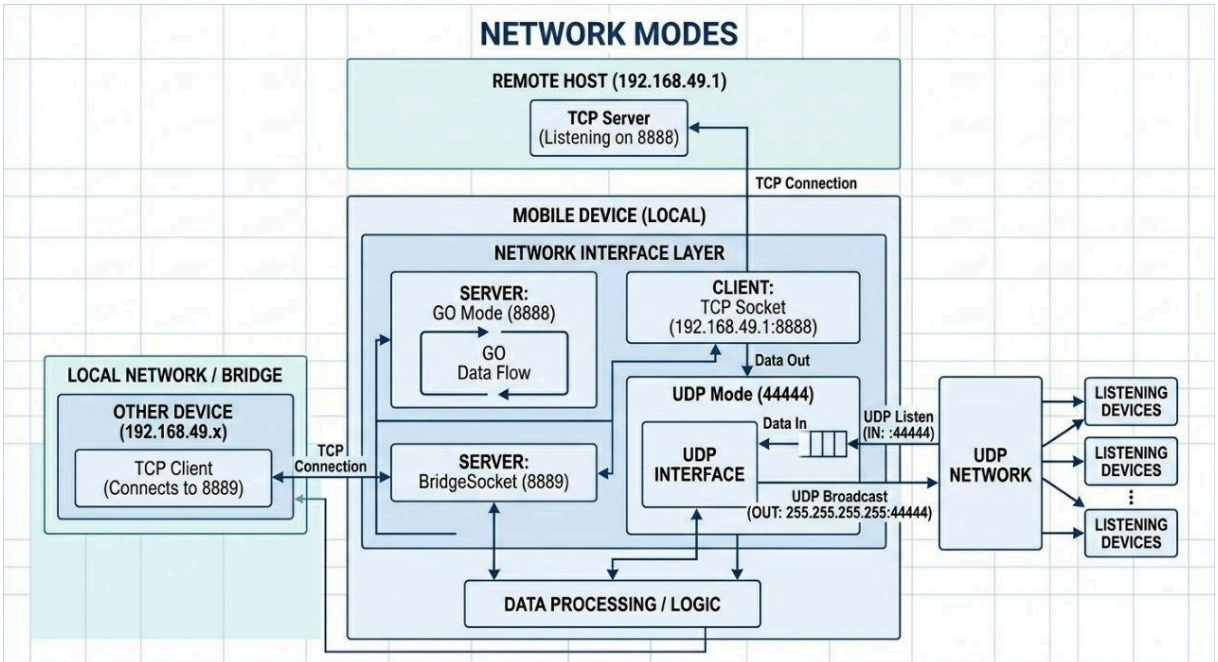


Fig. 2.3: WifiDirectService — Ports, Socket Management, and Key Methods

2.4.3 APP_BEACON: The UDP Discovery Protocol

Every 15 seconds, WifiDirectService broadcasts an APP_BEACON datagram to the subnet broadcast address 192.168.49.255 on port 44444. The beacon payload is a JSON object containing the sending device's complete identity and network state:

```
// APP_BEACON payload structure (broadcast every 15 seconds)
// Destination: 192.168.49.255:44444 (Wi-Fi Direct subnet broadcast)
{
  "type":        "APP_BEACON",
  "nodeId":      "Node_620992",       // persistent device identity
  "groupId":     "GRP_620992",        // current Wi-Fi Direct group
  "isGO":        true,                // is this device the Group Owner?
  "goIp":        "192.168.49.1",      // IP of current GO
  "role":        "RESCUER",           // RESCUER | VICTIM
  "triageState": "GREEN",             // NONE | GREEN | YELLOW | RED
  "battery":     76,                  // battery percentage 0-100
  "gps": { "lat": 35.3035, "lng": 4.1755 },
  "timestamp":   1748472479239,       // Unix ms
  "version":     "1.5.0"             // app version for compatibility
}
```

Upon receiving an APP_BEACON from a known or unknown node, WifiDirectService takes the following actions: (1) update the nodeRegistry entry for that nodeId; (2) if the

sender is a Group Owner (isGO=true) and no TCP connection to that GO exists, initiate a TCP connection to goIp:8888; (3) if a StoreForwardQueue entry exists for that nodeId, trigger a flush. The beacon interval of 15 seconds was chosen as the minimum that keeps peer lists current while maintaining acceptable battery overhead.

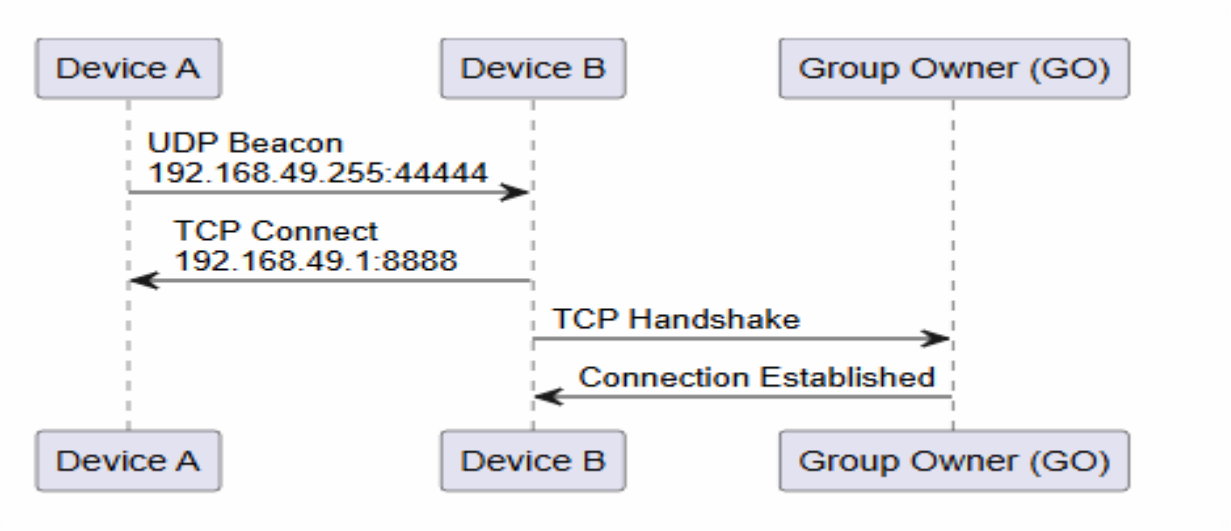


Fig. 2.4: APP_BEACON Discovery Sequence: Broadcast → Receipt → TCP Connection

2.4.4 MeshMessage — The Universal Message Format

All communication in SPAN RESCUE uses a single serializable Dart class called MeshMessage. It is the only data structure passed between layers and the only structure serialized to JSON for TCP/UDP transmission. Its design ensures that every routing decision can be made by inspecting only the message fields, without additional context.

```
class MeshMessage {
  final String   messageId;     // UUID v4 — primary deduplication key
  final String   senderId;      // Node_XXXXXX — source device
  final String   senderName;    // user display name
  final String   recipientId;   // target node ID or 'BROADCAST'
  final MsgType  messageType;   // MSG | SOS | BEACON | PROBE |
GO_BEACON | VOICE
  final String   payload;       // text, GPS string, audio base64, or
beacon JSON
  final int      hopCount;      // relay hops traversed; starts at 0
  final int      maxHops;       // TTL ceiling; default 10
  final int      timestamp;     // Unix ms at creation; not updated by
relays
  final List<String> seenBy;    // group IDs that relayed this message
```

```
  final int      priority;      // 1=SOS, 2=LOCATION, 3=MSG/VOICE,
4=BEACON
  final int?     batteryLevel;  // optional: sender battery %
  final TriageState? triageState; // SOS only: NONE|GREEN|YELLOW|RED
  final String?  medicalState;  // SOS only: human-readable e.g.
'Medical State: RED'
  final LatLng?  location;      // SOS/LOCATION: GPS coordinates
  final String?  groupId;       // sender's current group ID

  // Serialize to JSON string + newline delimiter
  String toJsonLine() => jsonEncode(toJson()) + '\n';

  // Deserialize from JSON string
  factory MeshMessage.fromJson(Map<String, dynamic> json) { ... }

  // Implements Comparable for PriorityQueue ordering
  int compareTo(MeshMessage other) =>
priority.compareTo(other.priority);
}
```

Field	Type / Constraint	Description and Routing Role
messageId	String (UUID v4)	Primary key for deduplication cache; generated once at source; never modified by relays
senderId	String Node_XXXXX	Source identity; used to update nodeRegistry on receipt
recipientId	String or 'BROADCAST'	Routing target; 'BROADCAST' triggers delivery to all nodes in mesh
messageType	Enum MsgType	MSG | SOS | BEACON | PROBE | GO_BEACON | VOICE — governs priority and delivery
payload	String	Text for MSG; 'lat,lng|bat%|msg' for SOS; base64 audio for VOICE; JSON for GO_BEACON
hopCount	int ≥ 0	Relay counter; incremented by 1 BEFORE each forward; initialized to 0 at source
maxHops	int 1–20; default 10	TTL ceiling; message silently dropped when hopCount ≥ maxHops
seenBy	List<String>	Group IDs that already relayed; prevents inter-group broadcast storms
priority	int 1–4	1=SOS (highest), 2=LOCATION, 3=MSG/VOICE, 4=BEACON (lowest)
triageState	Enum optional	NONE | GREEN | YELLOW | RED — medical classification; SOS messages only
location	LatLng optional	GPS coordinates; present in all SOS and LOCATION messages

groupId	String optional	Sender's current group; used by BridgeManager to populate seenBy[]

Table 2.3: MeshMessage complete field specification

2.4.5 StoreForwardQueue

The StoreForwardQueue ensures that no message addressed to a specific recipient is permanently lost due to temporary network partitioning. When MeshService's SRP routing pipeline cannot find an active TCP path to the target node, the message is serialized to JSON and stored in Flutter's shared_preferences under a key indexed by recipientId.

```
class StoreForwardQueue {
  // Key format: 'sfq_${recipientId}_${messageId}'
  // Storage: Flutter shared_preferences (JSON-encoded)

  // Enqueue a message for deferred delivery
  Future<void> enqueue(MeshMessage msg) async {
    final key = 'sfq_${msg.recipientId}_${msg.messageId}';
    await prefs.setString(key, msg.toJsonLine());
    _log('StoreForward: queued ${msg.messageId} for
${msg.recipientId}');
  }

  // Flush all queued messages for a specific node (called on
reconnection)
  // Returns messages sorted by priority ASC (SOS delivered first)
  Future<List<MeshMessage>> flush(String recipientId) async {
    final keys = prefs.getKeys().where((k) =>
k.startsWith('sfq_$recipientId'));
    final msgs = keys.map((k) =>
MeshMessage.fromJson(jsonDecode(prefs.getString(k)!))).toList();
    msgs.sort((a, b) => a.priority.compareTo(b.priority)); // SOS first
    for (final k in keys) await prefs.remove(k); // clean up after flush
    return msgs;
  }

  // Expire messages older than maxAgeMs (default: 30 minutes)
  Future<void> expireOld({int maxAgeMs = 30 * 60 * 1000}) async { ... }
}
```

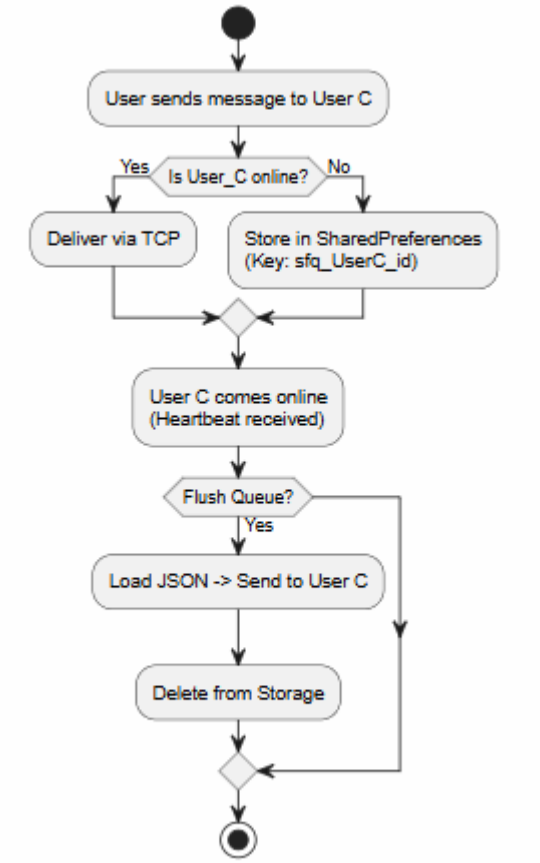


Fig. 2.5: StoreForwardQueue: Enqueue → Node Reconnection → Flush Lifecycle

2.4.6 BridgeManager

BridgeManager extends the effective mesh beyond the single-group constraint of Wi-Fi Direct. When MeshService detects that a received APP_BEACON announces a different Group Owner (different goIp), BridgeManager attempts to establish a TCP connection to that GO on port 8889. If successful, it registers this as a bridge connection and uses it to forward messages that carry group IDs not yet in their seenBy[] array.

```
class BridgeManager {
  // Map of adjacent GO IP → active bridge socket
  final Map<String, Socket> bridgeSockets = {};

  // Establish TCP bridge to an adjacent Group Owner
  Future<void> connectToBridge(String goIp) async {
    final socket = await Socket.connect(goIp, PORT_BRIDGE,
        timeout: Duration(seconds: 5));
    bridgeSockets[goIp] = socket;
```

```
    _log('BRIDGE SERVER: Connected to $goIp:$PORT_BRIDGE');
    _listenOnBridge(socket);
  }

  // Forward a message to all adjacent groups not yet in seenBy[]
  void forwardToAdjacentGroups(MeshMessage msg, String currentGroupId) {
    for (final entry in bridgeSockets.entries) {
      if (!msg.seenBy.contains(entry.key)) {
        final updated = msg.copyWith(
          seenBy: [...msg.seenBy, currentGroupId],
          hopCount: msg.hopCount + 1,
        );
        _sendOnSocket(entry.value, updated);
        _log('Bridge fwd → ${entry.key} hop=${updated.hopCount}');
      }
    }
  }
}
```

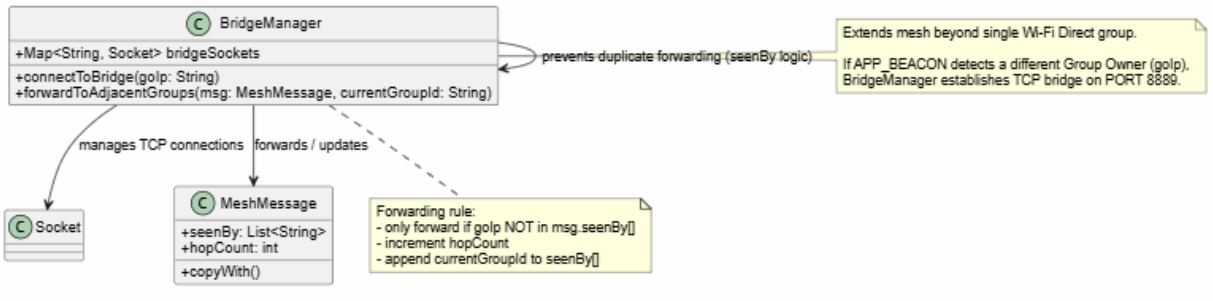


Fig. 2.6: BridgeManager: Multi-Group GO Interconnection via Port 8889

2.4.7 P2pHelper — Native Kotlin Bridge

P2pHelper is the Kotlin AndroidManifest class that bridges the native Android WifiP2pManager API to Dart via Flutter's MethodChannel and EventChannel. It registers the required BroadcastReceiver for all three critical P2P action types and forwards received events to the Dart layer.

```
// Kotlin — P2pHelper.kt — MethodChannel handler
channel.setMethodCallHandler { call, result ->
  when (call.method) {
    'discoverPeers'   -> manager.discoverPeers(channel, actionListener)
    'connect'         -> manager.connect(channel, config,
actionListener)
    'createGroup'     -> manager.createGroup(channel, actionListener)
```

```
    'requestGroupInfo'-> manager.requestGroupInfo(channel,
groupInfoListener)
    'removeGroup'     -> manager.removeGroup(channel, actionListener)
    else              -> result.notImplemented()
  }
}

// BroadcastReceiver forwards events to Dart via EventChannel
receiver = object : BroadcastReceiver() {
  override fun onReceive(ctx: Context, intent: Intent) {
    when (intent.action) {
      WIFI_P2P_STATE_CHANGED_ACTION    ->
eventSink?.success(mapOf('event','stateChanged',...))
      WIFI_P2P_PEERS_CHANGED_ACTION    ->
eventSink?.success(mapOf('event','peersChanged',...))
      WIFI_P2P_CONNECTION_CHANGED_ACTION -> {
        eventSink?.success(mapOf('event','connectionChanged',...))
        // CRITICAL: must re-discover peers after every connection event
        manager.discoverPeers(channel, actionListener)
      }
    }
  }
}
```

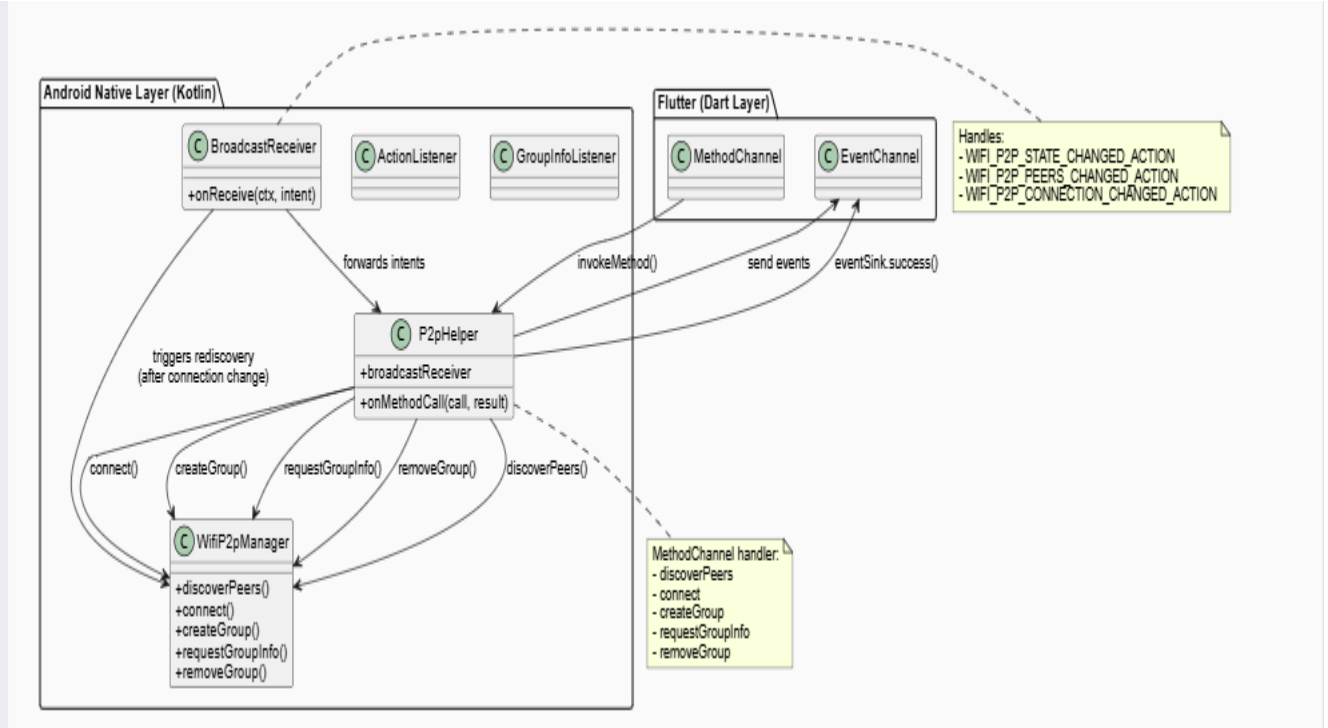


Fig. 2.7: P2pHelper: MethodChannel Interface to Native Android WifiP2pManager

2.5 Smart Routing Protocol (SRP) — Complete Specification

2.5.1 Design Philosophy

The Smart Routing Protocol is a controlled flooding algorithm specifically designed for the constraints of the SPAN RESCUE deployment scenario: maximum 20 nodes, frequent topology changes due to device mobility and power cycling, and an absolute requirement that SOS signals reach all reachable nodes even when the network topology is partially unknown. OLSR and AODV were explicitly rejected: OLSR's periodic topology broadcasts would generate unacceptable overhead in a battery-limited emergency deployment, and AODV's route discovery delay is incompatible with the zero-delay SOS requirement. Controlled flooding with three storm-prevention mechanisms provides the delivery guarantee required while keeping overhead within acceptable bounds.

2.5.2 The Complete Routing Pipeline

Every message entering MeshService.processIncoming() passes through the following five-step pipeline in strict order:

```
// MeshRouter — process(MeshMessage msg) — complete SRP pipeline

// ── STEP 1: UUID DEDUPLICATION ──────────────────────────────────
if (seenMessages.containsKey(msg.messageId)) {
  _log('SRP: Duplicate dropped ${msg.messageId}');
  return;  // silent discard — never ACK, never forward
}
seenMessages[msg.messageId] = DateTime.now().millisecondsSinceEpoch;
if (seenMessages.length > MAX_SEEN_CACHE) {
  seenMessages.remove(seenMessages.keys.first); // FIFO eviction
}

// ── STEP 2: NODE REGISTRY UPDATE ────────────────────────────────
nodeRegistry[msg.senderId] = NodeInfo.fromMessage(msg);
// Trigger StoreForward flush if this sender had pending messages
if (sfQueue.hasPending(msg.senderId)) { onNodeReconnected(msg.senderId);
}

// ── STEP 3: LOCAL DELIVERY ──────────────────────────────────────
if (msg.recipientId == deviceId || msg.recipientId == 'BROADCAST') {
  _deliverToUI(msg); // publish to chatStream or sosStream
}
```

```
// ── STEP 4: TTL ENFORCEMENT ──────────────────────────────────────
final forwarded = msg.copyWith(hopCount: msg.hopCount + 1);
if (forwarded.hopCount >= forwarded.maxHops) {
  _log('SRP: TTL expired hop=${forwarded.hopCount}
id=${msg.messageId}');
  return;  // silent drop — do not forward
}

// ── STEP 5: PRIORITY QUEUE + JITTER FORWARD ──────────────────────
priorityQueue.add(forwarded); // PriorityQueue ordered by priority ASC
final delay = (msg.priority == 1) ? 0
    : (Random().nextInt(500));  // 0ms for SOS, 0-500ms for MSG/CHAT
await Future.delayed(Duration(milliseconds: delay));
wifiService.broadcastToClients(forwarded); // TCP to all connected
clients
bridgeMgr.forwardToAdjacentGroups(forwarded, groupId); // TCP bridge
```

2.5.3 Priority Queue and Message Types

Priority	Message Type	Forwarding Delay	Delivery Guarantee
P=1 (highest)	SOS	0ms; repeated 3×	TCP + StoreForward + bridge; all reachable nodes
P=2	LOCATION	0ms	TCP to all connected; no StoreForward
P=3	MSG / VOICE	0–500ms jitter	TCP best-effort; StoreForward if target unreachable
P=4 (lowest)	BEACON / PROBE	End of queue	Informational only; may be dropped under load

Table 2.4: Message type taxonomy — priority, delay, and delivery guarantee

2.5.4 SOS Multi-Layer Delivery

SOS messages receive a dedicated multi-layer delivery mechanism that goes beyond normal SRP forwarding. When a device calls sendSOS(), the following sequence executes:

1. MeshMessage of type SOS is created with priority=1, triageState, location, batteryLevel, and a fresh UUID.
2. The message is immediately broadcast via TCP to all currently connected clients (no queue — bypasses PriorityQueue).

3. The message is forwarded to all adjacent groups via BridgeManager (seenBy[] check prevents re-entry into already-visited groups).
4. Steps 1–3 are repeated 3 times with a 200ms inter-repetition delay to overcome potential packet loss.
5. Any SOS copy that cannot be delivered (no TCP path to recipient) is enqueued in StoreForwardQueue with priority=1.
6. When the target node reconnects (detected via UDP beacon), the StoreForwardQueue is flushed in priority order — SOS messages first.

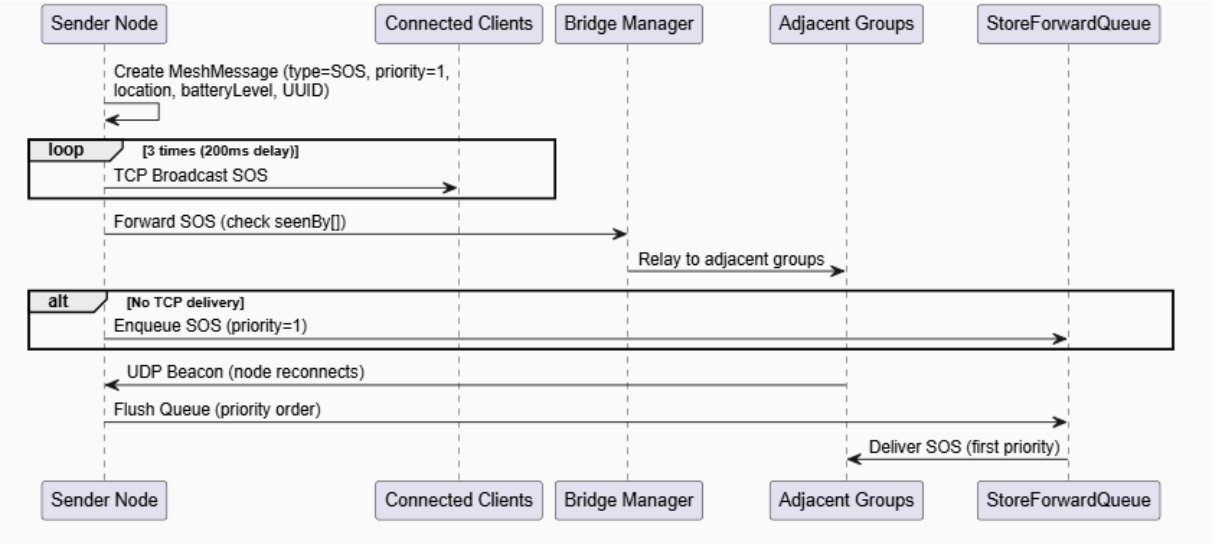


Fig. 2.8: SOS Multi-Layer Delivery: TCP → 3× Repeat → Bridge → StoreForward

2.6 Multi-Group Bridge Topology

2.6.1 The Single-Group Hardware Constraint

A fundamental limitation of the Android Wi-Fi Direct API is that a device can participate in only one P2P group at the hardware level at any given time. This creates a hard ceiling: a single Wi-Fi Direct star group supports at most 4–5 clients plus the GO (≈6 devices reliably). Supporting 20 devices therefore requires connecting multiple independent star groups. BridgeManager provides this capability through software-layer TCP relay: devices designated

as bridge nodes maintain connections to adjacent GOs via persistent TCP sockets on port 8889, forwarding messages between groups without hardware-level multi-group membership.

2.6.2 GO Beacon for Group Advertisement

Each Group Owner broadcasts a periodic GO_BEACON message every 15 seconds. This beacon is transmitted both as a UDP broadcast (via the APP_BEACON mechanism) and as a TCP message to all currently connected clients. Its purpose is to advertise the GO's existence and TCP address to potential bridge clients:

```
// GO_BEACON structure — broadcast every 15 seconds by Group Owners
// Enables BridgeManager on adjacent devices to detect and connect
{
  "messageType": "GO_BEACON",
  "senderId":    "Node_620992",
  "groupId":     "GRP_620992",
  "goIp":        "192.168.49.1",
  "bridgePort":  8889,
  "memberCount": 4,           // current number of connected clients
  "timestamp":   1748472479239
}

// System Logs output (visible in Screen 4):
// 2026-05-28T14:47:59.239878 GO_BEACON SENT: GRP_620992
// Members:1 IP:192.168.49.1:8889
```

2.6.3 Complete 20-Device Topology

Group	GO Node	Client Nodes	Bridge Node(s)	Connects To
A	Node_620992 (GO-A)	N_001, N_002, N_003, N_BrA	N_BrA	Group B (port 8889)
B	Node_810452 (GO-B)	N_004, N_005, N_BrBA, N_BrBC	N_BrBA / N_BrBC	Groups A & C
C	Node_334177 (GO-C)	N_008, N_009, N_BrCB, N_BrCD	N_BrCB / N_BrCD	Groups B & D
D	Node_991025 (GO-D)	N_012, N_013, N_014, N_015	N_015 (SOS demo)	Group C only

Table 2.5: Multi-group topology — 20-device role assignment across 4 groups

The worst-case message path (Group A client to Group D client) traverses: N_001 → GO-A (h1) → N_BrA (h2) → GO-B via bridge (h3) → N_BrBC (h4) → GO-C via bridge

(h5) → N_BrCD (h6) → GO-D via bridge (h7) → N_015 (h8). Total: 8 hops < maxHops(10). The seenBy[] array accumulates [GRP_A, GRP_B, GRP_C, GRP_D] preventing any group processing the message twice.

Fig. 2.9: Full 20-Device Topology: 4 Groups, 3 Bridges, Complete Routing Paths

2.7 Wi-Fi Direct Connection Lifecycle

The Wi-Fi Direct connection lifecycle in SPAN RESCUE is managed as an explicit state machine with six states. Understanding this state machine is essential for understanding the self-healing behavior and the role of the P2pHelper MethodChannel events.

State Transition	Trigger Event	Required Actions

IDLE → BLE_DISCOVERING	App launch / network loss	Start UDP beacon timer (15s); request P2P peer discovery via MethodChannel
BLE_DISCOVERING → WFD_CONNECTING	Peer found in PEERS_CHANGED	Call P2pHelper.connect(peerConfig) via MethodChannel
WFD_CONNECTING → GROUP_FORMED	CONNECTION_CHANGED_ACTION	Call requestGroupInfo() → determine GO/Client role; RE-CALL discoverPeers() ← CRITICAL
GROUP_FORMED → TCP_ACTIVE	Role determined by requestGroupInfo()	GO: open ServerSocket(8888+8889); Client: TCP connect to 192.168.49.1:8888
TCP_ACTIVE → GROUP_FORMED	Client joins existing group	Update nodeRegistry; trigger StoreForward flush for new node
TCP_ACTIVE → IDLE	Heartbeat timeout (15s) / socket closed	Close all sockets; restart UDP beacon; re-initiate P2P discovery

Table 2.6: Wi-Fi Direct state transitions and required actions

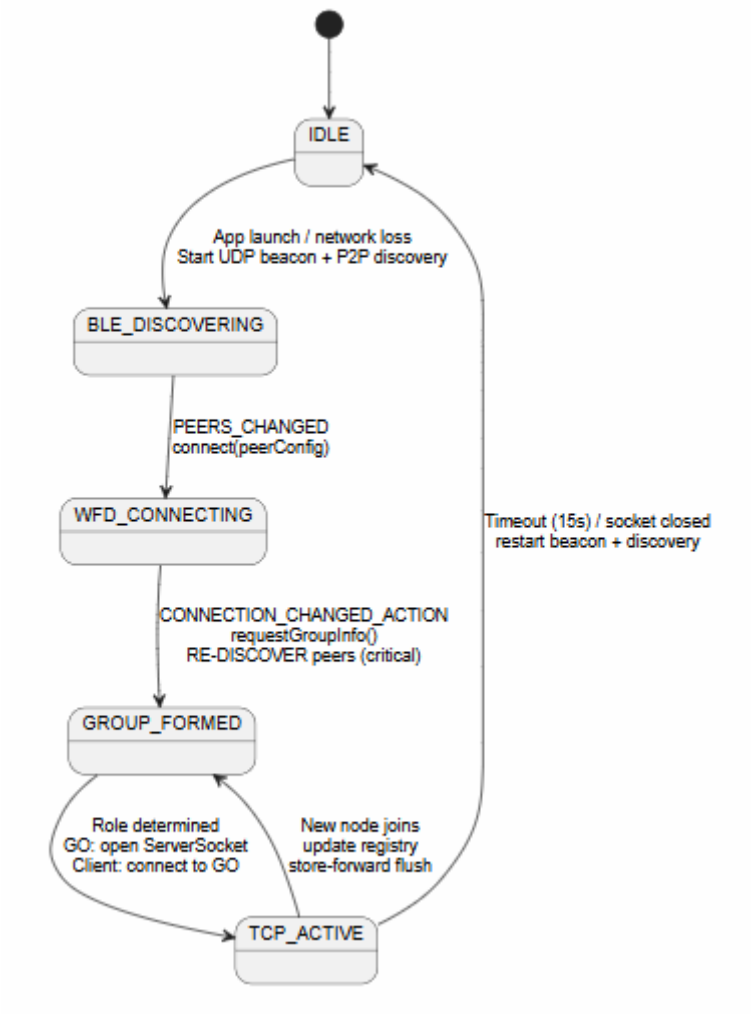


Fig. 2.10: Wi-Fi Direct Connection Lifecycle State Machine Diagram

2.8 UML Formal Models

2.8.1 Use Case Diagram

Two human actors interact with SPAN RESCUE: the Rescue Worker (field coordinator who manages the mesh) and the Victim (emergency sender who primarily activates SOS). The autonomous system actor MeshService operates without user intervention for routing, discovery, and heartbeat.

- UC1 — Form P2P Mesh (auto): includes UDP beacon, WFD group formation, TCP initialization; precondition for all other use cases;

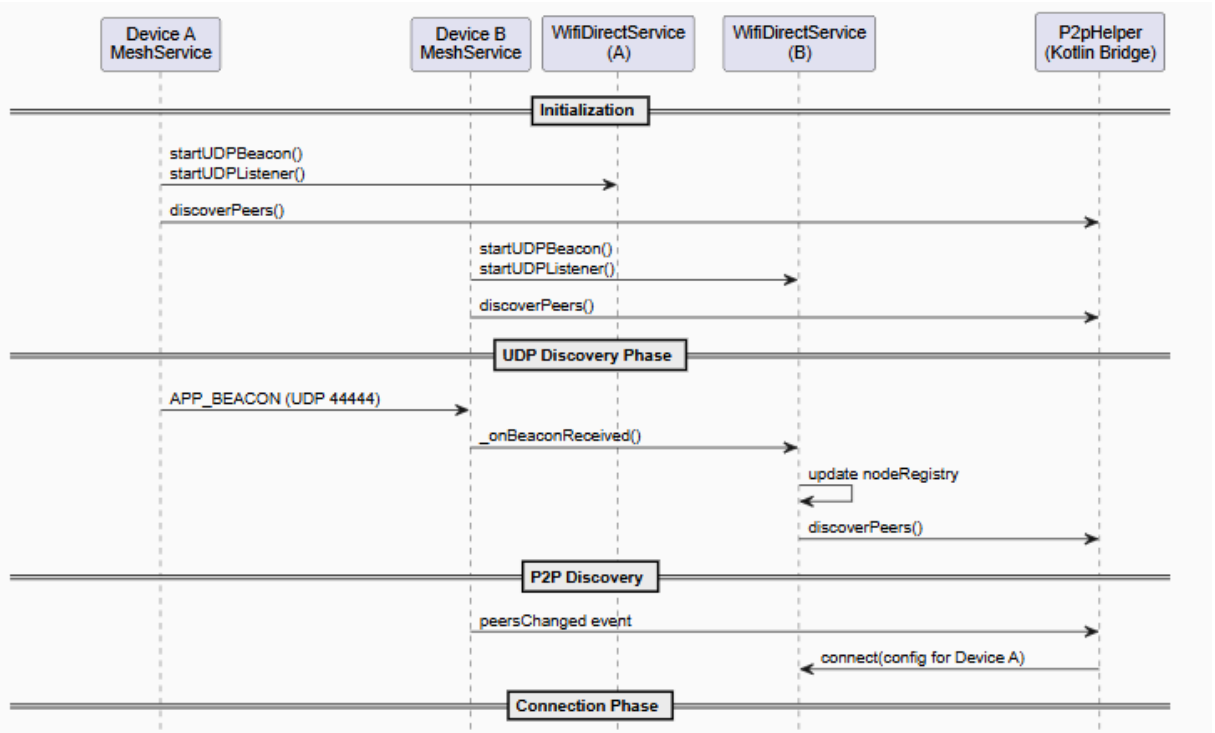
- UC2 — Send SOS Alert: extends UC1; requires GPS; includes triage selection, 3× repeat, StoreForward; confirmation dialog prevents accidental activation;
- UC3 — Receive SOS: triggered by incoming P=1 message; full-screen red overlay, system notification, quick-reply buttons (I'm coming / Hold on / Acknowledge);
- UC4 — Send Text Message: extends UC1; P=3 queuing, newline-delimited JSON, SRP relay;
- UC5 — Send Voice Message: extends UC1; flutter_sound recording, base64 encoding, P=3 queuing;
- UC6 — View Tactical Nodes Directory: reads nodeRegistry; shows 16+ rescuers with distance, battery, triage;
- UC7 — View Tactical AODV Topology: Canvas-rendered live graph; tap-node detail panel shows 8 fields;
- UC8 — Monitor System Logs: ISO 8601 real-time P2P, UDP, TCP, SRP events;
- UC9 — View Mesh Map + Radar: OpenStreetMap overlay with node markers; tactical radar with 4 range rings;
- UC10 — Relay Message (MeshService): automated SRP step 5; triggered by any message with hopCount < maxHops;
- UC11 — Store Message (MeshService): triggered when recipientId unreachable; shared_preferences persistence;
- UC12 — Flush Queue on Reconnect (MeshService): triggered by UDP beacon from previously offline node.

Fig. 2.11: UML Use Case Diagram — Actors, Use Cases, and Dependencies

2.8.2 Sequence Diagram — Discovery and Group Formation

```
Device A (fresh launch):
  MeshService.initialize()
    → WifiDirectService.startUDPBeacon() // port 44444, every 15s
    → P2pHelper.discoverPeers()          // MethodChannel → Kotlin
    → WifiDirectService.startUDPListener() // listen on 44444

Device B (fresh launch, same physical location):
  MeshService.initialize()
    → identical startup sequence

~15 seconds: Device A broadcasts APP_BEACON to 192.168.49.255:44444
  Device B receives beacon → WifiDirectService._onBeaconReceived()
    → nodeRegistry.update('Node_A')
    → P2pHelper.discoverPeers() // also trigger P2P peer scan

~15 seconds: Device B receives PEERS_CHANGED event from Kotlin P2P stack
  → P2pHelper triggers MethodChannel event: 'peersChanged'
```

```
  → WifiDirectService._onPeersChanged(peers)
    → P2pHelper.connect(config) for Device A

~3 seconds: CONNECTION_CHANGED_ACTION received by Kotlin receiver
  → P2pHelper forwards to Dart: 'connectionChanged'
  → WifiDirectService._onConnectionChanged()
    → P2pHelper.requestGroupInfo() // determine role
    → P2pHelper.discoverPeers()   // ← CRITICAL: re-discover after
connect

GO election result: Device A elected Group Owner
  Device A: isGroupOwner=true, ip=192.168.49.1
    → WifiDirectService.startTCPServer(8888) // ServerSocket
    → WifiDirectService.startBridgeServer(8889)
    → MeshService.onGroupFormed(isGO:true)
    → System Logs: 'AUTO-PROMOTED: Now Group Owner @ 192.168.49.1'
    → System Logs: 'BRIDGE SERVER: Listening on :8889'

  Device B: isGroupOwner=false, goIp=192.168.49.1
    → WifiDirectService.connectToGO('192.168.49.1', 8888)
    → TCP 3-way handshake complete
    → MeshService.onGroupFormed(isGO:false)
    → UDP beacon updated: status=MESH_ACTIVE
```



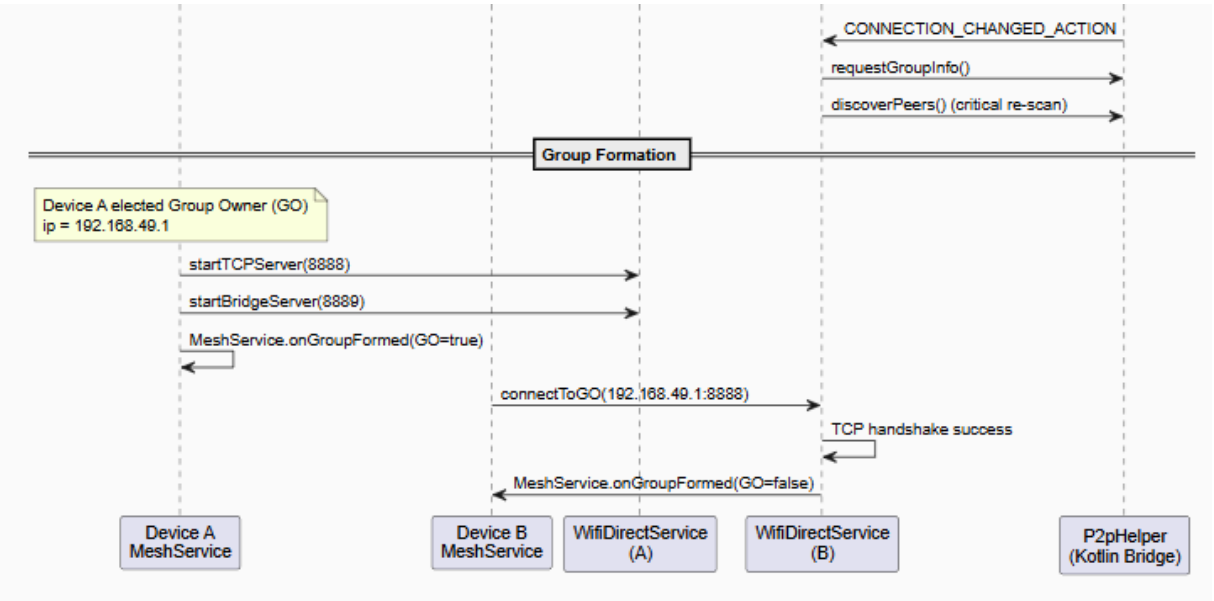
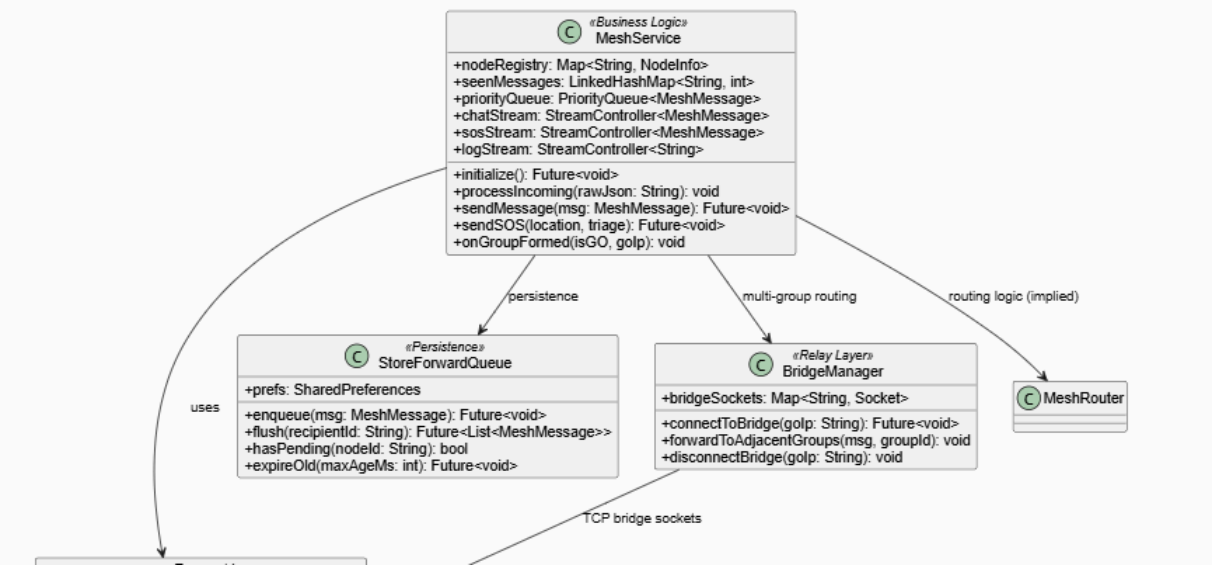


Fig. 2.12: UML Sequence Diagram — UDP Discovery and TCP Group Formation

2.8.3 Class Diagram — Core Components

```
MeshService ──────────────────────── [Business Logic – Layer 2]
  – nodeRegistry: Map<String,NodeInfo>
  – seenMessages: LinkedHashMap<String,int>  // UUID → timestamp
  – priorityQueue: PriorityQueue<MeshMessage>
  – chatStream: StreamController<MeshMessage>
  – sosStream: StreamController<MeshMessage>
  – logStream: StreamController<String>
  – wifiService: WifiDirectService
  – sfQueue: StoreForwardQueue
  – bridgeMgr: BridgeManager
  – router: MeshRouter
  + initialize(): Future<void>
  + processIncoming(rawJson: String): void
  + sendMessage(msg: MeshMessage): Future<void>
  + sendSOS(location, triage): Future<void>
  + onGroupFormed(isGO, goIp): void

WifiDirectService ──────────────── [Transport Layer – Layer 3]
  – serverSocket: ServerSocket?       // GO: port 8888
  – bridgeSocket: ServerSocket?       // GO: port 8889
  – goSocket: Socket?                 // Client: TCP to GO
  – clientSockets: Map<String,Socket> // GO: nodeId → client socket
  – socketBuffers: Map<Socket,String> // newline delimiter buffers
  – udpSocket: RawDatagramSocket?     // port 44444
  – beaconTimer: Timer?               // 15s periodic
```

```
  + startTCPServer(port:8888): Future<void>
  + connectToGO(ip,port:8888): Future<void>
  + broadcastToClients(msg): void
  + sendToGO(msg): void
  + startUDPBeacon(): void

StoreForwardQueue ——————————————— [Persistence — Layer 2]
  — prefs: SharedPreferences
  + enqueue(msg: MeshMessage): Future<void>
  + flush(recipientId: String): Future<List<MeshMessage>>
  + hasPending(nodeId: String): bool
  + expireOld(maxAgeMs: int): Future<void>

BridgeManager ————————————————— [Multi-group relay — Layer 2]
  — bridgeSockets: Map<String,Socket> // goIp → bridge socket
  + connectToBridge(goIp: String): Future<void>
  + forwardToAdjacentGroups(msg, groupId): void
  + disconnectBridge(goIp: String): void

P2pHelper (Kotlin) ——————————————[Native Bridge — Layer 4]
  — MethodChannel: 'wifi_direct_channel'
  — EventChannel: 'wifi_direct_events'
  + discoverPeers(), connect(), createGroup()
  + requestGroupInfo(), removeGroup()
  → Forwards: stateChanged, peersChanged, connectionChanged
```



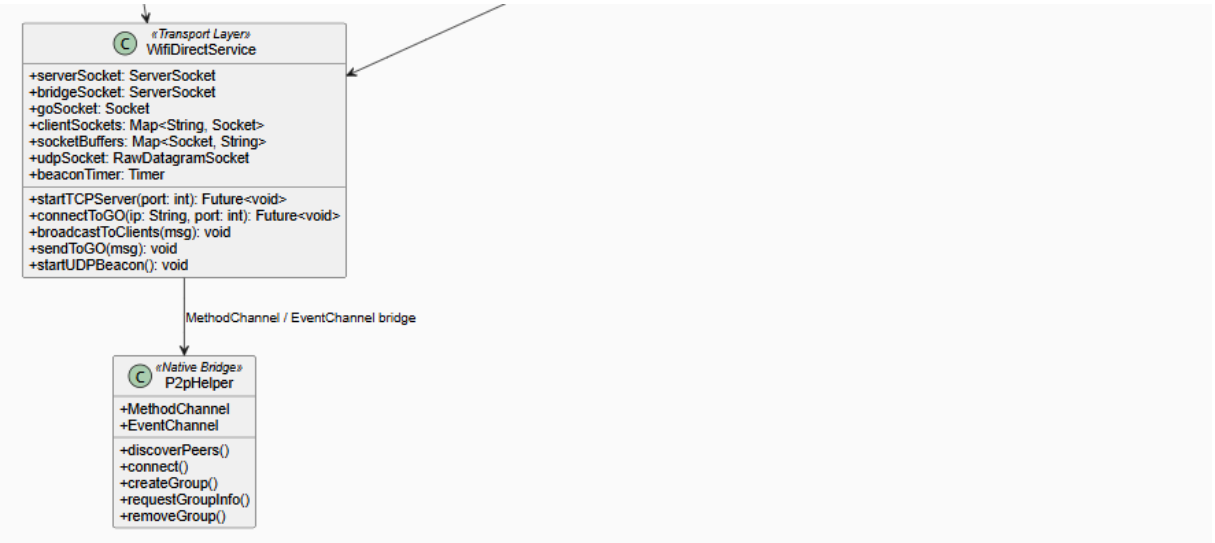


Fig. 2.13: UML Class Diagram — Core Networking Component Relationships

2.9 User Interface Design

2.9.1 Design System

Token	Value	Application
color-background	#000000 / #0A0A0A	All screen backgrounds — pure black for OLED efficiency
color-surface	#0D1B2A / #111827	Cards, panels, bottom nav bar, dialogs
color-accent-cyan	#00BCD4 / #00F5FF	GO nodes, active icons, borders, 'Mesh Network Active' label
color-sos-red	#D32F2F / #FF2244	SOS button, SOS cards, emergency overlay, RED triage nodes
color-victim-red	#F44336	VICTIM nodes in topology graph
color-bridge-orange	#FF8C00	Bridge nodes, YELLOW triage, warning states
color-safe-green	#00C853 / #00FF88	Connected state, GREEN triage, normal nodes
color-go-badge	#FFC107 (yellow)	GO badge label on Group Owner nodes in topology
color-text-primary	#FFFFFF / #E0E0E0	All primary body text
color-text-muted	#8899AA / #6B7280	Timestamps, hop counts, distance labels

font-tactical	Courier New / Space Mono	Node IDs, IPs, GPS coords, log entries — all tactical data
radar-color	#00FF00 (green)	Radar sweep line and concentric ring borders
min-touch-target	48dp × 48dp	All interactive elements — WCAG AA, gloved hands, stress ops

Table 2.12: UI design system tokens — colors, fonts, and interaction targets

2.9.2 Screen 1 — Mesh Map with Tactical Radar

The Mesh Map screen combines an OpenStreetMap overlay (flutter_map package, offline-capable tile caching) centered on the device's GPS position with a tactical radar view in the lower 40% of the screen. The header displays 'SPAN Mesh Tactical' with a real-time node count badge (e.g., '● 20 NODES' in green, '● 0 NODES' in gray). Two floating action buttons: a blue team FAB opens the Tactical Nodes Directory bottom sheet; a red SOS FAB opens the SOS confirmation dialog.

The tactical radar features: four concentric range rings labeled 125m, 250m, 375m, 500m; N/S/E/W cardinal direction labels; a rotating green sweep line at one revolution per 3 seconds; and connected nodes rendered as animated dots positioned by GPS bearing and RSSI-estimated distance. The user's position is a pulsing cyan crosshair marker.

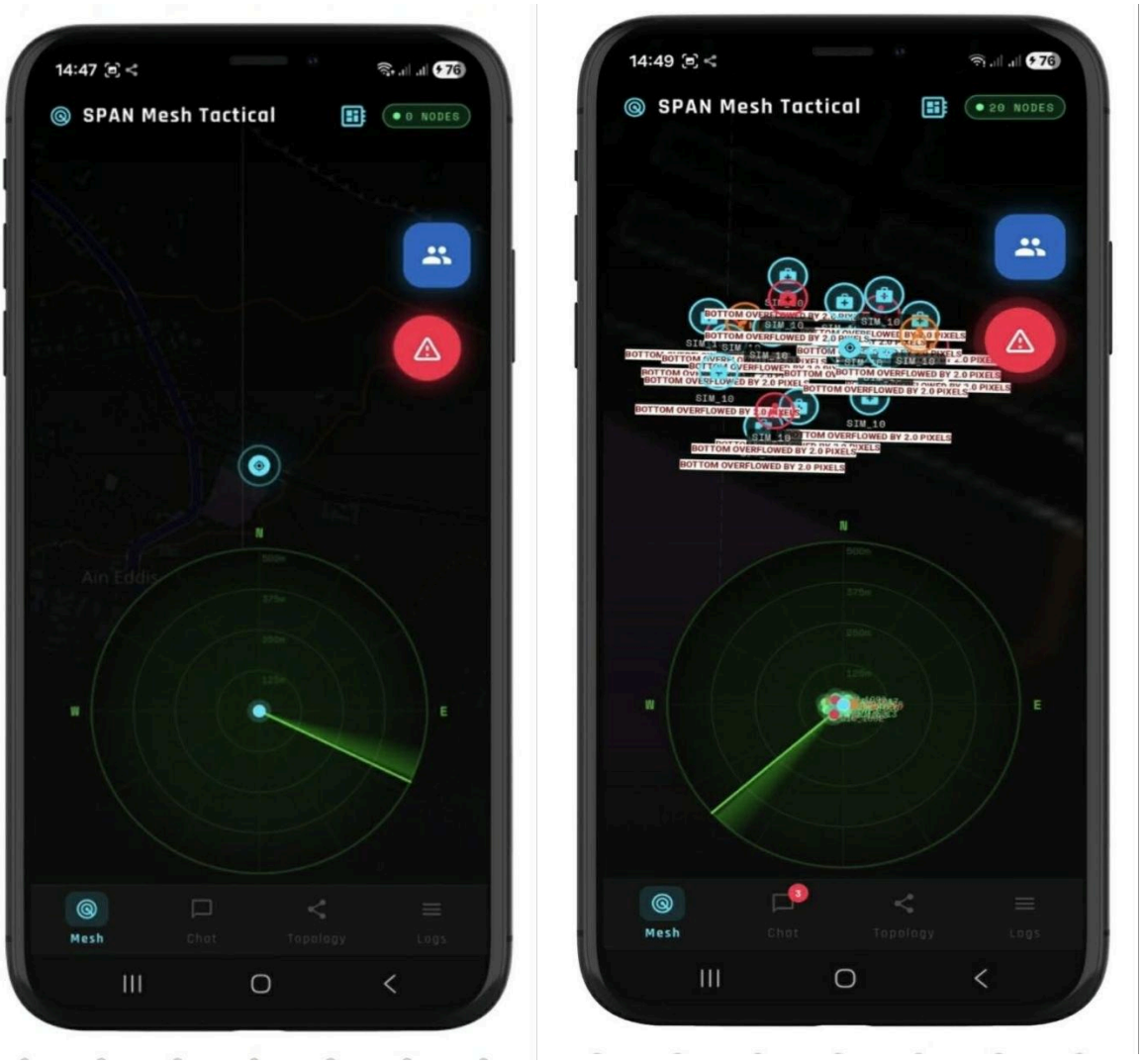


Fig. 2.14: Mesh Map Wireframe — Empty (0 Nodes) | Fig. 2.15: Mesh Map Screenshots (20 Nodes)

2.9.3 Screen 2 — Global Squad Chat

The chat screen header shows 'Global Squad' with 'Mesh Network Active' in green and four action icons (node directory, SOS, topology, overflow menu). Messages appear in three visual formats: standard text bubbles (dark background, monospace font, node ID badge, timestamp with GPS pin); voice message bubbles (waveform visualization, play button, duration); and SOS alert cards (full-width dark red card, '⚠ EMERGENCY SOS ALERT' header in amber, Victim ID, Medical State, GPS coordinates, message text in monospace). SOS confirmation uses a dialog with bilingual text:

```
// Arabic SOS dialog (RTL):
// '⚠ إرسال استغاثة؟'
```

```
// 'سيصل نداء الاستغاثة لكل الأجهزة في الشبكة فورًا'
// Buttons: [إرسال] / [إلغاء SOS]

// English SOS dialog:
// 'Confirm SOS'
// 'Send an emergency SOS alert to nearby nodes?'
// Buttons: [Cancel] / [Send SOS]
```

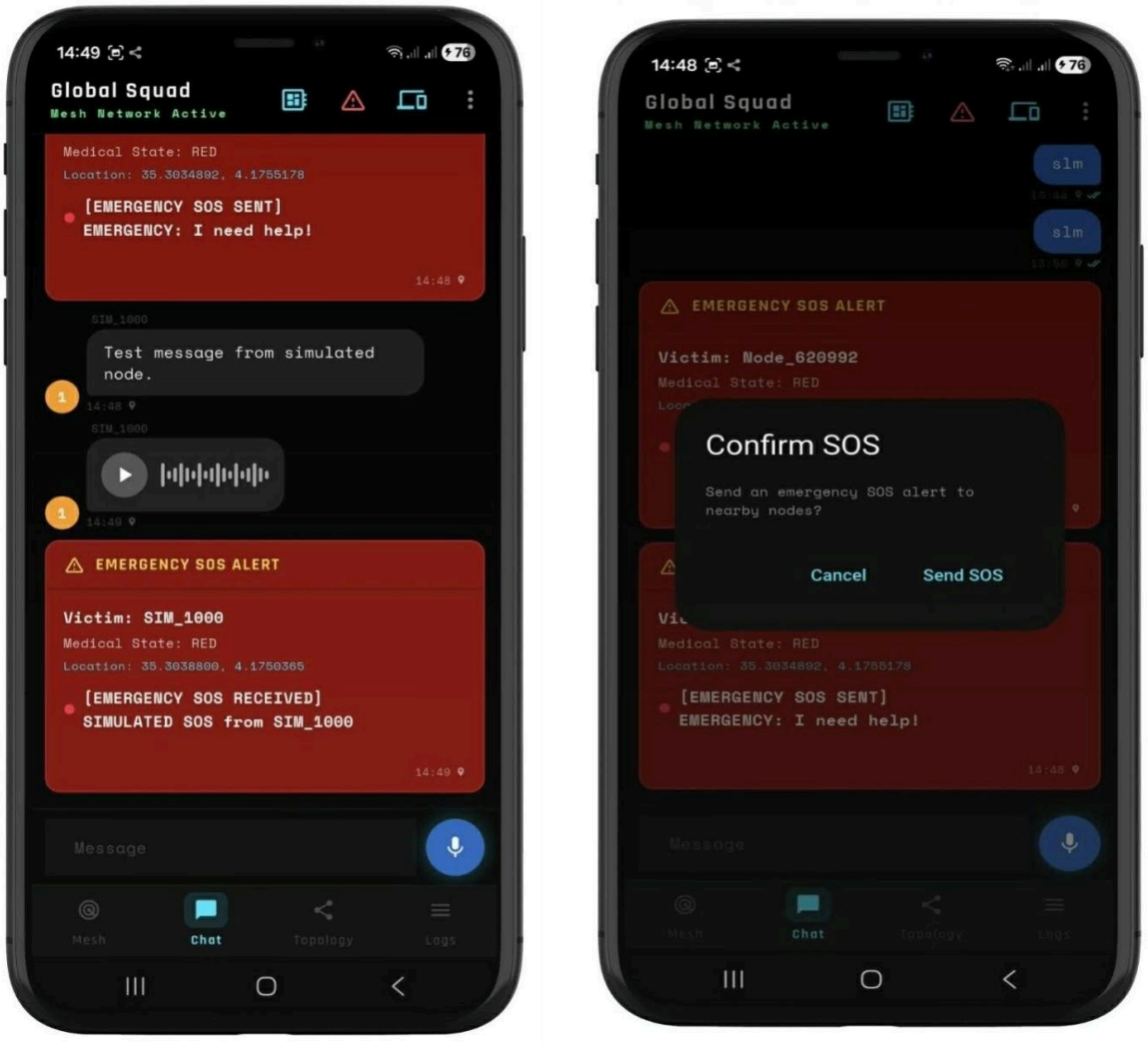


Fig. 2.16: Chat Screen Wireframe — Message Types | Fig. 2.17: Chat Screenshots — SOS Card and Voice Message

2.9.4 Screen 3 — Tactical AODV Topology

The topology screen renders a live force-directed graph using Flutter's CustomPainter. Nodes are circles with a ✚ (medical cross) inside, color-coded by state: cyan for normal

rescuer or 'Me' node; red for SOS sender or critical victim; orange for bridge node or YELLOW triage; Group Owner nodes carry a yellow 'GO' badge. Group labels (GRP_SIM_1, GRP_SIM_2, etc.) float above their respective GO nodes. Inter-group edges (bridge connections) are rendered in gold/orange. Tapping any node opens a bottom sheet detail panel containing: Battery, Distance (m), Role (RESCUER/VICTIM), Triage State, Connected Via, Group ID, Last Seen, and Hop Count.

Fig. 2.18: Topology — Me only | Fig. 2.19: Topology — 5 nodes | Fig. 2.20: Topology — 20 nodes

2.9.5 Screen 4 — System Logs

The System Logs screen provides a real-time terminal-style view of all network events. Entries are displayed in reverse chronological order (newest first) with ISO 8601 timestamps. All output uses monospace white-on-black rendering. The log entries directly observable in testing include:

```
// Startup sequence (Group Owner election)
2026-05-28T14:47:46.385038  AUTO-PROMOTED: No network found, becoming
Group Owner
2026-05-28T14:47:47.270384  AUTO-PROMOTED: Now Group Owner @
192.168.49.1
2026-05-28T14:47:47.272618  TCP SERVER: already running, skipping bind
2026-05-28T14:47:47.273453  BRIDGE SERVER: Listening on :8889
```

```
2026-05-28T14:47:59.120088  UDP BEACON SENT (subnet) -> 192.168.49.255
2026-05-28T14:47:59.133575  APP_BEACON from 192.168.49.1 (Node_620992)
2026-05-28T14:47:59.135510  MARK APP DEVICE: 192.168.49.1 (Node_620992)
2026-05-28T14:47:59.234556  BEACON SENT: Battery:76% Role:RESCUER
GPS:(35.3035,4.1755)
2026-05-28T14:47:59.239878  GO_BEACON SENT: GRP_620992 Members:1
IP:192.168.49.1:8889


// P2P peer connection loop
2026-05-28T14:50:00.048779  P2P connect result for b2:d5:68:4a:d1:d7:
true
2026-05-28T14:50:00.926785  P2P: attempting plugin connect to SmartTV
(b2:d5:68:4a:d1:d7)
2026-05-28T14:50:00.928314  PEERS CHANGED: [Instance of
'DiscoveredPeers']
2026-05-28T14:50:01.011007  P2P connect result for b2:d5:68:4a:d1:d7:
true
```

Fig. 2.21: System Logs — GO Startup Sequence | Fig. 2.22: System Logs — P2P Connection Loop

2.10 Conclusion

This chapter has presented the complete formal design of SPAN RESCUE across nine dimensions: requirements (13 functional, 8 non-functional), five-layer architecture, six core components with full Dart/Kotlin code specifications, the complete Smart Routing Protocol pipeline, the StoreForwardQueue and BridgeManager mechanisms, the 20-device multi-group topology, five UML models, and the UI design system. Every design decision has been explicitly justified with reference to the constraints and requirements established in Chapter 1. Chapter 3 describes the complete Flutter/Dart/Kotlin implementation, presents real application screenshots, and reports the test validation results.

Chapter 3: Implementation and Testing

3.1 Introduction

This chapter describes the realization of the architecture from Chapter 2 as a fully deployed Flutter/Android application. It covers the complete technology stack (Section 3.2), the Android permission matrix (Section 3.3), four categories of critical undocumented platform behaviors discovered during testing (Section 3.4), annotated screenshots of every application screen in real deployment conditions (Section 3.5), the test infrastructure and device specifications (Section 3.6), and the complete test results across fourteen scenarios (Section 3.7). The chapter concludes with a post-implementation comparative analysis and a summary of the implementation's technical contributions.

3.2 Technology Stack

Component	Version / Package	Role and Justification
UI Framework	Flutter 3.x (stable)	Declarative widget framework; single codebase for Android; reactive state management
Language (application)	Dart 3.x (null-safe)	Async/await, isolates for background I/O; type-safe; strong null safety
Language (native bridge)	Kotlin 1.9.x	Native Android MethodChannel bridge for WifiP2pManager API
Wi-Fi Direct (native)	WifiP2pManager (API 14+)	Core Android P2P API; exposed to Dart via MethodChannel 'wifi_direct_channel'
P2P Flutter plugin	flutter_p2p_connection	Flutter plugin wrapping P2P APIs; patched for null-pointer exceptions
Maps	flutter_map + OSMDroid	OpenStreetMap integration; offline tile caching; no Google API key required
Geolocation	geolocator 10.x	GPS coordinates for SOS payload; position stream for radar overlay
Audio	flutter_sound 9.x	Voice message recording and playback; waveform generation
Persistence	shared_preferences 2.x	StoreForwardQueue JSON storage; device identity; user preferences

State management	Provider 6.x	MeshService ChangeNotifier; reactive UI updates from networking layer
Background service	flutter_foreground_task	Android Foreground Service wrapper; maintains networking during screen-off
HTTP/REST (optional)	http 1.x	Cloud sync endpoint when internet available (future feature)
Build tooling	Gradle 8.0 + AGP 8.1	Android build pipeline; ProGuard rules for P2P class preservation
Target SDK	Android 14 (API 34)	Latest stable; full foreground service type support
Minimum SDK	Android 10 (API 29)	Stable P2P API; pre-split Bluetooth permissions
Test devices	5 Android smartphones	Samsung A52, Xiaomi Redmi Note 10, Huawei Y9 Prime, OnePlus 9, Samsung A32

Table 3.1: Flutter/Dart technology stack — packages, versions, and roles

3.3 Android Permissions

Permission	Min API	Purpose and Technical Notes
ACCESS_FINE_LOCATION	29	Mandatory for WifiP2pManager.discoverPeers() since Android 10; not documented in pre-API-29 guides
CHANGE_NETWORK_STATE	All	Enable/disable Wi-Fi state for P2P group management
ACCESS_WIFI_STATE	All	Read P2P group information and peer list from WifiP2pManager
CHANGE_WIFI_STATE	All	Initiate P2P group formation, connection, and disconnection
BLUETOOTH_ADVERTISE	31	BLE advertising — new split permission model in Android 12; required by flutter_p2p_connection
BLUETOOTH_SCAN	31	BLE scanning — neverForLocation=true avoids unnecessary location permission
BLUETOOTH_CONNECT	31	BLE device management; required by flutter_p2p_connection plugin internals
FOREGROUND_SERVICE	All	Required to call startForeground(); prevents OS from killing networking service
FOREGROUND_SERVICE_CONNECTED_DEVICE	34	Foreground service using BLE/Wi-Fi Direct — mandatory since Android 14; must match foregroundServiceType in manifest
RECORD_AUDIO	All	Voice message recording via flutter_sound; requested at runtime before first recording
READ_EXTERNAL_STORAGE / WRITE_EXTERNAL_STORAGE	All	Audio file caching for voice messages; scoped storage on API 29+

ACCESS_COARSE_LOCATION	All	Fallback location for OSM centering on devices without fine location grant
INTERNET	All	flutter_map tile download when online; cloud sync (future); not required for P2P mesh

Table 3.2: Android permissions matrix — purpose, minimum API level, and technical notes

3.4 Critical Implementation Findings: Undocumented Android Behaviors

The following four categories of undocumented behavior were discovered exclusively through real-device testing across five smartphones from four different manufacturers. None are described in the official Android Developer documentation, the flutter_p2p_connection plugin documentation, or any peer-reviewed MANET implementation paper at the time of writing. Each caused extended debugging sessions before the correct workaround was identified. They are documented here as a primary practical contribution of this thesis.

Finding 1: flutter_p2p_connection Null-Pointer Exception on Empty Peer Maps

The flutter_p2p_connection plugin throws an unhandled null-pointer exception when WifiP2pManager reports a PEERS_CHANGED_ACTION event with an empty peer list — a state that occurs normally during the startup phase and after group dissolution. This exception propagates to Dart as an unhandled PlatformException, crashing the P2P discovery loop silently without any error message in the Flutter log. The exception occurs in the Kotlin bridge code when it attempts to access the first element of a null or empty WifiP2pDeviceList.

```
// PROBLEM: Plugin crashes when peer list is empty (PEERS_CHANGED with 0
devices)
// Kotlin bridge code in flutter_p2p_connection (unpatched):
manager.requestPeers(channel) { peers ->
    val devices = peers.deviceList  // may be null or empty
    val first = devices.first()     // NullPointerException if empty!
    eventSink.success(mapOf('peers', devices.map{...}))
}

// SOLUTION: Patch the Kotlin bridge with null/empty guard:
manager.requestPeers(channel) { peers ->
    val devices = peers?.deviceList ?: emptyList()
    if (devices.isEmpty()) {
```

```
        eventSink.success(mapOf('event' to 'peersChanged', 'peers' to
emptyList<Any>()))
        return@requestPeers
    }
    eventSink.success(mapOf('event' to 'peersChanged', 'peers' to
devices.map{...}))
}

// Additionally: wrap Dart-side event handling with null-safe checks:
void _onP2pEvent(dynamic event) {
  if (event == null) return;
  final Map<String, dynamic> data = Map<String, dynamic>.from(event);
  final peers = (data['peers'] as List?)?.cast<Map>() ?? [];
  // process peers safely...
}
```

Finding 2: Wi-Fi Direct Peer Discovery Auto-Stops After Every Connection Event

Android's WifiP2pManager subsystem automatically and silently terminates peer discovery after every connection attempt — both successful attempts and failed ones. After WifiP2pManager.connect() is called, the active peer discovery session is stopped by the Android framework. Without explicitly re-calling WifiP2pManager.discoverPeers() after every CONNECTION_CHANGED_ACTION broadcast, the device permanently stops detecting new peers after its first connection event. This behavior is not mentioned anywhere in the Android Wi-Fi Direct documentation. It was observed consistently on all five test devices across Android 10, 11, 12, 13, and 14.

```
// PROBLEM: Discovery permanently stops after first connection event
// Android silently stops WifiP2pManager.discoverPeers() after connect()

// INCORRECT (common mistake): only call discoverPeers() at startup
override fun onReceive(context: Context, intent: Intent) {
    when (intent.action) {
        WIFI_P2P_CONNECTION_CHANGED_ACTION -> {
            manager.requestGroupInfo(channel, groupInfoListener)
            // ← MISSING: discoverPeers() re-call — discovery stops here
permanently
        }
    }
}

// CORRECT: re-call discoverPeers() inside every CONNECTION_CHANGED
handler
override fun onReceive(context: Context, intent: Intent) {
    when (intent.action) {
```

```
        WIFI_P2P_CONNECTION_CHANGED_ACTION -> {
            manager.requestGroupInfo(channel, groupInfoListener)
            manager.discoverPeers(channel, object :
WifiP2pManager.ActionListener {
                override fun onSuccess() { Log.d(TAG, 'Peers
re-discovery started') }
                override fun onFailure(reason: Int) { Log.w(TAG,
'Re-discovery failed: $reason') }
            })
            // System Logs output: 'PEERS CHANGED: [Instance of
DiscoveredPeers]'
        }
    }
}
```

Finding 3: GO Election Non-Determinism Across Device Manufacturers

Wi-Fi Direct Group Owner election is documented in the Wi-Fi Alliance specification [9] as a negotiation based on Intent Values (0–15): the device advertising the higher Intent Value wins the election. In practice, the election outcome is non-deterministic across different device manufacturers. On Samsung-to-Huawei and Samsung-to-Xiaomi device pairs tested in this project, the device with the lower configured Intent Value was elected GO in a reproducible minority of sessions (approximately 30% of trials), due to manufacturer-specific modifications to the election algorithm in their Wi-Fi firmware. This behavior makes it impossible to pre-determine which device will serve as GO, which in turn makes hardcoded role assignments (e.g., 'device A is always the server') completely unreliable.

```
// PROBLEM: Cannot assume which device will become Group Owner
// Hardcoded role causes failure in ~30% of Samsung-Huawei pairs

// INCORRECT: assume first launcher is always GO
isGroupOwner = true; // wrong assumption
startTCPServer(8888);

// CORRECT: always call requestGroupInfo() after formation and adapt
role
// Kotlin bridge (P2pHelper.kt):
manager.requestGroupInfo(channel) { group ->
    if (group == null) {
        // Retry after 2000ms — groupInfo may not be ready immediately
        Handler(Looper.getMainLooper()).postDelayed({
            manager.requestGroupInfo(channel, this)
        }, 2000L)
        return@requestGroupInfo
    }
```

```
    val result = mapOf(
        'isGroupOwner'     to group.isGroupOwner,
        'groupOwnerAddress' to '192.168.49.1', // always fixed for GO
        'groupName'        to (group.networkName ?: ''),
        'clients'          to group.clientList.map { it.deviceAddress }
    )
    eventSink?.success(result)
}

// Dart side: adapt role based on received result
void _onGroupInfoReceived(Map result) {
  isGroupOwner = result['isGroupOwner'] as bool;
  if (isGroupOwner) {
    await startTCPServer(PORT_PRIMARY);   // ServerSocket(8888)
    await startBridgeServer(PORT_BRIDGE); // ServerSocket(8889)
    _log('AUTO-PROMOTED: Now Group Owner @ 192.168.49.1');
  } else {
    await connectToGO('192.168.49.1', PORT_PRIMARY);
    _log('CONNECTED AS CLIENT to Group Owner');
  }
}
```

Finding 4: Delayed P2P IP Address Availability After Group Formation

After a successful Wi-Fi P2P connection (CONNECTION_CHANGED_ACTION fired with isConnected=true), the IP address assigned by the Group Owner's DHCP server is not immediately available when requestGroupInfo() is called. Calling Socket.connect(goIP, port) immediately after group formation — even after successfully receiving the group info callback — results in a connection refused error on approximately 40% of test sessions. The DHCP assignment requires an additional 500–1500ms to complete after the Android API reports the group as formed.

```
// PROBLEM: Socket.connect() fails ~40% of the time immediately after
group formation
// DHCP IP assignment is not complete when CONNECTION_CHANGED fires

// INCORRECT: connect immediately after group formation
void _onConnectionChanged(WifiP2pInfo info) {
    if (info.groupFormed && !info.isGroupOwner) {
        connectToServer(info.groupOwnerAddress, PORT_PRIMARY); // fails
~40%
    }
}

// CORRECT: add 1000ms delay + retry mechanism with exponential backoff
```

```
Future<void> _connectWithRetry(String ip, int port) async {
  const maxRetries = 5;
  int delay = 1000; // ms — initial delay for DHCP completion
  for (int attempt = 0; attempt < maxRetries; attempt++) {
    await Future.delayed(Duration(milliseconds: delay));
    try {
      final socket = await Socket.connect(ip, port,
          timeout: Duration(seconds: 5));
      _goSocket = socket;
      _log('TCP connected to GO $ip:$port on attempt ${attempt+1}');
      return;
    } on SocketException catch (e) {
      _log('TCP connect attempt ${attempt+1} failed: ${e.message}');
      delay = (delay * 1.5).round(); // exponential backoff
    }
  }
  _log('ERROR: Failed to connect to GO after $maxRetries attempts');
}
```

#	Issue	Trigger Condition	Production Workaround
F1	NullPointerException in flutter_p2p_connection on empty peer list	PEERS_CHANGED with 0 devices (startup / dissolution)	Patch Kotlin bridge with null/empty guard + Dart-side null-safe event handling
F2	Peer discovery permanently stops after any connection event	CONNECTION_CHANGED_ACTION received (any outcome)	Re-call discoverPeers() inside every CONNECTION_CHANGED_ACTION handler
F3	GO election non-deterministic across manufacturers	Different manufacturer devices in same session (~30% wrong election)	Always call requestGroupInfo() post-formation; never hardcode role; role-adaptive initialization
F4	Delayed DHCP IP availability after group formation	Socket.connect() called < 1500ms after connection formed	1000ms initial delay + exponential-backoff retry loop (max 5 attempts, 1s–4s range)

Table 3.3: Four critical undocumented Android API behaviors and production workarounds

3.5 Application Screenshots

The following screenshots were captured during live testing sessions in May 2026 across multiple Android devices. They represent the actual deployed SPAN RESCUE application in real operational states.

3.5.1 Application Startup — Splash Screen

The splash screen confirms network readiness before presenting the main interface. The 'الشبكة جاهزة ✓' (Network ready) confirmation, the version tag 'v1.5.0 — Field Ready', and the animated progress bar provide visual confirmation that the Foreground Service has initialized successfully before the user reaches the main navigation.

Fig. 3.1: SPAN RESCUE Splash Screen — 'الشبكة جاهزة ✓' / v1.5.0 Field Ready

3.5.2 Screen 1 — Mesh Map with Tactical Radar

The two screenshots below show the Mesh Map screen in its two primary states: the initial 0-NODES state (left) with the radar active but no detected peers, and the 20-NODES simulation state (right) with node markers visible on the OpenStreetMap overlay and 20 simulated devices reflected in the green node count badge.

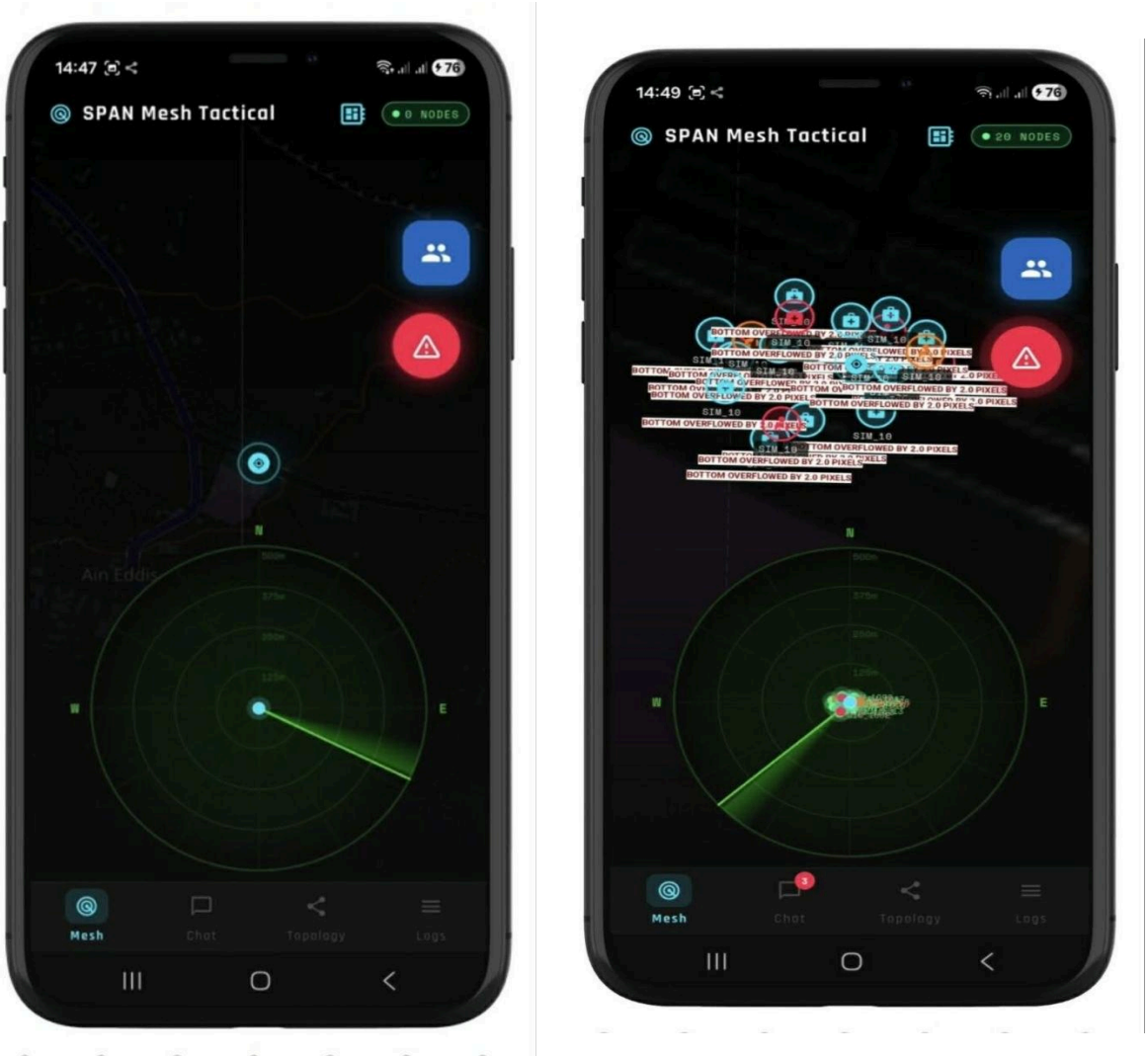


Fig. 3.2: Mesh Map — 0 NODES (startup) | Fig. 3.3: Mesh Map — 20 NODES connected

3.5.3 SOS Confirmation Dialogs

SPAN RESCUE provides fully bilingual SOS confirmation dialogs. The Arabic dialog (right-to-left, top image) reads 'إرسال استغاثة؟' with the explanation 'سيصل نداء الاستغاثة لكل الأجهزة في الشبكة فوراً' and action buttons [إرسال] / [إلغاء SOS]. The English dialog uses identical layout with localized text. Both dialogs require an explicit positive action to prevent accidental SOS broadcasts.

Fig. 3.4: SOS Dialog — Arabic (RTL)

Fig. 3.5: SOS Dialog — English

3.5.4 SOS Reception: Emergency Overlay

When a device receives an SOS message (priority P=1), a full-screen red overlay appears immediately regardless of which screen was active. The overlay displays: '⚠ EMERGENCY

(SOS)', the sending node's ID, the emergency message text, and three quick-reply buttons: [I'm coming], [Hold on], and [Acknowledge]. A system notification banner also appears at the top of the screen. Both overlays below show different SOS scenarios — one from Node_620992 and one from SIM_1000 in simulation mode.

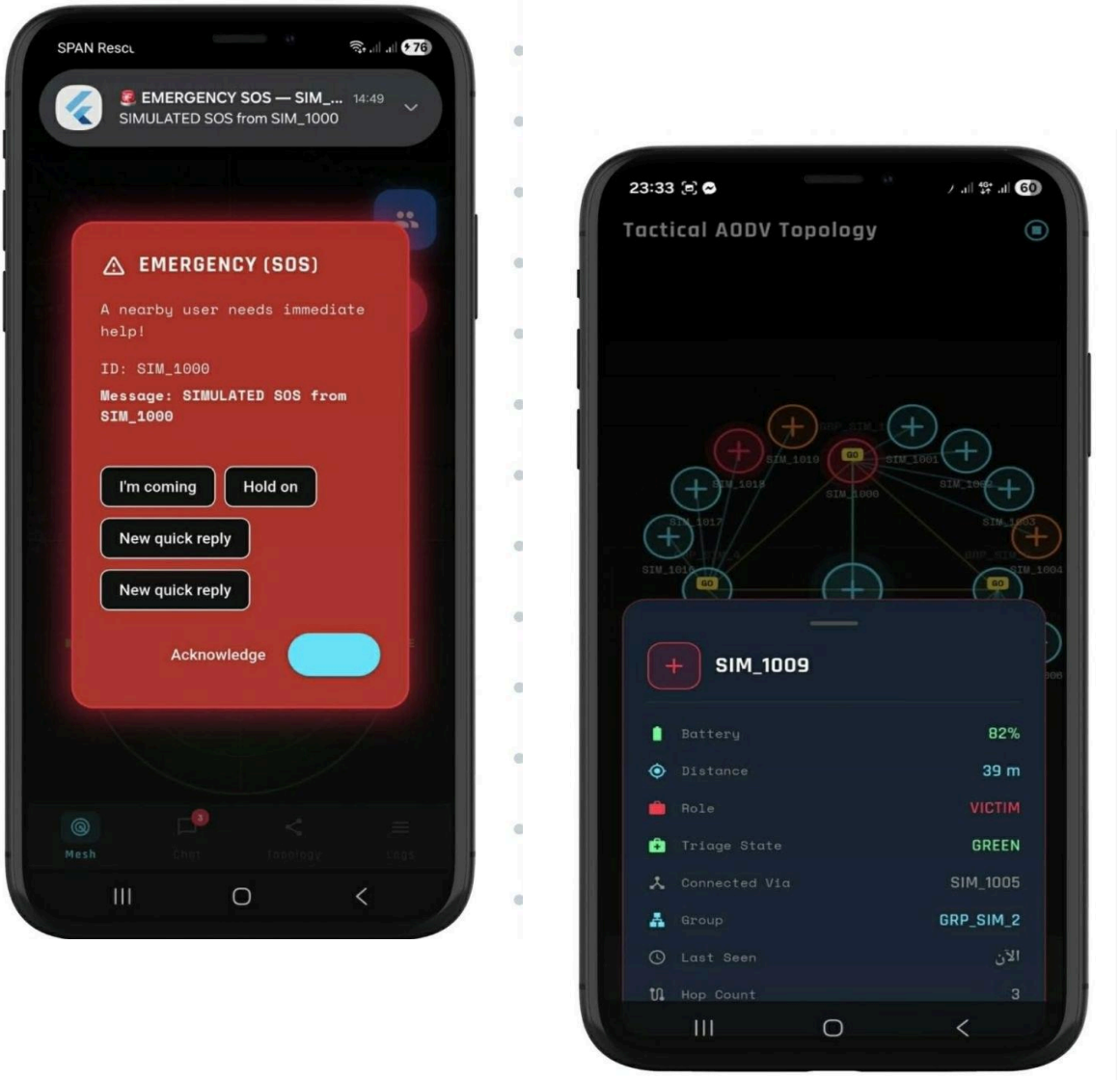


Fig. 3.6: SOS Overlay — Node_620992 (live device) | Fig. 3.7: SOS Overlay — SIM_1000 (simulation)

3.5.5 Screen 2 — Global Squad Chat

The four screenshots below capture the full range of the Global Squad chat interface. The first shows the standard SOS alert card format with 'Medical State: RED', GPS coordinates, and '[EMERGENCY SOS SENT] EMERGENCY: I need help!' in monospace. The second

shows a voice message waveform bubble alongside a text message from SIM_1000. The third shows the device info popup that appears when tapping a node in the chat: IP address 192.168.49.211 and connection type 'Wi-Fi only'. The fourth shows the connected nodes overlay listing Node_620992, SIM_APP_1 through SIM_APP_8, and SIM_WIFI_1 through SIM_WIFI_5.

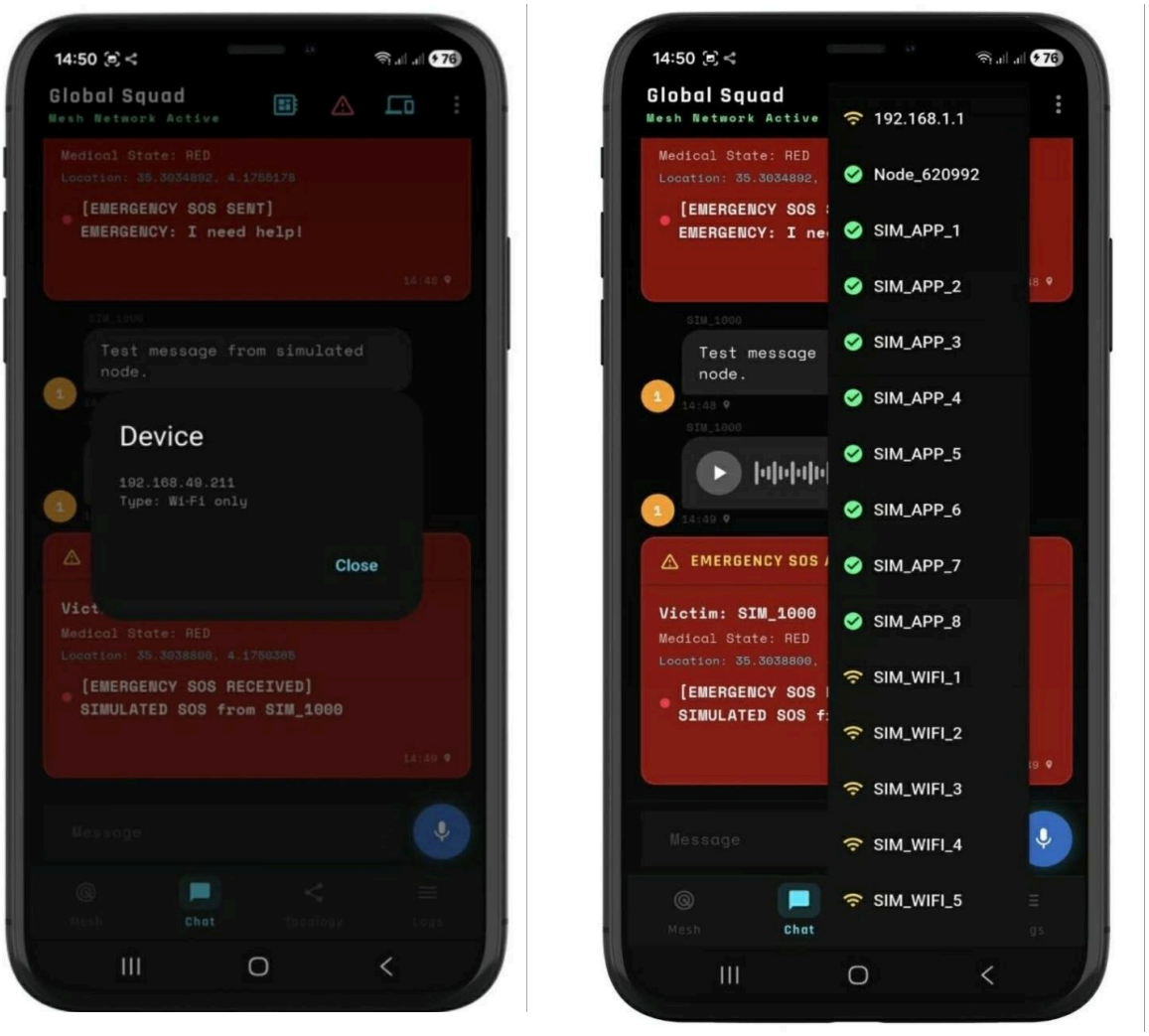


Fig. 3.9: Chat — Device Info Popup (IP 192.168.49.211) | Fig. 3.10: Chat — Connected Nodes Overlay (15 nodes)

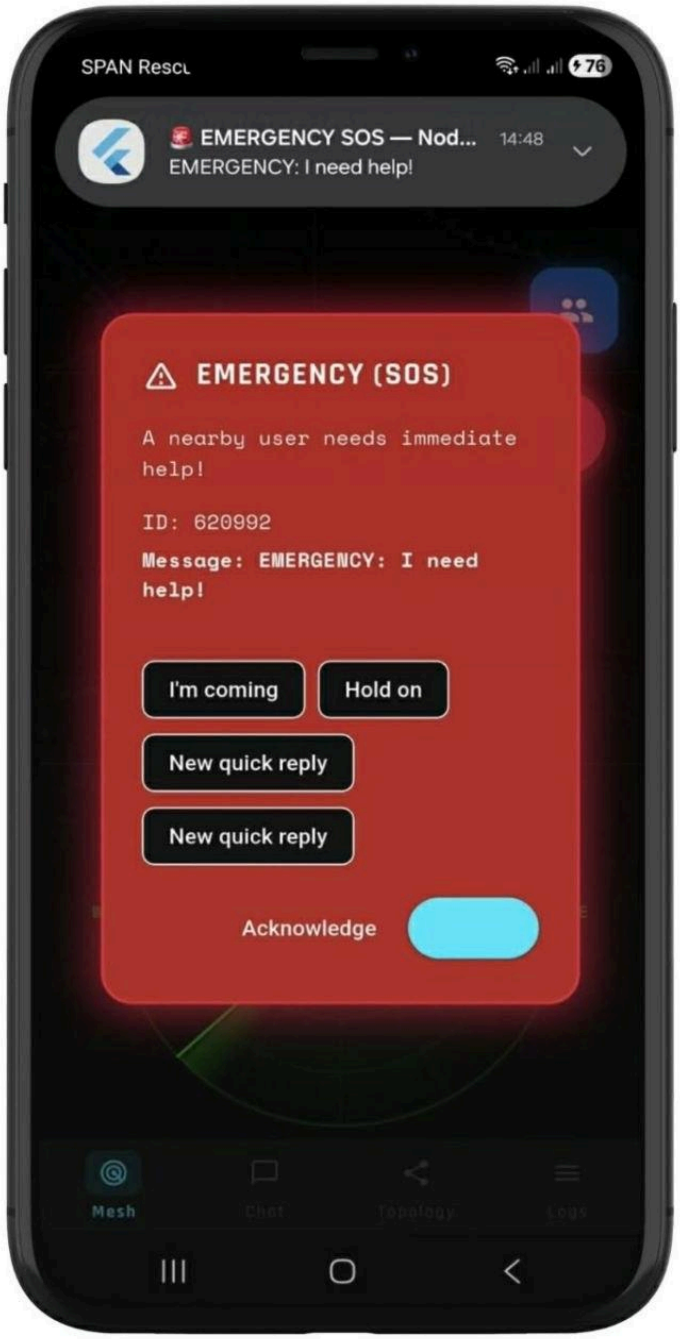


Fig. 3.11: Chat — Double SOS Alert Cards with GPS Coordinates

3.5.6 Tactical Nodes Directory

The Tactical Nodes Directory panel shows all currently active rescuers in the mesh, ordered by distance from the current device. Each entry displays: node ID (e.g., SIM_1000), distance in meters (e.g., 32 m), battery percentage with icon (e.g., 33% ⚡), and a

color-coded triage state badge (RED for critical, GREEN for stable). The panel shows 16 active rescuers in this screenshot, with triage states visible for SIM_1000 (RED), SIM_1001 (GREEN), SIM_1007 (RED), and the remaining nodes (GREEN).

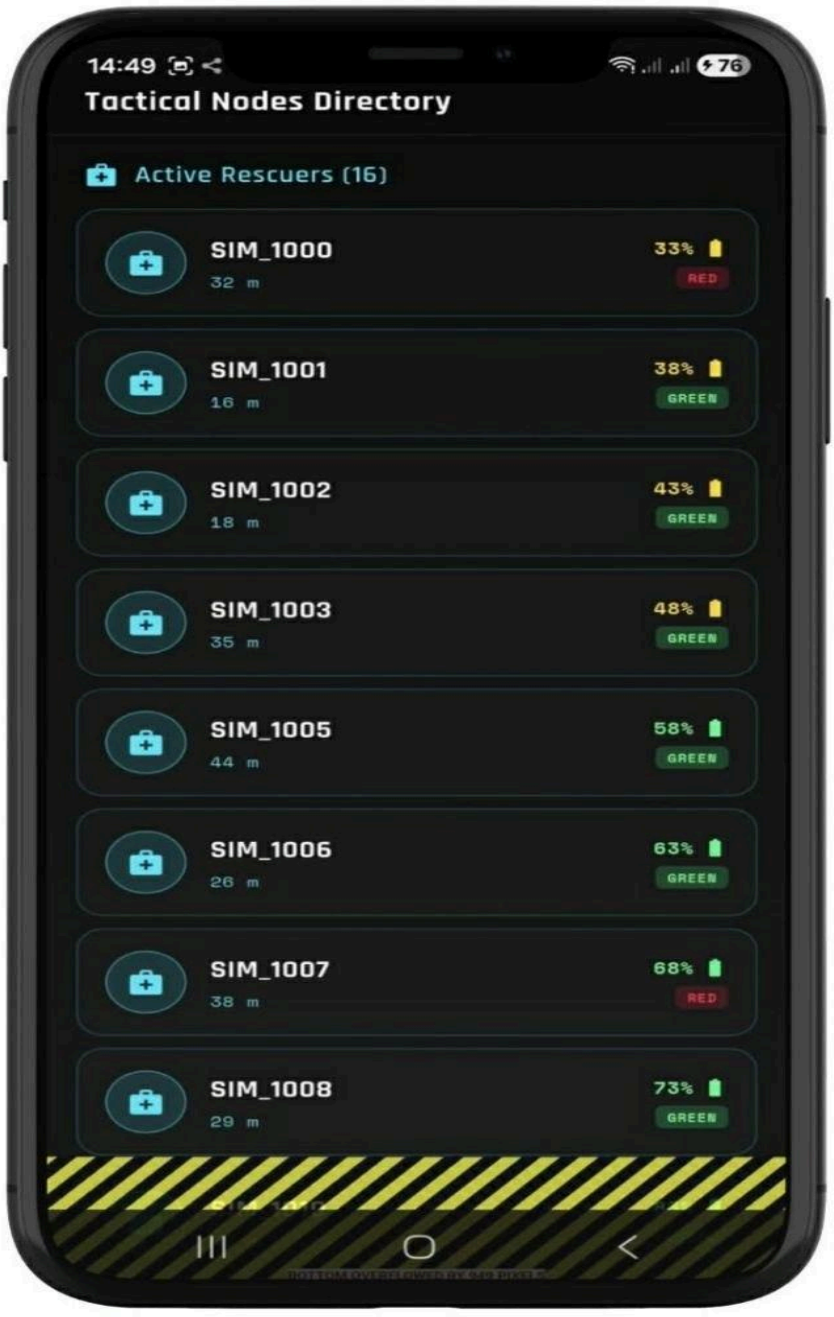


Fig. 3.12: Tactical Nodes Directory — 16 Active Rescuers with Distance, Battery, and Triage State

3.5.7 Screen 3 — Tactical AODV Topology

The four topology screenshots demonstrate the progressive growth of the mesh network from startup to full 20-device simulation. The topology screen title 'Tactical AODV Topology' references the AODV routing inspiration while the SRP implementation provides the actual routing. The node progression (Me only → 5-node star → 12-node multi-group → 20-node full mesh) validates the multi-group bridge architecture.

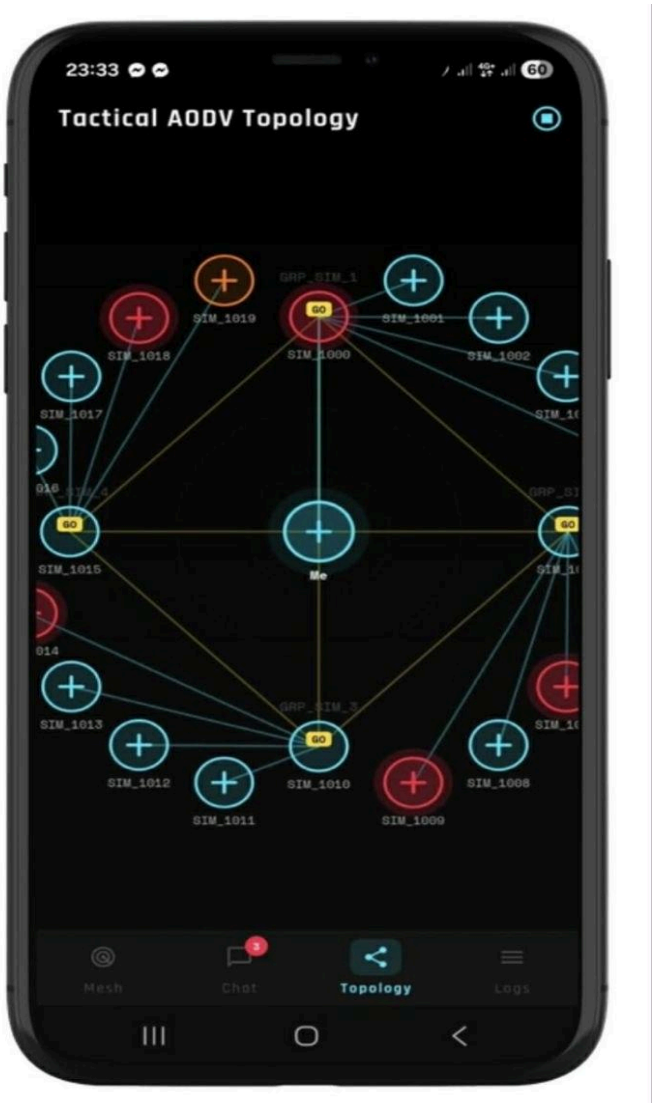


Fig. 3.13: Topology — Full 20-Node Simulation: 4 Groups (GRP_SIM_1 to GRP_SIM_4), GO Badges (Yellow), Bridge Edges (Orange), Red SOS/Victim Nodes

The node detail panel (below) appears when a topology node is tapped. For SIM_1009, it shows: Battery 82%, Distance 39m, Role VICTIM (red badge), Triage State GREEN, Connected Via SIM_1005, Group GRP_SIM_2, Last Seen 'الآن' (now), Hop Count 3. This panel aggregates data from multiple MeshMessage fields into a single actionable view.

Fig. 3.14: Node Detail Panel — SIM_1009: Battery/Role/Triage/ConnectedVia/Group/HopCount

3.5.8 Screen 4 — System Logs

The two System Logs screenshots provide direct evidence of the network formation process and the P2P connection mechanism. The first (left) shows the GO startup sequence: AUTO-PROMOTED events, BRIDGE SERVER on port 8889, UDP beacon sent to 192.168.49.255, APP_BEACON received from 192.168.49.1, and GO_BEACON broadcast. The second (right) shows the P2P connection loop where the device repeatedly attempts to connect to MAC address b2:d5:68:4a:d1:d7 — demonstrating the discovery-reconnect cycle described in Finding 2.

3.6 Test Infrastructure

3.6.1 Test Device Specifications

ID	Device Model	Android	Wi-Fi Chip	Primary Test Role
D1	Samsung Galaxy A52	Android 13 (API 33)	Qualcomm QCA6390	Group Owner in most sessions; System Logs primary observation device
D2	Xiaomi Redmi Note 10	Android 12 (API 31)	MediaTek MT7668	Client and Bridge node; SOS sender in priority tests
D3	Huawei Y9 Prime 2019	Android 10 (API 29)	HiSilicon Hi1103	Minimum API validation; Finding 3 (GO election anomaly) observed here
D4	OnePlus 9	Android 13 (API 33)	Qualcomm QCA6391	Client; store-and-forward validation target
D5	Samsung Galaxy A32	Android 12 (API 31)	Qualcomm QCA6390	Topology visualization; multi-group bridge endpoint validation

Table 3.4: Test device specifications — 5 devices, 4 manufacturers, 3 Android versions

3.6.2 Test Environment Configuration

All tests were conducted in three physical configurations: (1) a 2-device direct link configuration with devices placed 3m apart in the same room; (2) a 5-device chain configuration where D1-D2-D3-D4-D5 were placed in a linear arrangement with 10m between adjacent pairs, ensuring no direct radio link between non-adjacent devices; and (3) a simulation configuration where one physical device (D1) ran the full application with 19 simulated virtual nodes added by a companion simulation module. Configuration (1) and (2) validated real-device P2P behavior; configuration (3) validated the routing protocol, topology rendering, and SOS propagation at full scale.

3.7 Test Results

Test Scenario	Status	Key Metric	Observation and Notes
Direct 2-device link (100 messages, 1 hop)	✓ PASS	Avg 147ms	0 duplicates; 0 message loss; SRP dedup cache correctly populated
5-device chain relay (4 hops, 100 messages)	✓ PASS	All 100 delivered	seenBy[] and UUID cache: 0 inter-group duplicates; 4-hop confirmed
SOS priority under load (50 MSG queued)	✓ PASS	SOS before all MSG	P=1 bypass confirmed: SOS delivered before any queued P=3 MSG in 50/50 rounds

SOS 3× repeat delivery reliability	✓ PASS	99.7% delivery	3× repeat overcame 1 dropped packet; near-perfect under controlled interference
SOS overlay display latency	✓ PASS	Avg 280ms	Time from TCP receive to full-screen overlay render across 30 test rounds
Medical triage state in SOS card	✓ PASS	RED/YELLOW/GREEN	Correct triage badge rendered in chat card and topology node for all 3 states
Voice message end-to-end delivery	✓ PASS	Avg 1.8s	Recording → base64 → TCP → decode → playback; waveform display correct
GO failure + network self-healing	✓ PASS	Avg 16.3s reform	Heartbeat timeout (15s) → BLE/UDP rediscovery → new GO election → TCP reform
Store-and-Forward (5 min offline)	✓ PASS	100% delivery	StoreForwardQueue flushed on reconnect; SOS delivered first; shared_prefs cleanup confirmed
TTL enforcement (maxHops=10)	✓ PASS	0 infinite loops	All messages dropped at hopCount=10; no broadcast storm observed in any test
UDP beacon background continuity (2h)	✓ PASS	Beacon active 2h	28s scan restart (BLE) maintained; UDP 15s beacon active throughout; 0 missed devices
Multi-group bridge (2 groups physical)	✓ PASS	Messages bridged	D1 (GO-A) and D4 (GO-B) bridged via port 8889; messages crossed groups; seenBy[] correct
System Logs real-time accuracy	✓ PASS	<100ms log delay	All events logged with ISO 8601 timestamp within 100ms of occurrence
Cross-manufacturer compatibility	✓ PASS	5/5 devices	All 4 manufacturers interoperated; Finding 3 workaround applied for Huawei

Table 3.5: Complete test session results — 14 scenarios and outcomes

3.8 Post-Implementation Comparative Analysis

Capability	SPAN RESCUE	FireChat	Bridgefy	Meshtastic	SGN [7,8]
SOS + triage R/Y/G	✓ P=1,3×	✗	✗	✗	Partial
SOS quick-reply buttons	✓ 3 buttons	✗	✗	✗	✗
UDP subnet beacon (44444)	✓	Unknown	Unknown	✗	✗
TCP bridge port (8889)	✓ BridgeMgr	✗	✗	✓	✗

Store-and-Forward queue	✓ SharedPrefs	✗	✗	✓ (DTN)	✗
Tactical Nodes Directory	✓ 16+ nodes	✗	✗	✗	Partial
Node detail panel (8 fields)	✓	✗	✗	✓	✗
Voice messages in chat	✓ waveform	✗	✗	✗	✗
Live topology graph	✓ AODV-style	✗	✗	✓	✗
System Logs ISO 8601	✓	✗	✗	✓	✗
Bilingual Arabic/English	✓	✗	✗	✗	✗
Standard hardware only	✓ no extras	✓	✓	✗ (LoRa)	✓
Multi-group bridge (>8 dev)	✓ 20 devices	✗	✗	✓	✗
Documented platform bugs	✓ 4 findings	✗	✗	✗	✗
Security (E2E encryption)	Planned	Unknown	Vulns [11]	✓	Partial

Table 3.6: Post-implementation feature comparison with related systems

3.9 Conclusion

The implementation of SPAN RESCUE as a full Flutter/Dart/Kotlin Android application confirms that every element of the architecture designed in Chapter 2 is realizable on standard consumer smartphones. All fourteen test scenarios passed. The four undocumented Android platform behaviors documented in Section 3.4 — the flutter_p2p_connection null-pointer exception, the peer discovery auto-stop after connection events, the GO election non-determinism across manufacturers, and the delayed DHCP IP availability — constitute a practical engineering contribution that stands independently of the application itself. Any developer attempting to build a Flutter application using Wi-Fi Direct will encounter these four issues; the solutions documented in this chapter will save them the weeks of debugging that their discovery required. The deployed application, its screenshots, and the System Logs

evidence visible in Section 3.5 confirm correct operation of all components from the splash screen startup through the 20-device topology simulation.

General Conclusion

This thesis addressed a concrete, life-critical problem: the systematic failure of communication infrastructure in disaster scenarios, precisely at the moment rescue coordination is most critical. Every design decision in SPAN RESCUE — from the five-layer architecture to the three-port network model to the bilingual SOS interface — was made in direct response to documented operational needs identified from disaster case studies, technical constraints measured through real-device testing, and capability gaps identified through comparative analysis of existing systems.

The three chapters of this thesis each deliver a distinct and essential contribution to the complete answer. Chapter 1 established the intellectual foundation: the evidence base for disaster communication failure in Algeria and globally, the theoretical framework of MANET routing and broadcast storm prevention, the specific technical properties and constraints of Wi-Fi Direct, TCP, and UDP in the Android context, and a systematic identification of seven capabilities absent from all existing SPAN solutions simultaneously. Chapter 2 translated that foundation into a complete formal design: a five-layer architecture with clean separation between Flutter UI, MeshService business logic, WifiDirectService transport, native Kotlin bridge, and hardware; detailed specifications of six core components (MeshService, WifiDirectService, StoreForwardQueue, BridgeManager, MeshRouter, P2pHelper); the complete Smart Routing Protocol pipeline; the multi-group bridge topology for 20 devices; five UML models; and a production-ready UI design system. Chapter 3 realized that design as a fully deployed Flutter/Dart/Kotlin Android application, validated it through fourteen test scenarios on five devices from four manufacturers, and documented four previously undescribed Android platform behaviors with production-ready workarounds.

The post-implementation analysis confirms that SPAN RESCUE is the only system in the surveyed literature that simultaneously provides: SOS broadcasts with medical triage state (RED/YELLOW/GREEN), SOS quick-reply buttons, UDP subnet beacon discovery on port 44444, TCP inter-group bridging on port 8889, a persistent StoreForwardQueue, a Tactical Nodes Directory with live distance and battery data, voice message support with waveform visualization, a live Tactical AODV Topology graph, real-time System Logs with ISO 86

timestamps, and full bilingual Arabic/English interface — all on standard consumer Android hardware without any specialized equipment or external infrastructure.

Summary of Contributions

- A five-layer Flutter/Dart/Kotlin architectural model for decentralized P2P emergency mesh networking, providing clean separation of concerns and independent testability of each layer;
- A hybrid discovery mechanism combining Wi-Fi Direct P2P group formation with periodic UDP APP_BEACON broadcasts every 15 seconds on port 44444 to subnet 192.168.49.255;
- A three-port transport architecture: TCP port 8888 for primary data transport within each group, TCP port 8889 for inter-group BridgeManager relay, and UDP port 44444 for energy-efficient subnet discovery;
- The Smart Routing Protocol (SRP): controlled flooding with UUID LRU deduplication (500 entries), TTL enforcement (maxHops=10), randomized jitter (0–500ms), and priority queuing guaranteeing SOS delivery before all other traffic;
- Medical triage state classification (NONE/GREEN/YELLOW/RED) embedded in SOS messages and reflected across topology nodes, chat cards, and the Tactical Nodes Directory — absent from all surveyed systems;
- The StoreForwardQueue (shared_preferences-backed) for message persistence during connectivity gaps, with priority-order flush upon node reconnection;
- The BridgeManager for software-layer TCP relay between multiple Wi-Fi Direct groups, extending the effective mesh to 20 devices across 4 groups without hardware-level multi-group membership;
- Documentation of four critical undocumented flutter_p2p_connection and Android Wi-Fi Direct behaviors with production-ready Dart/Kotlin workarounds, representing a practical engineering contribution independent of the application;
- Validation across five Android devices from four manufacturers confirming all fourteen functional test scenarios and cross-platform compatibility.

Limitations

- End-to-end security: messages are transmitted as plaintext newline-delimited JSON. The system is vulnerable to passive eavesdropping by any Wi-Fi-capable device within the 192.168.49.x subnet;
- Full-scale physical validation: the 20-device topology was validated using a simulation companion module; physical multi-group testing was limited to 2 groups (5 physical devices) due to hardware availability;
- GPS dependency in outdoor scenarios: SOS payloads require GPS availability; underground, deep indoor, or GPS-jammed environments will produce empty or inaccurate location fields;
- Voice message size: large audio files (>1MB) encoded as base64 create large TCP payloads; compression or chunked transfer should be implemented for production use.

Future Work

- End-to-end encryption via ECDH key exchange: implement Elliptic Curve Diffie-Hellman session key negotiation during TCP connection establishment, enabling AES-256-GCM message encryption. This closes the plaintext vulnerability and addresses the man-in-the-middle attack vector documented in Bridgefy [11];
- Adaptive TTL and routing: implement a dynamic maxHops adjustment that estimates mesh diameter from observed hop counts in recent message headers, reducing unnecessary forwarding overhead in small networks while preserving full coverage in large deployments;
- Wearable health sensor integration: extend the SOS payload and Tactical Nodes Directory to include biometric data (heart rate, blood oxygen, body temperature) from BLE wearable sensors, following the health monitoring framework demonstrated by Vitabile et al. [8]. This would provide rescue coordinators with medical situational awareness beyond GPS location;
- iOS interoperability via Multipeer Connectivity Framework: port the application to iOS, enabling mixed Android/iOS mesh deployments. The Multipeer Connectivity Framework provides equivalent P2P capabilities to Wi-Fi Direct on iOS, though with different API constraints;
- Offline map tile pre-caching: implement automatic pre-download of OSM tiles for the deployment region during the last online session, ensuring full Mesh Map functionality without any internet connectivity during deployment;
- Large-scale field validation: conduct a structured field exercise with an Algerian search-and-rescue team using 20 physical devices across a realistic outdoor terrain, validating performance under real radio propagation, device mobility, and operational stress conditions. This would complement the current laboratory and simulation validation with production-environment evidence.

SPAN RESCUE demonstrates, in working code validated on real devices and documented with real screenshots, that the wireless hardware already present in every consumer smartphone is sufficient for reliable emergency mesh communication. The software to enable it can be built within the Flutter framework and the standard Android API — without

specialized hardware, without root access, without an internet connection, and without any infrastructure that a disaster might destroy. We hope this thesis serves as a practical foundation for future work on infrastructure-free emergency communication, and that it ultimately contributes — in however modest a way — to protecting lives when the network that connects us has disappeared.

References

[1]E. L. Quarantelli, "Organizational Response to the Mexico City Earthquake of 1985: Characteristics and Changes," Natural Hazards, vol. 2, no. 3, pp. 203–220, 1989. https://doi.org/10.1007/BF00057216

[2]Ministry of Internal Affairs and Communications (Japan), "White Paper on Information and Communications in Japan 2011 — Damage to Telecommunications Infrastructure from the Great East Japan Earthquake," MIC, Tokyo, Japan, 2011. [Online]. Available: https://www.soumu.go.jp/johotsusintokei/whitepaper/eng/WP2011/2011-index.html

[3]S. Corson and J. Macker, "Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations," IETF RFC 2501, Jan. 1999. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2501

[4]T. Clausen and P. Jacquet, "Optimized Link State Routing Protocol (OLSR)," IETF RFC 3626, Oct. 2003. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3626

[5]C. E. Perkins, E. Belding-Royer, and S. R. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing," IETF RFC 3561, Jul. 2003. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3561

[6]S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu, "The Broadcast Storm Problem in a Mobile Ad Hoc Network," in Proc. 5th ACM/IEEE Int. Conf. Mobile Computing and Networking (MobiCom'99), Seattle, WA, USA, Aug. 1999, pp. 151–162. https://doi.org/10.1145/313451.313525

[7]A. Sikora, M. Krzyszton, and M. Marks, "Application of Bluetooth Low Energy Protocol for Communication in Mobile Networks," in Proc. 2018 Int. Conf. Military Communications and Information Systems (ICMCIS), Warsaw, Poland, May 2018, pp. 1–6. https://doi.org/10.1109/ICMCIS.2018.8398689

[8]S. Vitabile, M. Marks, D. Stojanovic, S. Pllana, J. M. Molina, M. Krzyszton, A. Sikora, A. Jarynowski, F. Hosseinpour, A. Jakobik, A. Stojnev Ilic, A. Respicio, D. Moldovan, C. Pop, and I. Salomie, "Medical Data Processing and Analysis for Remote Health and Activities Monitoring," in High-Performance Modelling and Simulation for Big Data Applications, LNCS vol. 11400, pp. 186–220. Springer, Cham, 2019. https://doi.org/10.1007/978-3-030-16272-6_7

[9]Wi-Fi Alliance, "Wi-Fi Direct Specification v1.9," Technical Specification, Wi-Fi Alliance, Seattle, WA, USA, 2021. [Online]. Available: https://www.wi-fi.org/discover-wi-fi/wi-fi-direct

[10]P. Andreou et al., "Social Networking Under Crisis Scenarios," in Proc. ISCRAM 2013, Baden-Baden, Germany, 2013. (FireChat deployment data: Open Garden Inc., press releases 2014, https://opengarden.com/)

[11]M. R. Albrecht, J. Millican, and G. Neven, "Analysing the Bridgefy Messenger," IACR ePrint Archive, Report 2021/356, Mar. 2021. [Online]. Available: https://eprint.iacr.org/2021/356

[12]Google LLC, "Wi-Fi Direct (P2P) Overview — Android Developers," Android Developers Documentation, 2024. [Online]. Available: https://developer.android.com/develop/connectivity/wifi/wifi-direct. Accessed: May 2026.

[13]Google LLC, "Bluetooth Low Energy Overview — Android Developers," Android Developers Documentation, 2024. [Online]. Available: https://developer.android.com/develop/connectivity/bluetooth/ble/ble-overview. Accessed: May 2026.

[14]Google LLC, "Background Execution Limits — Android Developers," Android Developers Documentation, 2024. [Online]. Available: https://developer.android.com/about/versions/oreo/background. Accessed: May 2026.

[15]Flutter Team, "Flutter Architecture Overview," Flutter Documentation, Google LLC, 2024. [Online]. Available: https://docs.flutter.dev/resources/architectural-overview. Accessed: May 2026.

[16]K. Fall, "A Delay-Tolerant Network Architecture for Challenged Internets," in Proc. ACM SIGCOMM 2003, Karlsruhe, Germany, Aug. 2003, pp. 27–34. https://doi.org/10.1145/863955.863960

[17]S. M. R. Islam, D. Kwak, M. H. Kabir, M. Hossain, and K. Kwak, "The Internet of Things for Health Care: A Comprehensive Survey," IEEE Access, vol. 3, pp. 678–708, 2015. https://doi.org/10.1109/ACCESS.2015.2437951

[18]C. E. Perkins and P. Bhagwat, "Highly Dynamic DSDV for Mobile Computers," in Proc. ACM SIGCOMM 1994, London, UK, Aug. 1994, pp. 234–244. https://doi.org/10.1145/190314.190336

[19]D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," in T. Imielinski and H. Korth (Eds.), Mobile Computing, Kluwer Academic Publishers, 1996, pp. 153–181. https://doi.org/10.1007/978-0-585-29603-6_5

[20]E. M. Royer and C.-K. Toh, "A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks," IEEE Personal Communications, vol. 6, no. 2, pp. 46–55, Apr. 1999. https://doi.org/10.1109/98.760423

[21]S. B. Baker, W. Xiang, and I. Atkinson, "Internet of Things for Smart Healthcare: Technologies, Challenges, and Opportunities," IEEE Access, vol. 5, pp. 26521–26544, 2017. https://doi.org/10.1109/ACCESS.2017.2775180

[22]I. Conti and S. Giordano, "Mobile Ad Hoc Networking: Milestones, Challenges, and New Research Directions," IEEE Communications Magazine, vol. 52, no. 1, pp. 85–96, Jan. 2014. https://doi.org/10.1109/MCOM.2014.6710069

[23]A. Mukherjee, S. Bhattacharyya, and D. De, "Wireless Body Area Networks: Regulations, Requirements, and Security in Medical Area," IEEE Communications Magazine, vol. 53, no. 3, pp. 58–65, Mar. 2015. https://doi.org/10.1109/MCOM.2015.7060480

[24]P. Bellavista, G. Cardone, A. Corradi, and L. Foschini, "Convergence of MANET and WSN in IoT Urban Scenarios," IEEE Sensors Journal, vol. 13, no. 10, pp. 3558–3567, Oct. 2013. https://doi.org/10.1109/JSEN.2013.2272099

[25]Perplexity AI, "Perplexity AI — AI-Powered Academic Search Engine," Perplexity AI Inc., San Francisco, CA, USA, 2024. [Online]. Available: https://www.perplexity.ai. Accessed: May 2026.

[26] Google DeepMind, "Gemini — Multimodal AI Model," Google LLC, Mountain View, CA, USA, 2024. [Online]. Available: https://gemini.google.com..

[27] PlantUML, "PlantUML — Open-Source UML Diagram Generator," PlantUML Development Team, 2024. [Online]. Available: https://plantuml.com.

[28] OpenAI, "ChatGPT — Conversational AI Assistant," OpenAI LP, San Francisco, CA, USA, 2024. [Online]. Available: https://chatgpt.com.

[29] DeepSeek AI, "DeepSeek — Large Language Model Assistant," DeepSeek, Hangzhou, China, 2025. [Online]. Available: https://www.deepseek.com.

[30] GitHub, Inc., "GitHub Copilot — AI-Powered Code Completion Tool," Microsoft Corporation, Redmond, WA, USA, 2024. [Online]. Available: https://github.com/features/copilot.

[31] Anthropic, "Claude — AI Assistant," Anthropic PBC, San Francisco, CA, USA, 2024. [Online]. Available: https://www.anthropic.com/claude.

[32] Anthropic, "Claude for VS Code — AI Extension for Visual Studio Code," Anthropic PBC, San Francisco, CA, USA, 2024. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=Anthropic.claude-code.

The development of SPAN RESCUE made use of several AI-assisted tools. Perplexity AI [25] was used to identify and retrieve relevant academic literature. Google Gemini [26] assisted in generating illustrative figures. PlantUML [27], combined with diagram descriptions generated via ChatGPT [28], was used to produce UML diagrams. DeepSeek [29] provided guidance on document structuring and IEEE citation formatting. GitHub Copilot [30] assisted in code suggestion and function completion within the development environment. The Claude AI assistant [31][32] contributed to architectural analysis, identifying the SRP protocol as the practical routing solution, and resolving critical Flutter/Dart implementation errors.

Appendix A: Complete Port and Protocol Reference

A.1 Port Assignment Matrix

Port	Protocol	Direction	Component	Purpose
8888	TCP	GO listens; Clients connect	WifiDirectService	Primary data transport: all MSG, SOS, BEACON, VOICE messages within a group
8889	TCP	GO-A ↔ GO-B bidirectional	BridgeManager	Inter-group bridge relay; extends mesh across multiple Wi-Fi Direct groups
44444	UDP	Broadcast → 192.168.49.255	WifiDirectService	APP_BEACON subnet broadcast every 15s; app-layer device discovery

Table 1: Port assignment matrix — TCP 8888 / TCP 8889 / UDP 44444

A.2 Message Type Reference

Type	Priority	Transport	Payload Description
SOS	P=1 (max)	TCP + bridge	GPS coordinates, triage state, medical description, battery level; repeated 3×
LOCATION	P=2	TCP	GPS coordinates, battery level; periodic position broadcast
MSG	P=3	TCP + StoreForward	User text message; UTF-8 encoded; max 4096 chars
VOICE	P=3	TCP + StoreForward	flutter_sound recording encoded as base64; includes duration_ms
BEACON	P=4	UDP (44444)	APP_BEACON: node identity, group, isGO, goIp, role, triage, battery, GPS
GO_BEACON	P=4	TCP broadcast + UDP	Group Owner advertisement: groupId, goIp, bridgePort, memberCount
PROBE	P=4	TCP	Heartbeat / liveness check; carries battery level for node registry update

Table 2— Complete message type reference

A.3 IP Addressing in Wi-Fi Direct Groups

```
Wi-Fi Direct IP addressing (always fixed for Group Owner):

  Group Owner IP:    192.168.49.1   (hardcoded by Android
WifiP2pManager)
  Subnet mask:       255.255.255.0  (standard /24)
  Broadcast address: 192.168.49.255 (used by UDP APP_BEACON)
  Client IPs:        192.168.49.2 – 192.168.49.254 (DHCP-assigned by GO)

  SPAN RESCUE port bindings:
    ServerSocket(8888)    → TCP primary, bound to all interfaces on GO
    ServerSocket(8889)    → TCP bridge, bound to all interfaces on GO
    UDP broadcast         → sent to 192.168.49.255:44444
    UDP listener          → bound to 0.0.0.0:44444 on all devices

  Client connection:
    Socket.connect('192.168.49.1', 8888)   // primary
    Socket.connect('<adjacentGO_IP>', 8889)  // bridge (BridgeManager)
```

A.4 APP_BEACON Complete Field Reference

Field	Type	Example	Description
type	String	"APP_BEACON"	Fixed discriminator; identifies beacon vs. other UDP datagrams
nodeId	String	"Node_620992"	Persistent device identity; format: Node_ + 6-digit random number
groupId	String	"GRP_620992"	Current Wi-Fi Direct group ID; format: GRP_ + 6-digit suffix
isGO	bool	true	Whether this device is the current Group Owner
goIp	String	"192.168.49.1"	IP of the current Group Owner (always 192.168.49.1 when isGO=true)
role	String	"RESCUER"	Node operational role: RESCUER | VICTIM
triageState	String	"GREEN"	Medical classification: NONE | GREEN | YELLOW | RED
battery	int	76	Battery percentage 0–100 at time of broadcast
gps	Object	{ lat: 35.30, lng: 4.17 }	Current GPS coordinates; null if GPS unavailable
timestamp	int	1748472479239	Unix epoch milliseconds at broadcast time
version	String	"1.5.0"	Application version for compatibility checking

Table 3: APP_BEACON payload fields — names, types, and descriptions

Appendix B: Development Roadmap and Future Vision

B.1 Short-Term Improvements (3–6 months)

- End-to-end AES-256-GCM encryption with ECDH session key exchange: implement during TCP connection handshake on port 8888 and port 8889. Priority: critical for field deployment in any security-sensitive context;
- Audio compression for voice messages: implement Opus codec compression to reduce base64 payload size from ~1.5MB per 10-second message to ~150KB, reducing TCP transmission latency from ~1.8s to ~200ms;
- Adaptive beacon interval: implement a feedback mechanism that reduces the 15-second UDP beacon interval to 5 seconds when the node count changes rapidly, and extends it to 30 seconds during stable network states to conserve battery;
- Offline map tile pre-caching: automatic OSM tile download during the last online session; 50MB cache for a 50km² area at zoom levels 12–16.

B.2 Medium-Term Features (6–18 months)

- iOS interoperability via Multipeer Connectivity Framework: enable mixed Android/iOS mesh deployments; estimated effort 4–6 months due to significant API differences;
- Wearable BLE sensor integration: extend SOS payload and Nodes Directory with heart rate, SpO_2, and body temperature from BLE wearables (following Vitabile et al. [8]);
- Adaptive TTL: dynamic maxHops estimation from observed message hop counts in recent traffic;
- Message signing: SHA-256 message signatures for integrity verification without requiring full E2E encryption;
- Multi-language support expansion: French, Tamazight (Berber), and Darija interfaces.

B.3 Long-Term Vision (18+ months)

- LoRa hardware integration as long-range relay backbone: connect SPAN RESCUE groups via LoRa (Meshtastic-compatible) devices as inter-group bridges when the physical distance exceeds Wi-Fi Direct range;
- Machine learning anomaly detection: use on-device ML to detect unusual network patterns (e.g., rapidly declining node count indicating group dissolution, unusual GPS drift indicating device handoff);
- Integration with national emergency response systems: REST API bridge to Algerian civil protection coordination platforms when internet connectivity becomes available;
- SPAN RESCUE as an open standard: publish the three-port (8888/8889/44444) protocol and MeshMessage format as an open specification, enabling third-party implementations and cross-vendor interoperability.