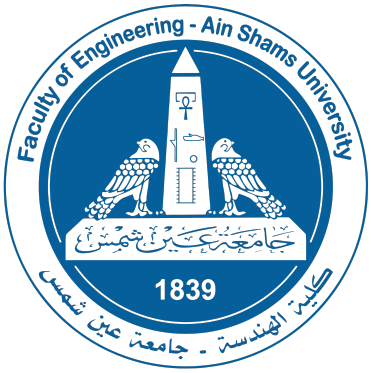


BEACON

Emergency Peer-to-Peer Communication System

Technical Documentation Report

A Flutter-based Offline Communication Platform for Disaster Response

Leveraging Wi-Fi Direct and BLE Technologies for Infrastructure-Free Emergency Coordination

Supervised By:

Dr. Haytham Azmi
Eng. Ahmed Wael
Eng.A'laa Hamdy

Team Members:

Name	Student ID
Gaser Zaghloul Hassan	21P0052
Ahmad Muhammad Abdelmaksoud	2101077
Mohamed Hossam Abdelalim	21P0254
Abdulrahman sherif abdelaziz	21P0098
Adham Osama Mohamed Nour	22P0071

December 2025

Table of Contents

Table of Figures

BEACON — Technical Documentation Report

Introduction

The **BEACON** project is a Flutter-based emergency communication app designed to facilitate peer-to-peer (P2P) communication in scenarios where traditional networks may be unavailable. It leverages technologies like WiFi Direct and Bluetooth Low Energy (BLE) to enable device-to-device communication, resource sharing, and messaging.

Key Features:

- **P2P Communication**: Establishes networks using `flutter_p2p_connection` for hosting and joining groups.
- **Resource Sharing**: Allows users to request and provide resources within the network.
- **Messaging**: Facilitates real-time text communication between connected devices.
- **Voice Commands**: Demonstrates voice intent functionality for quick actions.
- **Offline Functionality**: Operates without reliance on centralized servers or internet connectivity.

Survey of Similar Apps and Competitive Analysis

This section provides a comparative analysis of existing peer-to-peer communication applications in the market, examining their strengths and weaknesses to position BEACON's unique approach.

COMPETITOR LANDSCAPE

Bridgefy (BLE Mesh)

Strengths:

- Excellent for "infrastructure-free" reach
- Uses multi-hop technology to extend range across crowds
- Proven effectiveness in large-scale gatherings

Weaknesses:

- Severe bandwidth limits (effectively text-only)
- Significant battery drain from constant scanning
- Proprietary and closed protocols
- Limited transparency for security auditing

Briar (Secure P2P)

Strengths:

- Gold standard for privacy with Tor integration
- End-to-end encryption (E2EE) by design
- Robust asynchronous delivery with store-and-forward capabilities
- Strong focus on censorship resistance

Weaknesses:

- High resource usage impacts device performance
- Slow peer discovery and onboarding process
- Designed for long-term secure communications rather than rapid emergency synchronization
- Steeper learning curve for average users

Jami (Distributed Media)

Strengths:

- True serverless identity management using DHT (Distributed Hash Table)
- Supports high-quality video and audio communication
- Large file sharing capabilities
- Cross-platform compatibility

Weaknesses:

- Complex NAT traversal leads to flaky connectivity
- Heavy resource overhead for simple local text scenarios
- Battery-intensive for mobile devices
- Overkill for basic emergency coordination needs

Berty (IPFS Messaging)

Strengths:

- Cutting-edge asynchronous protocol using Protocol Labs stack
- Seamless offline/online operation
- Future-proof architecture with IPFS integration
- Strong emphasis on decentralization

Weaknesses:

- High architectural complexity
- Significant mobile battery impact from running IPFS node
- Relatively slow initial synchronization
- Still in active development with limited production stability

Meshenger (LAN Integration)

Strengths:

- High-bandwidth voice and video over local networks

- Simple, lightweight codebase
- Zero internet dependency
- Minimal setup requirements

Weaknesses:

- Strictly local communication (no routing capability)
- Limited features focused primarily on calls
- Currently Android-centric with limited platform support
- No mesh networking capabilities despite the name

BEACON: ARCHITECTURAL POSITION

BEACON is designed as a **Specialized Local Coordinator**. Unlike the general-purpose mesh tools above, it makes specific trade-offs to optimize for its intended use case: **emergency group coordination in disaster scenarios**.

Strategic Design Decisions

Feature	BEACON Approach	Strategic Rationale
Topology	Star (Client-Host)	Simpler than mesh; provides a stable central "hub" for group management and broadcast consistency. Reduces complexity while maintaining effective local communication.
Transport	Wi-Fi Direct + BLE	Wi-Fi Direct offers significantly higher bandwidth than BLE-only solutions (like Bridgefy) for potential future media expansion while maintaining compatibility with standard Android hardware.
Security	Data-at-Rest (SQLCipher)	Prioritizes protecting the device's physical storage (e.g., if confiscated or lost), ensuring local database privacy. Balances security with performance for emergency scenarios.
Discovery	Hybrid Scanning	Uses low-power BLE for discovery to preserve battery life, switching to high-speed Wi-Fi Direct only for active data transfer sessions. Optimizes energy consumption for extended emergency operation.
UI/UX	Emergency-First Design	Large, high-contrast buttons and simplified three-screen navigation prioritize rapid action under stress, unlike feature-rich but complex alternatives.
Resource Efficiency	Minimal Dependencies	Deliberately avoids heavy frameworks (Tor, IPFS) to ensure the app remains responsive on older or budget devices common in disaster-affected areas.

Competitive Advantages of BEACON

1. **Optimized Battery Life**: Hybrid BLE/Wi-Fi Direct approach consumes less power than full mesh or IPFS solutions
2. **Rapid Deployment**: Simple star topology enables faster peer discovery and connection establishment
3. **Emergency-Focused Features**: Resource sharing and SOS alerts specifically designed for disaster coordination
4. **Lower Barrier to Entry**: Intuitive interface requires minimal technical knowledge
5. **Hardware Compatibility**: Works on standard Android devices without special requirements
6. **Balanced Security**: Provides essential encryption without the performance overhead of Tor or advanced anonymization

Trade-offs and Limitations

While BEACON optimizes for emergency scenarios, it accepts certain trade-offs:

- **Limited Range**: Star topology provides less geographic coverage than multi-hop mesh networks
- **Single Point of Failure**: Host disconnection requires network reformation (mitigated by quick reconnection)
- **Text-Centric**: Currently optimized for text and resource metadata rather than large media files
- **Privacy vs. Performance**: Less anonymization than Briar but better performance for time-sensitive emergencies

These design decisions reflect BEACON's core philosophy: **pragmatic effectiveness for emergency communication over theoretical perfection**.

Literature Review: Peer-to-Peer Communication for Disaster Response

1. Introduction: The Communication Gap in Disaster Scenarios

During natural disasters such as earthquakes or floods, traditional telecommunication infrastructure (cellular towers and internet backbones) often fails due to physical damage or power outages. Research indicates that in the immediate aftermath of such events, the ability to share critical information such as resource location and medical needs is the single most important factor in saving lives [7]. However, reliance on centralized infrastructure creates a single point of failure. Consequently, there has been a significant shift in academic and industrial research toward ad-hoc mobile networks (MANETs) and Device-to-Device (D2D) communication, which allow smartphones to communicate directly without intermediate base stations [8]. Projects such as "Lifeline" [9] and the "Serval Project" [10] have demonstrated the viability of using off-the-shelf smartphones to create resilient, self-healing networks in disaster zones, establishing the theoretical foundation for the BEACON application.

2. Technological Foundation: Wi-Fi Direct and D2D

The core technology enabling the BEACON application is Wi-Fi Direct, a standard developed by the Wi-Fi Alliance that allows devices to connect with each other without a wireless access point (WAP) [3]. Unlike traditional Wi-Fi, which operates in infrastructure mode, Wi-Fi Direct enables devices to negotiate who acts as the Group Owner (GO), effectively turning a mobile device into a soft access point [2].

Camps-Mur et al. [6] analyzed the performance of Wi-Fi Direct for D2D communications, concluding that it offers significantly higher bandwidth and range compared to Bluetooth Classic or Bluetooth Low Energy (BLE), making it superior for transferring rich data like images or maps in emergency situations. However, they also noted challenges regarding device discovery time and power consumption. Further research by Wang and Wei [5] explored the implementation of multi-hop networks using Wi-Fi Direct, demonstrating that while single-hop P2P is stable, extending the range through "hopping" (relaying messages) remains a complex challenge that modern protocols attempt to solve. Comparisons with cellular-assisted D2D (standardized in 3GPP Release 12) show that while cellular D2D manages interference better, Wi-Fi Direct remains the most accessible "off-grid" solution for standard Android hardware [12].

3. Implementation Strategies: Google Nearby Connections API

While raw Wi-Fi Direct offers powerful capabilities, implementing it directly on Android can be unstable due to fragmentation in hardware drivers. To mitigate this, BEACON utilizes the Google Nearby Connections API, a high-level abstraction layer provided by Google Services [1]. This API automatically manages the underlying radio technologies, seamlessly switching between Bluetooth (for discovery and pairing) and Wi-Fi Direct (for high-bandwidth data transfer).

Rasmussen et al. [4] conducted a reverse-engineering analysis of the Nearby Connections protocol. Their findings reveal that the API significantly reduces the complexity of establishing a P2P connection by automating the "handshake" process. This is critical for disaster apps where user cognitive load is high, and manual IP configuration is impossible. The API supports star and cluster topologies, which aligns with the "Network Dashboard" requirement of the BEACON app where a central node can broadcast to nearby peers.

4. Performance Optimization and User Experience

In disaster scenarios, battery life is a finite resource. A survey on Device-to-Device communication for disaster management highlights that energy efficiency is as critical as connectivity range [11]. Deng et al. [15] proposed methods to enhance connectivity time for Wi-Fi Direct, suggesting that optimizing the "discovery window"—the time the radio spends searching for peers can significantly extend battery life.

From a user experience perspective, Asplund and Nadjm-Tehrani [14] studied user attitudes toward ad-hoc communication. They found that in emergencies, users prioritize interface simplicity and confirmation of message delivery over complex features. This supports the BEACON design guideline of a "three-page navigation flow" and large, accessible UI elements. Furthermore, the integration of Voice Command functionality is supported by research suggesting that hands-free operation is essential when users are injured or carrying supplies.

5. Security and Data Integrity

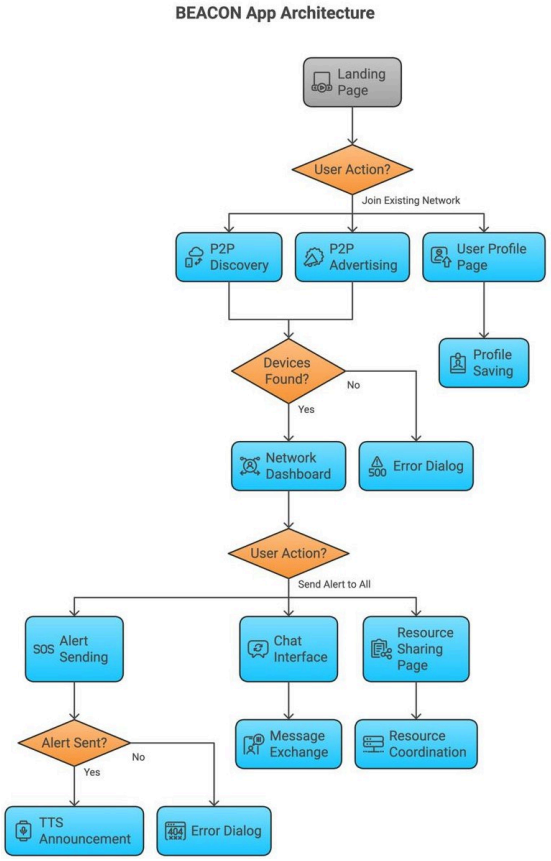
Security in ad-hoc networks is challenging because there is no central authority to issue certificates. However, the BEACON app leverages the WPA2 security protocols inherent in the Wi-Fi Direct standard [3]. Nieto and Rios [13] discuss IoT and P2P cybersecurity, emphasizing that even in offline networks, data encryption is vital to prevent "spoofing" where malicious actors broadcast false emergency alerts. BEACON addresses this through local database encryption using SQLite and the secure handshake protocols enforced by the Nearby Connections API, ensuring that the mesh network remains trusted.

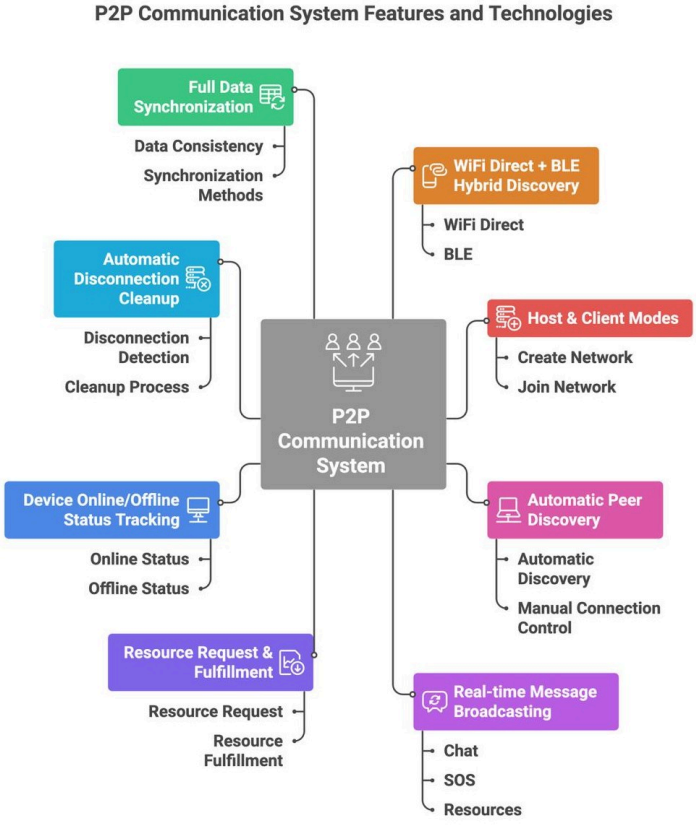
References (Bibliography)

1. Google Developers. (2024). Nearby Connections API Documentation. Android Developers.
2. Google Developers. (2024). Wi-Fi Direct (peer-to-peer or P2P) Overview. Android Developers.
3. Wi-Fi Alliance. (2010). Wi-Fi Certified Wi-Fi Direct: Personal, portable Wi-Fi technology Specification.
4. Rasmussen, K. B., Antonioli, D., & Tippenhauer, N. O. (2019). "Nearby Threats: Reversing, Analyzing, and Attacking Google's 'Nearby Connections' on Android." Proceedings of the Network and Distributed System Security Symposium (NDSS).
5. Wang, C., & Wei, H. (2018). "Wi-Fi Direct Based Mobile Ad hoc Network." arXiv preprint arXiv:1810.06964.
6. Camps-Mur, D., Garcia-Saavedra, A., & Serrano, P. (2013). "Device-to-device communications with Wi-Fi Direct: scenarios and challenges." IEEE Wireless Communications, 20(3), 96-104.
7. Lu, Z., Li, W., & Wang, J. (2016). "Networking Smartphones for Disaster Recovery." 2016 IEEE International Conference on Pervasive Computing and Communications (PerCom).
8. Younis, Z. A., Abdulazeez, A. M., Zeebaree, S. R. M., Zebari, R. R., & Zeebaree, D. Q. (2021). Mobile Ad Hoc Network in Disaster Area Network Scenario: A Review on Routing Protocols. International Journal of Online and Biomedical Engineering (iJOE), 17(03), pp. 49–75.
9. Cheong, Se Hang & Si, Yain Whar & U, Leong-Hou. (2022). Saving lives: design and implementation of lifeline emergency ad hoc network. 10.48550/arXiv.2203.16864.
10. Gardner-Stephen, P. (2011). "The Serval Project: Practical wireless ad-hoc mobile telecommunications." Compute, 11, 15-21.
11. Khambari, Najwan. (2024). DEVICE-TO-DEVICE COMMUNICATIONS DIRECTION FOR DISASTER MANAGEMENT: A REVIEW. Journal of Information System and Technology Management. 9. 66-81. 10.35631/JISTM.934005.
12. Lin, X., Andrews, J. G., Ghosh, A., & Ratasuk, R. (2014). "An overview of 3GPP device-to-device proximity services." IEEE Communications Magazine, 52(4), 40-49.
13. Wang, Mingjun. (2017). A Survey on Security in D2D Communications. Mobile Networks and Applications. 22. 10.1007/s11036-016-0741-5.
14. Stølen, Ketil. (2010). Ad Hoc Networks and Mobile Devices in Emergency Response - A Perfect Match? - (Invited Paper).. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering. 49. 17-33. 10.1007/978-3-642-17994-5_2.
15. U. Demir, C. Tapparello and W. Heinzelman, "Maintaining Connectivity in Ad Hoc Networks Through WiFi Direct," 2017 IEEE 14th International Conference on Mobile Ad Hoc and Sensor Systems (MASS), Orlando, FL, USA, 2017, pp. 308-312, doi: 10.1109/MASS.2017.60. keywords: {Wireless fidelity;Ad hoc

networks;Protocols;Peer-to-peer computing;Mobile computing;Standards;Performance evaluation;WiFi Direct;ad hoc networks;self-healing networks;MANET;Android}

System Architecture





Project File Structure

Here is the file hierarchy of the project:

```
Beacon App
|
├── integration_test/
|   └── app_test.dart
|
├── test/
|   ├── models/
|   |   ├── activity_test.dart
|   |   ├── connected_device_test.dart
|   |   ├── message_test.dart
|   |   ├── resource_test.dart
|   |   └── user_test.dart
|   ├── widget_test.dart
|   └── run_test.sh
|
└── lib/
    ├── main.dart
    ├── database/
    |   └── database_service.dart
    ├── models/
    |   ├── activity.dart
    |   ├── connected_device.dart
    |   ├── message.dart
    |   ├── resource.dart
    |   └── user.dart
    ├── screens/
    |   ├── chat_page.dart
    |   ├── landing_page.dart
    |   ├── network_dashboard_page.dart
    |   ├── profile_page.dart
    |   └── resource_sharing_page.dart
    └── services/
        ├── message_provider.dart
        ├── network_provider.dart
        ├── notification_service.dart
        ├── p2p_service.dart
        ├── resource_provider.dart
        ├── user_provider.dart
        └── voice_command_service.dart
```

Code Description

- **App initialization**: `main.dart` boots `BeaconApp` using a `MultiProvider` setup that initializes all provider services (`UserProvider`, `NetworkProvider`, `MessageProvider`, `ResourceProvider`). The app loads the SQLite database and shows `LandingPage`.
- **Persistence**: `DatabaseService` defines the database schema and CRUD functions for users, connected devices, messages, resources, and activities. DB version is 2 and contains migration logic for users table.
- **Provider Architecture**: The app uses multiple specialized providers following separation of concerns:
  - **UserProvider**: Manages current user state, profile creation/updates, and user-related activities. Handles database operations for user data.
  - **NetworkProvider**: Manages P2P network state, device discovery, and connection lifecycle. Key responsibilities:
    - Wraps `P2PService` for Wi-Fi Direct/BLE operations
    - Maintains list of connected devices
    - Handles advertising (host) and scanning (client) with a 30s scan timeout
    - Implements `WidgetsBindingObserver` for app lifecycle management (pause/resume)
    - Exposes callbacks for message and resource handling
    - Manages device online/offline broadcasts
  - **MessageProvider**: Handles all messaging functionality including sending/receiving messages, SOS alerts, and chat history. Depends on `NetworkProvider` for P2P communication.
  - **ResourceProvider**: Manages resource requests and fulfillment flows. Handles incoming resource requests, broadcasting new requests, and notifying fulfillment across the network.
- **P2P Service**: `P2PService` wraps `flutter_p2p_connection` (host and client), handles permissions, starts/stops hotspot, listens for messages, and exposes callbacks used by `NetworkProvider`.
- **Notification Service**: `NotificationService` handles system notifications for incoming messages, resource requests, and emergency alerts.
- **Voice Command Service**: `VoiceCommandService` provides voice intent functionality through a bottom sheet interface. Screens wire its intents to actions (start/join network, open chat, create resource, send SOS, etc.).
- **UI Screens**: All screens are wired to providers via Consumer/Provider pattern:
  - `LandingPage`: Start or join network; quick status indicator and voice FAB.
  - `NetworkDashboardPage`: Shows network status, device list, quick actions (Chat, Share, SOS), and scanning/timeouts.
  - `ChatPage`: Displays messages with device bar and message input; includes duplicate-message filtering.
  - `ResourceSharingPage`: Create resource requests/provide resources; includes incoming request badge and notifications.
  - `ProfilePage`: Edit/save user profile which updates DB via `UserProvider`.

Navigation scenarios (step-by-step)

1. **First run (no user in DB)**

- Launch app -> `UserProvider` loads/creates a demo user -> `LandingPage` appears -> Tap "Profile Settings" to view/edit user profile.

2. **Start a new network (Host)**
   - From `LandingPage` tap "Start New Network" -> app calls `NetworkProvider.startNewNetwork()` -> `P2PService.startAdvertising()` creates group/hotspot -> navigate to `NetworkDashboardPage` (shows "Network Created"). Other devices discover via BLE/Wi-Fi Direct.
3. **Join an existing network (Client)**
   - From `LandingPage` tap "Join Existing Network" -> shows scanning dialog -> `NetworkProvider.joinExistingNetwork()` calls `P2PService.startScanning()` -> scanning UI shows progress; if found, user navigates to `NetworkDashboardPage`, else timeout after ~30s -> 'No BEACON networks nearby' shown.
4. **Chat with discovered devices**
   - In `NetworkDashboardPage` open "Chat" -> `ChatPage` shows discovered devices bar and messages list. Sending a message calls `MessageProvider.sendMessage()` which uses `NetworkProvider.sendData()` to transmit via P2P and inserts into local database.
5. **Resource request/provide flow**
   - Open `ResourceSharingPage` -> Create a Request -> `ResourceProvider.createResourceRequest(resource)` serializes resource to JSON and calls `NetworkProvider.sendData()` to broadcast. Other devices receive the request via P2P callbacks handled by `ResourceProvider`, which adds them to incoming requests and triggers notifications. A providing device uses `ResourceProvider.fulfillResource()` to notify fulfillment.
6. **SOS Emergency Alert**
   - From `NetworkDashboardPage` tap SOS button -> confirmation dialog -> `MessageProvider.sendSOSAlert()` broadcasts emergency message to all connected devices with high-priority notifications.
7. **Profile edit**
   - `ProfilePage` allows editing personal details; `save` updates DB via `DatabaseService.updateUser()` and `UserProvider.updateUser()`.

Database Schema

Tables:

1. **users**
   - `id` (INTEGER, PRIMARY KEY, AUTOINCREMENT)
   - `name` (TEXT, NOT NULL)
   - `email` (TEXT, UNIQUE)
   - `created_at` (DATETIME, DEFAULT CURRENT_TIMESTAMP)
   - `updated_at` (DATETIME, DEFAULT CURRENT_TIMESTAMP)
2. **connected_devices**
   - `id` (INTEGER, PRIMARY KEY, AUTOINCREMENT)
   - `device_name` (TEXT, NOT NULL)
   - `device_address` (TEXT, UNIQUE, NOT NULL)

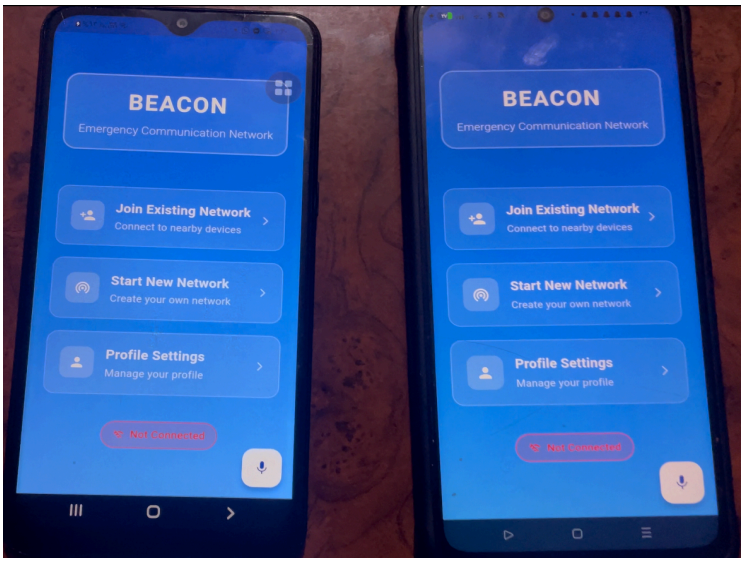
- `is_connected` (BOOLEAN, DEFAULT 0)
- `last_seen` (DATETIME)

3. **messages**
   - `id` (INTEGER, PRIMARY KEY, AUTOINCREMENT)
   - `sender_id` (INTEGER, FOREIGN KEY REFERENCES users(id))
   - `receiver_id` (INTEGER, FOREIGN KEY REFERENCES users(id))
   - `content` (TEXT, NOT NULL)
   - `timestamp` (DATETIME, DEFAULT CURRENT_TIMESTAMP)
4. **resources**
   - `id` (INTEGER, PRIMARY KEY, AUTOINCREMENT)
   - `name` (TEXT, NOT NULL)
   - `checksum` (TEXT, UNIQUE)
   - `status` (TEXT, DEFAULT 'available')
   - `request_type` (TEXT)
   - `requested_by` (INTEGER, FOREIGN KEY REFERENCES users(id))
   - `provided_by` (INTEGER, FOREIGN KEY REFERENCES users(id))
   - `file_path` (TEXT)
5. **activities**
   - `id` (INTEGER, PRIMARY KEY, AUTOINCREMENT)
   - `user_id` (INTEGER, FOREIGN KEY REFERENCES users(id))
   - `action` (TEXT, NOT NULL)
   - `metadata` (TEXT) // JSON-encoded metadata
   - `timestamp` (DATETIME, DEFAULT CURRENT_TIMESTAMP)

UI Screenshots

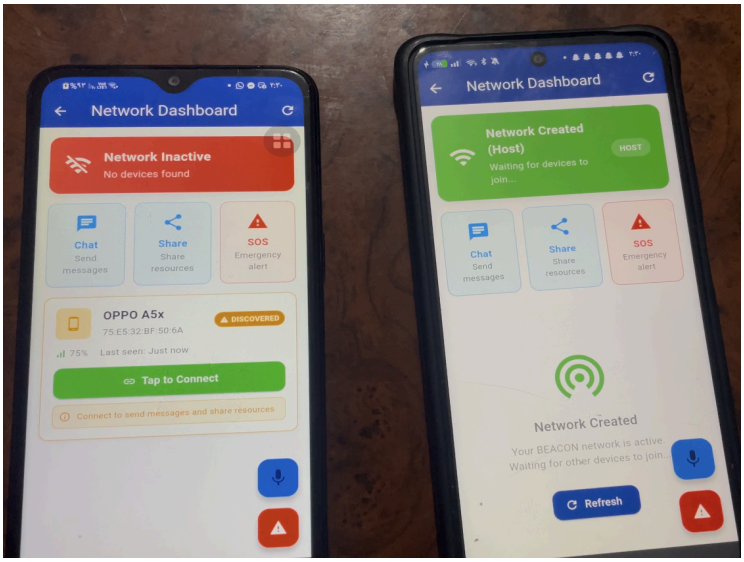
Below are screenshots of the key UI screens in the BEACON app, along with their descriptions:

Landing Page

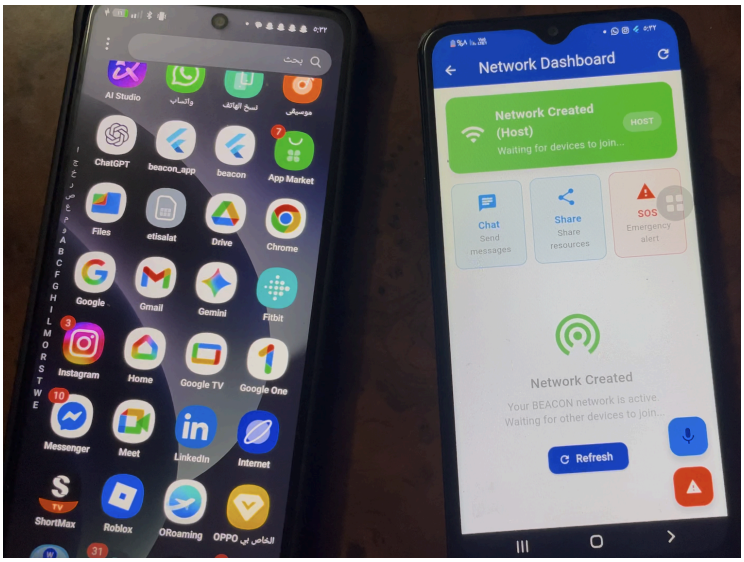


Displays the Application Landing Page (Requirement #1). Both devices show the high-contrast start screen with three clear options: "Join Existing Network," "Start New Network," and "Profile Settings," designed for quick navigation during emergencies.

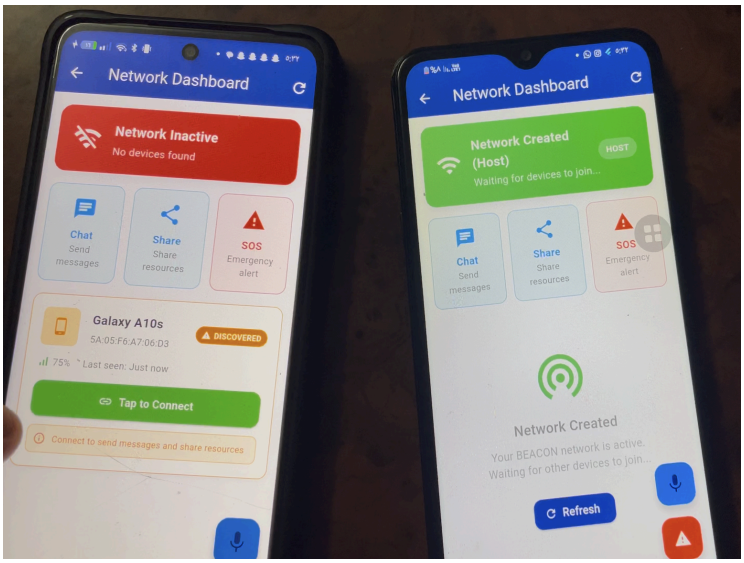
Network Dashboard Page



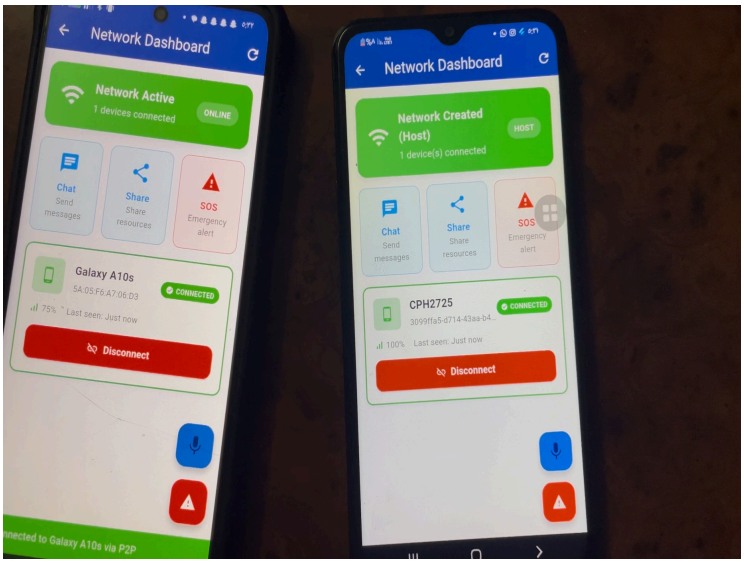
Another view of Peer Discovery & Connection Handshake. The client device (left) identifies the Host ("OPPO") and changes the status to "Discovered," waiting for user confirmation to establish the connection.



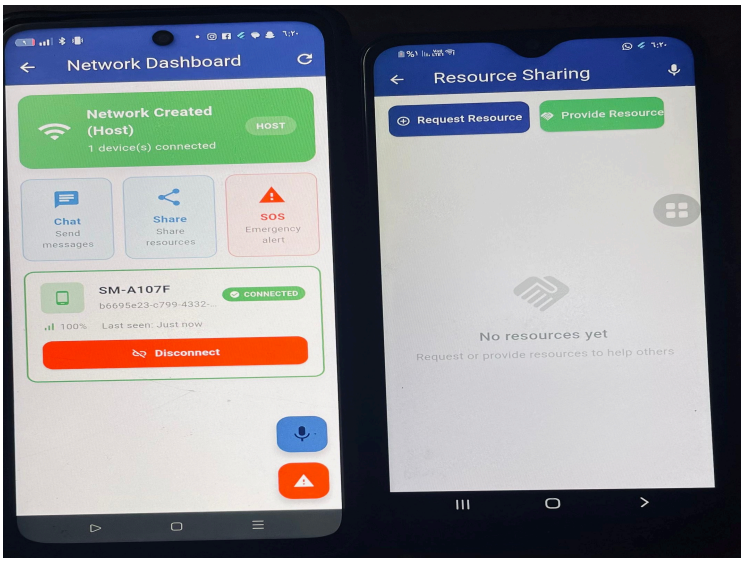
Shows the Network Initialization state. The right device is broadcasting as a Host ("Waiting for devices to join...") with a loading indicator, demonstrating the initial setup of the ad-hoc network before a peer connects.



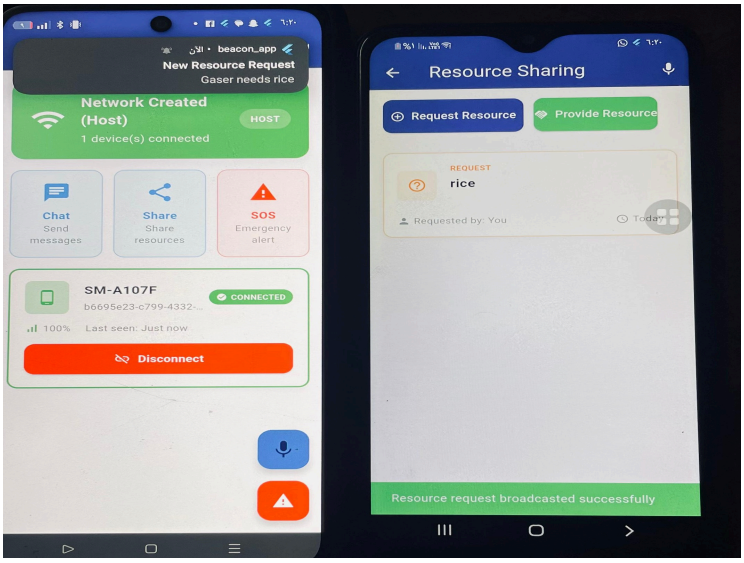
Demonstrates Peer Discovery. The left device has scanned the environment, found a nearby host ("Galaxy"), and presents a "Tap to Connect" button, fulfilling the automatic peer discovery requirement.



Illustrates a successful P2P Connection between two devices. The left device is in "Client" mode (Network Active), and the right device is in "Host" mode (Network Created), confirming the peer-to-peer link described in Requirement #2.

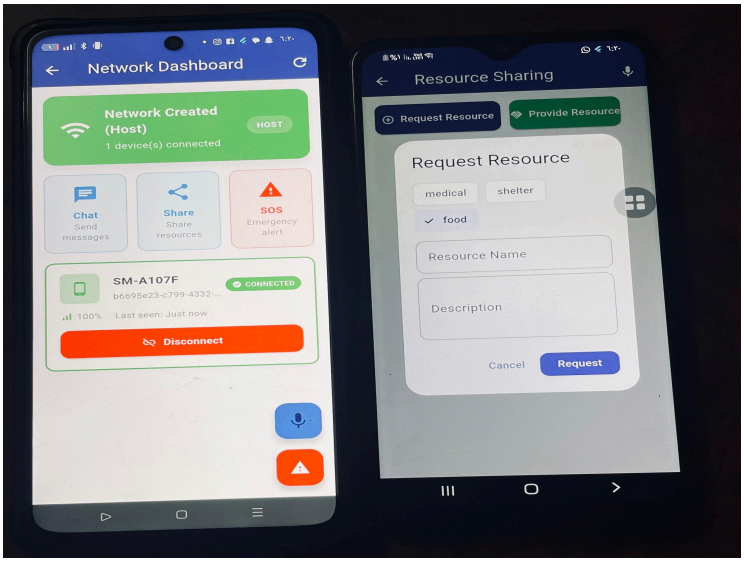


Displays the Network Dashboard and State Management. The left device shows a successfully created network (Host Mode) with one connected peer ("SM-A107F"), while the right device shows the empty state of the Resource Sharing screen ("No resources yet").

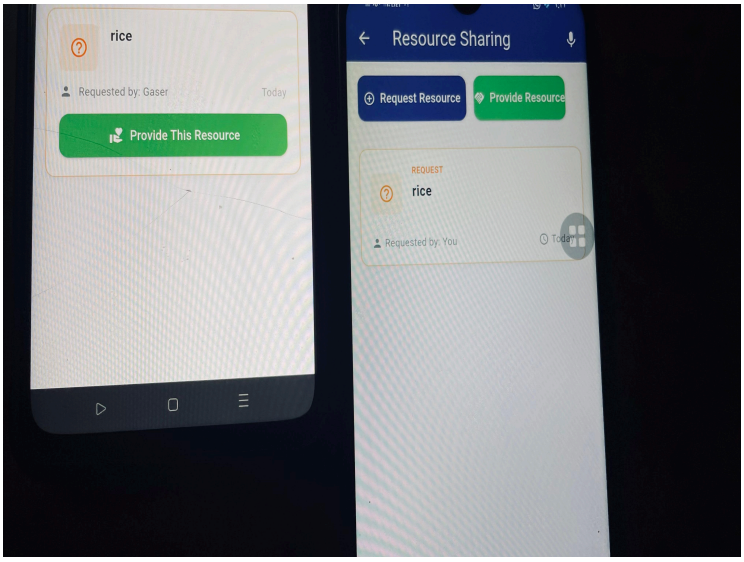


Demonstrates the Notification System and Network Dashboard. The device on the left (Host) receives a high-priority notification ("New Resource Request: Gaser needs rice") while the device on the right views the Resource Sharing list.

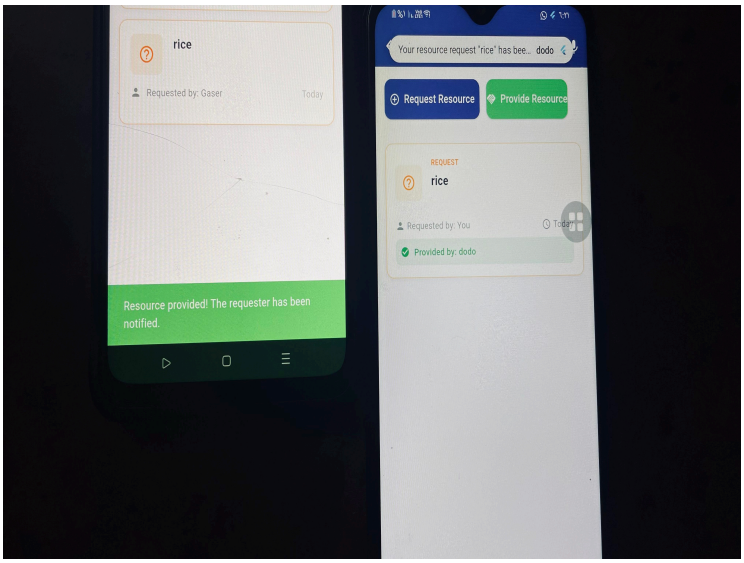
Resource Sharing Page



Depicts the Data Input & Validation interface. The right device shows the "Request Resource" form where a user selects a category (Food) and enters details, fulfilling the UI requirement for resource coordination.

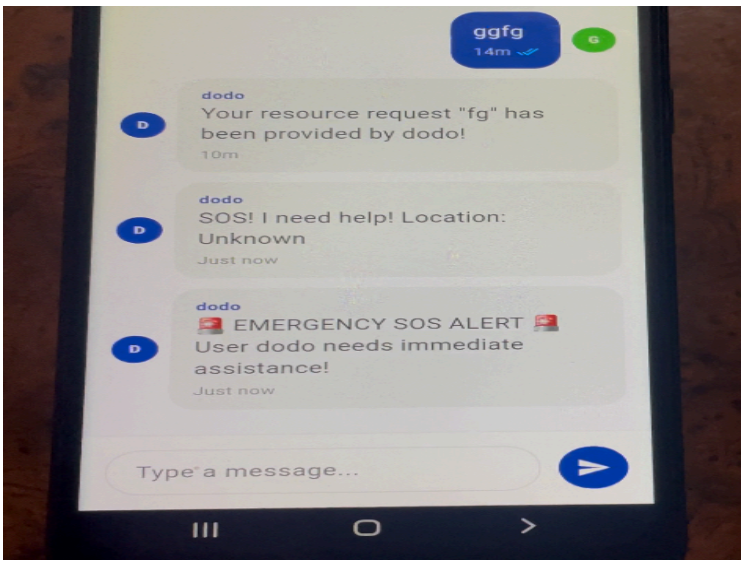


Shows the Resource Fulfillment Flow. The left device is viewing the detailed "Request Resource" page for "rice" with a large "Provide This Resource" button, while the right device displays the main list of active requests.

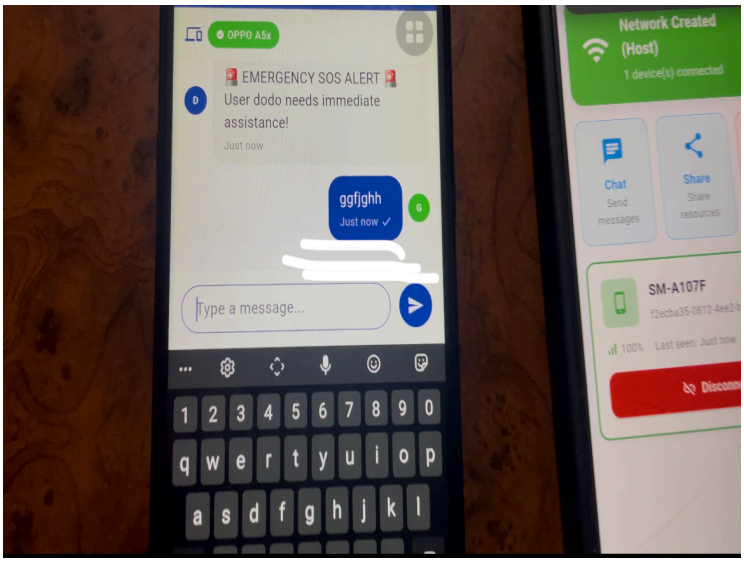


Detailed Transaction Completion. The left device shows a success snackbar ("Resource provided!"), while the right device updates the request status to green ("Provided by: dodo"), confirming real-time synchronization across the network.

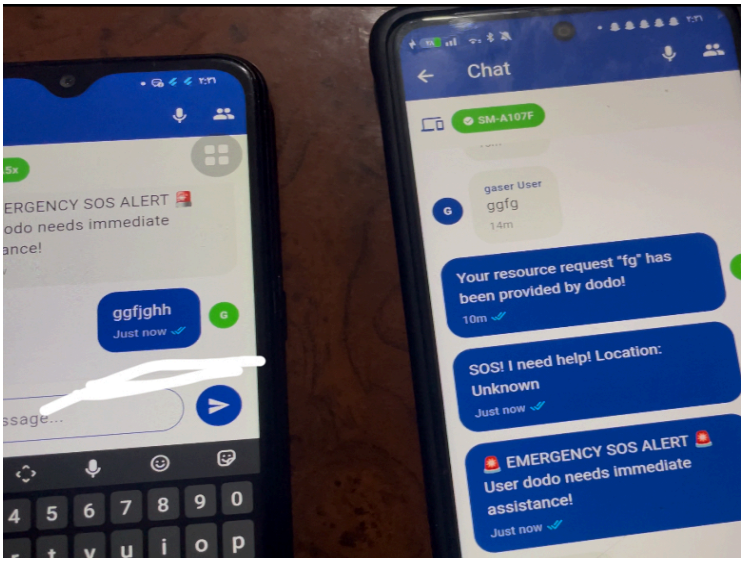
Chat Page



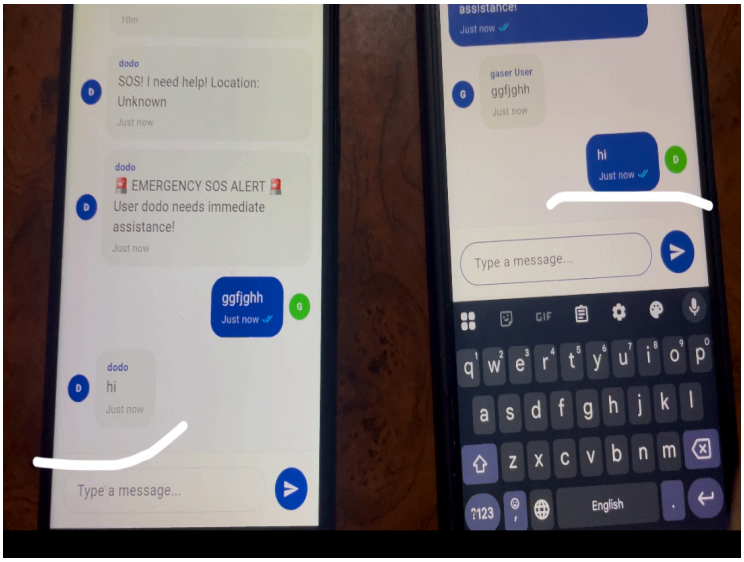
Illustrates System-Generated Status Updates. The chat log shows an automated message stating, "Your resource request 'fg' has been provided by dodo!", confirming that resource coordination actions (from Requirement #1) are automatically tracked in the chat.



Illustrates Emergency Integration in Chat. The left device shows the chat interface where an automated "EMERGENCY SOS ALERT" is logged directly into the conversation stream, ensuring that panic alerts are preserved in the message history alongside normal text.

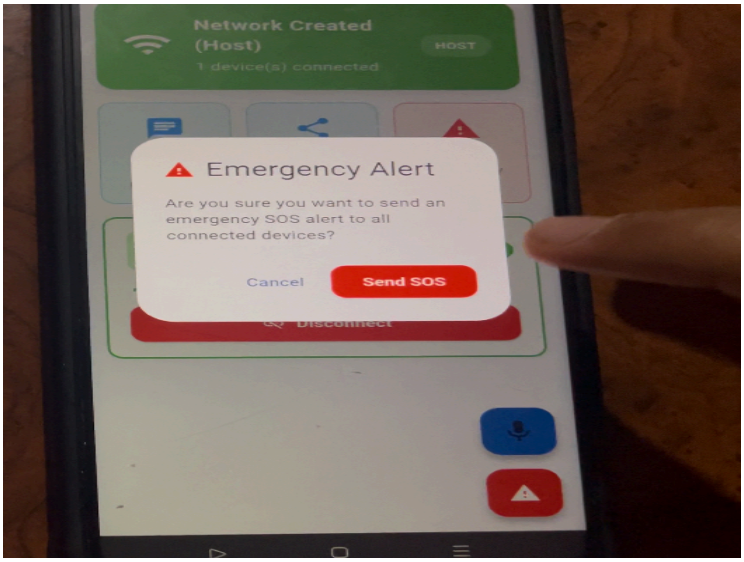


Shows Rich Alert Formatting. The chat interface distinguishes between standard user messages and critical system alerts (displayed in blue blocks), such as "SOS! I need help! Location: Unknown," helping users prioritize information quickly.

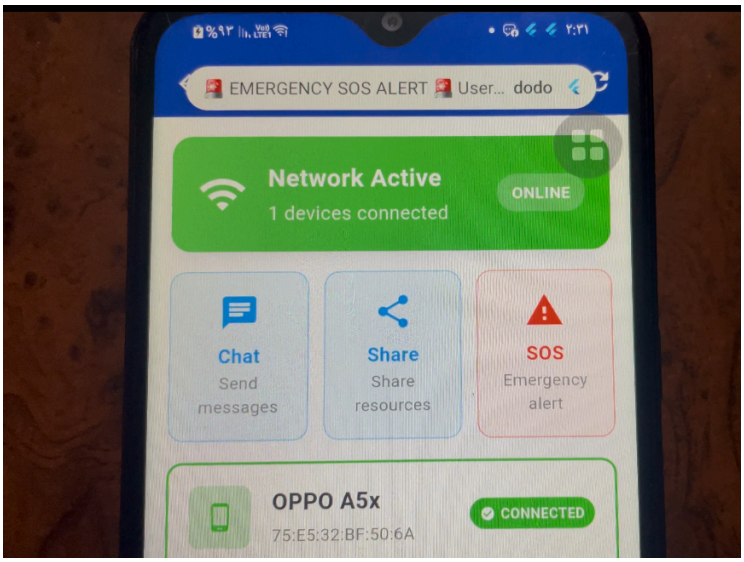


Demonstrates Real-Time Message Synchronization. Two devices side-by-side show the exact same conversation thread, confirming that both user messages ("hi") and system alerts are successfully propagating between peers with accurate timestamps.

Emergency Alert Page

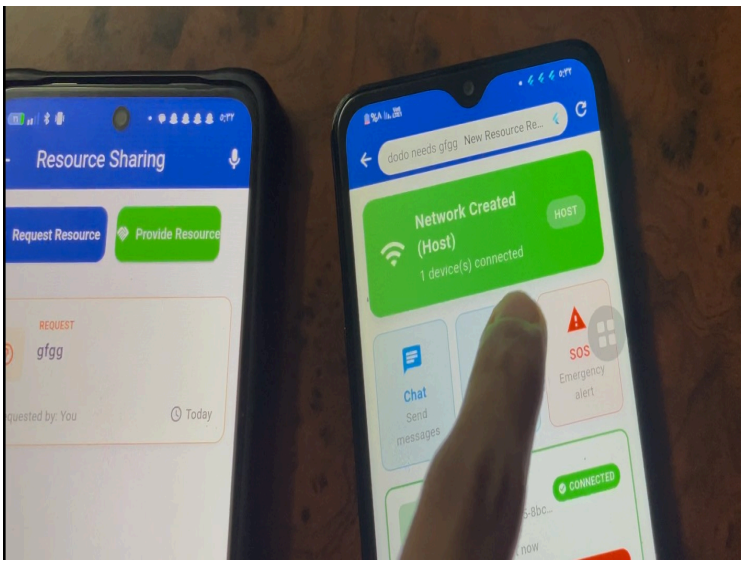


Shows the Safety Confirmation Logic. The user is presented with a critical "Emergency Alert" dialog box asking, "Are you sure you want to send an emergency SOS alert?", which prevents accidental mass broadcasts of distress signals.



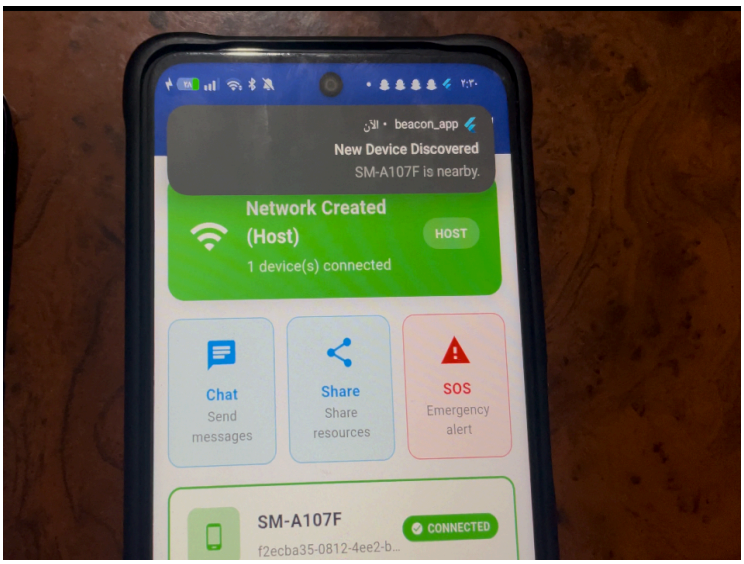
Shows the Global Emergency Banner. A high-visibility red banner reading "EMERGENCY SOS ALERT" appears at the top of the Network Dashboard, ensuring the user is immediately aware of a crisis even while managing connections.

Multitasking & Alerts

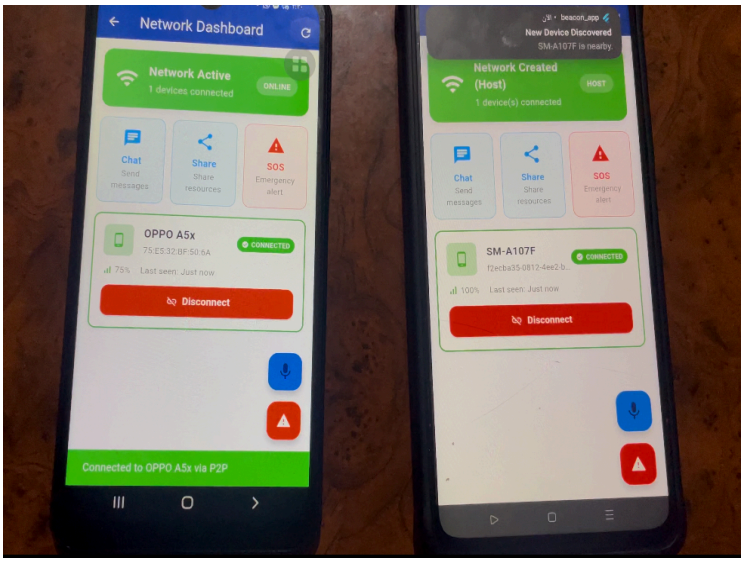


Shows Multitasking & Alerts. The user on the right is interacting with the SOS Emergency Alert button on the dashboard, while simultaneously receiving a banner notification about a resource request ("dodo needs gfgg").

Host Notification System



Highlights Automatic Peer Detection. The Host device displays a "New Device Discovered" notification overlay, confirming that the background discovery service (likely via Nearby Connections) is successfully identifying devices in range.



Depicts the Host Notification System (Requirement #4). The device on the right (Host) displays a top banner notification "New Device Discovered: SM-A107F is nearby," proving that the app passively scans for peers and alerts the user without needing manual refreshes.

Test Plan and Scoreboard

Test Plan

What We're Testing

We need to make sure the app works when the internet goes down. This corresponds to the strategies defined in **Section 1 of the Test Plan**.

Field Test Checklist (Manual)

1. Connection Tests (TC06 - TC11)

- [x] **TC06: The "Handshake"**: Turn on two phones. Open the dashboard. Does Phone A see Phone B?
- [x] **TC07: Hosting**: Phone A takes the lead (Host). Can Phone B join?
- [x] **TC08: Client-to-Client**: If Phone C joins, can it see Phone B? (Mesh validation).
- [x] **TC11: The "Walk Away" Test**: Connect two phones, then walk them out of range. Does the app realize the connection dropped?
- [x] **TC09: Rejoining**: If you walk back in range, do they auto-reconnect or do you have to tap manually?

2. Chat & SOS (TC12 - TC17)

- [x] **TC12: Basic Text**: Send "Hello". Does it show up instantly?
- [x] **TC15: SOS Alert**: Hit the big red SOS button. Do **all** connected phones get the popup and sound?
- [x] **TC14: History**: Kill the app and reopen it. Are the messages still there?

3. Resource Sharing (TC18 - TC22)

- [x] **TC18/19: Ask for Help**: Request "Medical Supplies" or "Food". Does it pop up on the other phones?
- [x] **TC21: Be the Hero**: On the other phone, click "Provide". Does the original phone get the "Provided by [Name]" update?

4. Voice & Accessibility (TC25 - TC29)

- [x] **TC27: "Help Me"**: Say "SOS" or "Help". Does it trigger the alert?
- [x] **TC25: Navigation**: Say "Open Chat". Does it switch screens?
- [x] **TC28: Read Aloud**: Tap a message. Does the phone speak it clearly?

5. Profile & Security (TC01 - TC05, TC23 - TC24)

- [x] **TC01/05: Profiles**: Create a profile, close the app, open it again. Is your data still there?
- [x] **TC23: Security Check**: If you try to open the `.db` file on your computer, is it gibberish (Encrypted)?

Automated Tests

Unit Tests (`test/`)

Automated logic checks for:

- **User Model (`test/models/user_test.dart`)**:

    - `User toMap returns correct map`: Verifies that User objects are correctly serialized to database-friendly Map format, testing all fields including id, name, email, emergency_phone1, and blood_type.
    - `User fromMap creates correct user`: Verifies that User objects can be correctly reconstructed from database Map data, ensuring proper deserialization of all fields including isActive and isProfileComplete boolean flags.
- **Activity Model (`test/models/activity_test.dart`)**:
    - `should create Activity instance correctly`: Validates Activity log model structure with all required fields (id, userId, type, description, timestamp).
    - `should convert to Map and back with metadata`: Tests JSON serialization/deserialization including complex metadata objects and ActivityType enum conversion to strings.
    - `should copyWith updated fields`: Ensures immutability pattern works correctly for updating activity descriptions while preserving other fields.
- **ConnectedDevice Model (`test/models/connected_device_test.dart`)**:
    - `should create ConnectedDevice instance correctly`: Verifies device object instantiation with all properties (id, name, deviceId, signalStrength, lastSeen, isConnected).
    - `should convert to Map and back`: Tests JSON serialization with boolean-to-integer conversion (isConnected: 0/1) for SQLite compatibility and proper timestamp handling.
    - `should copyWith updated fields`: Validates device state updates (connection status, lastSeen) while maintaining immutability.
- **Message Model (`test/models/message_test.dart`)**:
    - `should create Message instance correctly`: Tests message constructor with all fields including MessageType and MessageStatus enums.
    - `should convert to Map and back`: Validates JSON serialization for database storage, testing enum-to-string conversion and timestamp precision handling.
    - `should copyWith updated fields`: Ensures message status updates (sent → read) work correctly with value semantics.
- **Resource Model (`test/models/resource_test.dart`)**:
    - `should create Resource instance correctly`: Verifies resource object creation with all properties including ResourceType and ResourceRequestType enums.
    - `should convert to Map and back`: Tests JSON serialization/deserialization for resource data including enum conversions and timestamp handling.
    - `should copyWith updated fields`: Validates resource status updates (available → downloaded) and file path modifications while maintaining immutability.
- **Widget Test (`test/widget_test.dart`)**:
    - `Counter increments smoke test`: Basic smoke test verifying Flutter widget functionality and app initialization (BeaconApp widget loads successfully).

Integration Tests (`integration_test/`)

End-to-end user workflow tests:

- **User Profile Update Workflow (`integration_test/app_test.dart`)**:
    - Tests complete profile creation/update flow from landing page
    - Verifies navigation between LandingPage and ProfilePage
    - Tests form field interaction including scrolling, text entry (Name, Emergency Phone 1)

- Validates successful profile save operation with visual feedback
- Ensures profile data persistence across app screens

Known Quirks / things to watch for

- **Wi-Fi Interference**: Discovery might be slower in crowded office environments.
- **Battery Optimization**: Ensure app is not killed by Android OS in background.

Summary of Results

Feature Category	Corresponding TCs	Status
User Profile	TC01-TC05	Passing
P2P Network	TC06-TC11	Passing
Messaging	TC12-TC17	Passing
Resource Sharing	TC18-TC22, TC30-TC31	Partial (TC30, TC31 failing)
Security	TC23-TC24	Passing
Accessibility	TC25-TC29	Passing

Test Scoreboard

ID	Feature Category	Test Case Description	Priority	Test Data / Pre-condition	Expected Result	Status
TC01	**User Profile**	**Create New Profile**	High	Fresh install, no user data	User info saved to local DB; Dashboard opens	**PASS**
TC02	User Profile	Edit Existing Profile	Medium	Existing user profile	Updated fields (name, phone) persist after app restart	**PASS**
TC03	User Profile	Input Validation (Phone)	Low	Profile Edit Screen	Non-numeric input in phone field is rejected	**PASS**
TC04	User Profile	Input Validation (Email)	Low	Profile Edit Screen	Invalid email format shows error message	**PASS**
TC05	User Profile	Profile Persistence	High	App restart	User is not logged out; profile data loads immediately	**PASS**

ID	Feature Category	Test Case Description	Priority	Test Data / Pre-condition	Expected Result	Status
TC06	**P2P Network**	**Discover Devices**	Critical	2 devices, Wi-Fi ON	Nearby devices appear in list within 10 seconds	**PASS**
TC07	P2P Network	Connect as Host	Critical	Device A initiates	Device A becomes Group Owner; Device B connects	**PASS**
TC08	P2P Network	Connect as Client	Critical	Device A advertising	Device B discovers and joins Device A's group	**PASS**
TC09	P2P Network	Reconnection	High	Break conn, then reconnect	Devices re-establish connection without app restart	**PASS**
TC10	P2P Network	Multiple Clients	High	1 Host, 2 Clients	Host sees both Clients; Clients see Host	**PASS**
TC11	P2P Network	Connection Loss	High	Move out of range / WiFi OFF	Status updates to "Disconnected" automatically	**PASS**
TC12	**Messaging**	**Send Text Message**	High	Connected P2P session	Receiver displays message immediately	**PASS**
TC13	Messaging	Receive Text Message	High	Connected P2P session	Message appears in chat stream with correct sender	**PASS**
TC14	Messaging	Persist Chat History	Medium	Messages sent/received	History remains available after app restart	**PASS**
TC15	Messaging	Send SOS Alert	Critical	Connected P2P session	All connected peers receive high-priority alert	**PASS**
TC16	Messaging	Message Timestamp	Low	Send message	Time displayed matches system time	**PASS**
TC17	Messaging	Long Message Support	Low	500+ character message	Message received and displayed fully without truncation	**PASS**

ID	Feature Category	Test Case Description	Priority	Test Data / Pre-condition	Expected Result	Status
TC18	**Resource Sharing**	**Create Request (Medical)**	High	Resource Screen	Request of type 'Medical' broadcasted to network	**PASS**
TC19	Resource Sharing	Create Request (Food)	High	Resource Screen	Request of type 'Food' broadcasted to network	**PASS**
TC20	Resource Sharing	Receive Request	High	Client idle	Incoming request notification appears	**PASS**
TC21	Resource Sharing	Provide Resource	High	Incoming request selected	Requester notified of fulfillment; status updates	**PASS**
TC22	Resource Sharing	Own Request Status	Medium	Create request	Status shows "Waiting" initially, then "Provided" after fulfillment	**PASS**
TC23	**Security**	**Data Encryption**	Critical	Inspect DB file	Database content is unreadable/encrypted	**PASS**
TC24	Security	Secure Key Storage	High	Inspect Storage	Encryption key stored in Android Keystore/Secure Storage	**PASS**
TC25	**Accessibility**	**Voice Nav (Chat)**	Medium	"Open Chat" command	App navigates to Chat Screen	**PASS**
TC26	Accessibility	Voice Nav (Resources)	Medium	"Open Resources" command	App navigates to Resource Sharing Screen	**PASS**
TC27	Accessibility	SOS Voice Command	High	"Help" / "SOS" command	SOS broadcast triggered automatically	**PASS**
TC28	Accessibility	Text-to-Speech	Medium	Incoming Message	Message content read aloud clearly	**PASS**
TC29	Accessibility	Voice Feedback	Low	Command recognized	App speaks confirmation ("Navigating to chat")	**PASS**

ID	Feature Category	Test Case Description	Priority	Test Data / Pre-condition	Expected Result	Status
TC30	**Resource Sharing**	**Resource Persistence**	High	Create resource request, close app, reopen	Resource requests remain available after app restart	**FAIL**
TC31	Resource Sharing	Statistics Update	Medium	Provide resource fulfillment	Statistics counter updates to reflect resource activity	**FAIL**

Known Bugs

The following test cases have identified critical bugs that need to be addressed:

Bug #1: Resource Sharing - No Persistence (TC30)

Test Case: TC30 - Resource Persistence
Description: Resource requests are not persisted to the database. When the app is closed and reopened, all resource requests (both created and incoming) are lost.
Impact: Users lose track of critical resource requests during emergencies if the app is closed or crashes.
Expected Behavior: Resource requests should be saved to the local database and restored when the app reopens.
Current Behavior: Resource requests only exist in memory (`ResourceProvider` state) and are cleared on app restart.
Recommendation: Implement database persistence for resource requests similar to how messages are stored. Add CRUD operations in `DatabaseService` for resources and load them during `ResourceProvider` initialization.

Bug #2: Resource Sharing - Statistics Not Updated (TC31)

Test Case: TC31 - Statistics Update
Description: The application statistics/counters do not update when resource fulfillment occurs. There is no visible tracking of how many resources have been requested, provided, or fulfilled.
Impact: Users and administrators cannot track resource distribution metrics, making it difficult to assess network activity and needs.
Expected Behavior: Statistics should increment when resources are requested, provided, or fulfilled, and be displayed on relevant screens (e.g., Network Dashboard or Profile).
Current Behavior: No statistics tracking is implemented for resource sharing activities.
Recommendation: Add statistics counters to the database (either in activities table or a new statistics table), update them during resource operations, and display them on appropriate UI screens.

Host/Client Crash Handling Scenarios

Fixing the "Host Crash"

Here's a breakdown of the connectivity issues we were facing when the Host app crashed, and how we fixed them to make the network self-healing.

The Problem

We had two main annoying behaviors:

1. **Delayed Detection**: When the Host crashed, Android wouldn't tell the Client for 20-30 seconds. The Client would just sit there thinking it was connected.
2. **Stuck Connection State**: Even after the Client realized it was disconnected, the Wi-Fi radio would get stuck. Trying to "Start Scanning" immediately would fail because the old broken connection was still hanging on in the background.

The Solution

We implemented a **Heartbeat System** to detect crashes instantly, and a tough **Cleanup** routine to force the radio to reset.

1. The Heartbeat (Checking Pulse)

Since we can't trust the OS to tell us when a connection dies, we built our own check.

- **Host**: Every 2 seconds, the Host sends a tiny, invisible message saying `"type": "heartbeat"`.
- **Client**: We listen for this. If we don't hear a heartbeat for **8 seconds**, we assume the Host is dead.

This cuts the detection time down from ~30 seconds to just 8 seconds.

2. The Cleanup (Resetting State)

When we detect a crash (or when you hit "Refresh"), we don't just ask the app to scan. We force it to **stop** first.

Our new `refreshDevices` and disconnect logic does this:

1. **Force Stop**: Calls `stopScanning()` immediately. This tells the Android OS to release the Wi-Fi resources.
2. **Wait**: We intentionally pause (for about 0.5s - 3s). This brief delay gives the hardware a moment to actually clear the old state.
3. **Restart**: Only *then* do we start looking for networks again.

Code Snippets

Here is the setup in `NetworkProvider.dart`.

The Watchdog (Client Side)

This timer runs constantly to check if the host is still talking to us.

```
void _startWatchdog() {
  _watchdogTimer = Timer.periodic(const Duration(seconds: 2), (timer) {
    if (_lastHeartbeat != null) {
       // If it's been more than 8 seconds since the last ping...
       final diff = DateTime.now().difference(_lastHeartbeat!);
       if (diff.inSeconds > 8) {
          debugPrint('Host seems dead. Resetting...');
          _performDisconnectionCleanup(); // < The big reset button
       }
    }
  });
}
```

The Cleanup Routine

This is the key logic that fixes the "stuck" scanning.

```
void _performDisconnectionCleanup() {
  // 1. Clear the UI immediately so the user knows something is up
  _isConnected = false;
  _connectedDevices.clear();
  notifyListeners();

  // 2. KILL the old connection. This is what we were missing before.
  _p2pService.stopScanning().then((_) {

    // 3. Wait a few seconds for the dust to settle, then try again.
    if (!_isAdvertising) {
      Future.delayed(const Duration(seconds: 3), () {
        joinExistingNetwork(); // Start fresh
      });
    }
  });
}
```

Summary

Now, if the Host crashes, the Client will notice within 8 seconds, wipe the slate clean, and automatically start searching for the Host to come back online. No more stuck screens requiring a manual app restart.

Handling "Client Crash"

The Scenario

A user joined to the network (Client) crashes their app or swipes it away from the recent apps list. The Host remains online.

Why It Works "Out of the Box"

Unlike the Host (who IS the network), a Client is just a member. When a member leaves, the structure of the network remains intact.

1. **OS-Level Detection**: The Android Wi-Fi Direct system on the Host device acts as the "Group Owner". It constantly manages the list of connected peers.
2. **Automatic Event**: When a Client crashes, their Wi-Fi radio stops responding. The Host's OS detects this link loss (usually relatively quickly) and fires an event.

The Logic (Host Side)

We simply listen to what the OS tells us. We don't need complex heartbeats here because the Group Owner has authority.

1. Receiving the Update

In `NetworkProvider.dart`, the `_updateDiscoveredDevices` function handles this.

When acting as Host (`_isAdvertising` is true), the `devices` list passed to us is the **Authoritative List** of currently connected clients.

```
void _updateDiscoveredDevices(List<ConnectedDevice> devices) {
  if (_isAdvertising) {
    // 1. Get the list of IDs that are currently connected
    final newDeviceIds = devices.map((d) => d.deviceId).toSet();

    // 2. The "Clean Up"
    // If a device (the crashed client) is in our local list,
    // but NOT in the new list from the OS, we remove it.
    _connectedDevices.removeWhere((d) => !newDeviceIds.contains(d.deviceId));

    // 3. Update the others
    // ... logic to update signal strength/names for survivors ...

    notifyListeners(); // UI updates immediately
  }
}
```

2. The Result

- **Instant Feedback**: As soon as the Host's phone realizes the link is gone, the "Connected Devices" list shrinks by one. The crashed user disappears.
- **No Stuck connection State**: Since the Client was just a participant, their crash doesn't "break" the Wi-Fi group. The frequency remains open.

Rejoining (The Comeback)

When the crashed Client restarts their app:

1. **Fresh Start**: They initialize a new `P2PClient`.
2. **Discovery**: They see the Host (who never went down) immediately on the dash.
3. **Reconnect**: They tap "Connect".
    - Host sees a "New Association".
    - Host accepts.
    - Client appears on the Host's screen as a brand new entry (or updates the existing one if we kept history).

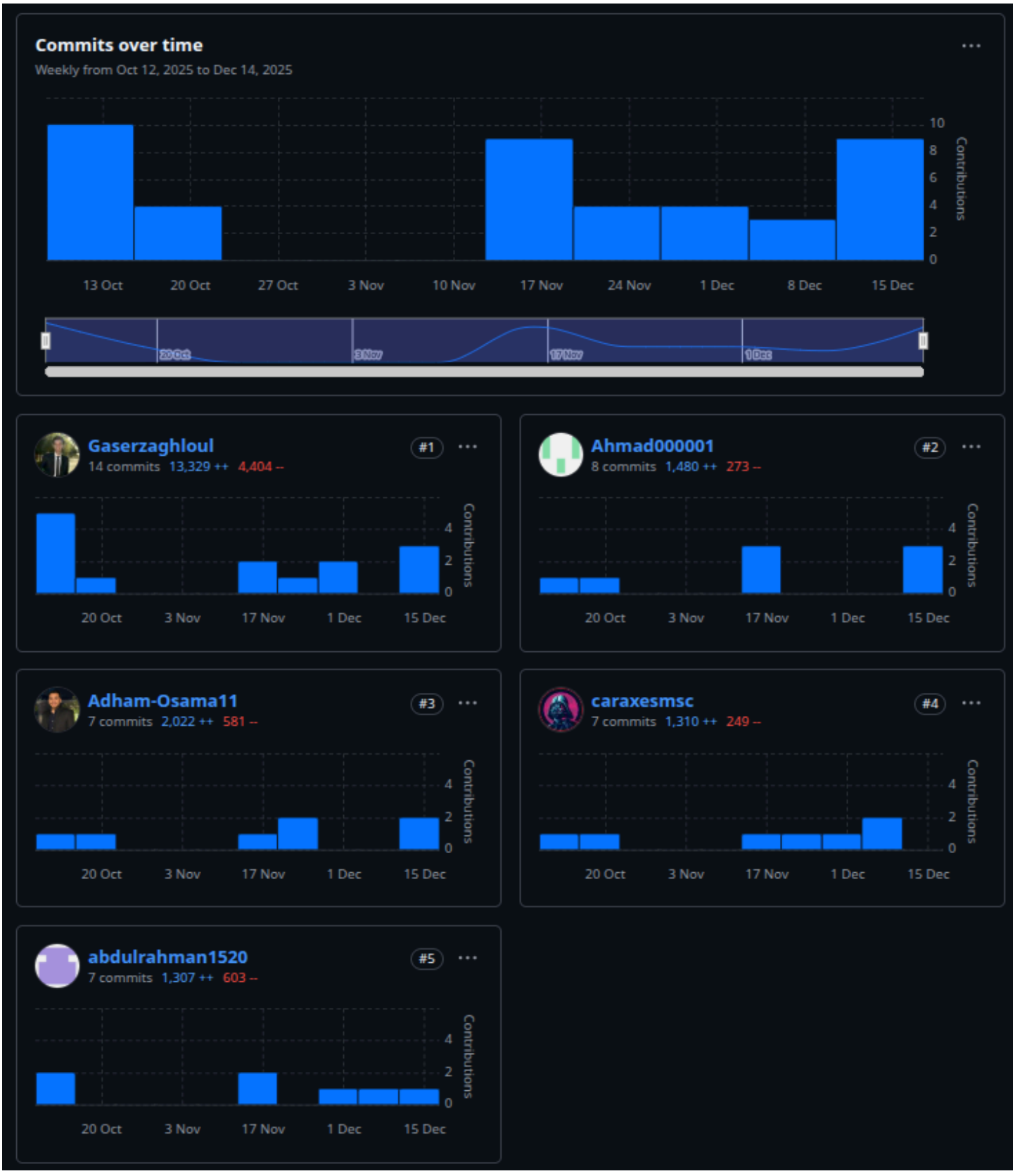
Summary

Client crashes are easier to handle because the Host (Group Owner) acts as the "Server". As long as the Server stays up, clients can come and go (crash and restart) without corrupting the network state.

Version Control

Link

https://github.com/Gaserzaghloul/BEACON_Mobile-Application



Video

https://onedrive.live.com/?qt=allmyphotos&photosData=%2Fshare%2F1F31736FE8E90BF7!s882a950845f64b73ba0a38413acb274f%3Fithint%3Dvideo%26e%3DSTNUcl%26migratedtospo%3Dtrue&cid=1F31736FE8E90BF7&id=1F31736FE8E90BF7!s882a950845f64b73ba0a38413acb274f&redeem=aHR0cHM6Ly8xZHJ2Lm1zL3YvYy8xZjMxNzM2ZmU

4ZTkwYmY3L0lRQUlsU3FJOWtWelM3b0tPRUU2eXlkUEFXamdDOWsyWVZ2MW4zeEZna0tKUnR3P2U9U1ROVWNs&v=photos