PayHook Bridge: Technical Documentation

Version: 1.0.0

Target Platform: Android 8.0 (Oreo) to Android 14+

Primary Function: Real-time Payment Interception & Webhook Forwarding

1. System Architecture Overview

PayHook Bridge acts as a **middleware agent** running on an Android device. It bridges the gap between the phone's internal communication channels (GSM/SMS, Push Notification System, and Email Protocols) and your external web server.

High-Level Data Flow

1. **Input Vectors:** The app monitors three distinct vectors:
   - **SMS:** Native Android Broadcasts (`android.provider.Telephony.SMS_RECEIVED`).
   - **Notifications:** Android System Binding (`NotificationListenerService`).
   - **Email:** IMAP Client (via `javax.mail`) polling Gmail.
2. **The Filter Engine (`FampayParser`):** Every detected message passes through a strict logic gate to determine if it is a valid **Fampay** transaction.
3. **The Network Layer (`WebhookManager`):** Validated data is serialized into JSON and POSTed to your server.
4. **The Backend (Your Server):** Receives the JSON, verifies the payment, and triggers your business logic.

2. Android Internal Logic

A. The Notification Scraper (`NotificationScraperService`)

This is the most complex component. Android treats notifications as protected data.

- **Mechanism:** The app runs a service that extends `NotificationListenerService` .
- **Permission Model:** This requires the `BIND_NOTIFICATION_LISTENER_SERVICE` permission. This is a "Signature" level permission, meaning normal apps cannot grant it to themselves. **However**, Android allows the *User* to manually grant this via the "Special App Access" settings menu.
- **Logic:**
  1. The OS notifies the service when a notification is posted.
  2. The service extracts the `package_name` (e.g., `com.fampay.in`), `title` , and `text` .
  3. It ignores ongoing notifications (like music players) and focuses on "posted" events.

B. The SMS Scraper (`SmsReceiver`)

- **Mechanism:** A `BroadcastReceiver` that listens for the system-wide `SMS_RECEIVED` broadcast.
- **Priority:** This runs instantaneously when an SMS hits the modem, even if the app is closed.

- **Logic:** It iterates through the PDUs (Protocol Data Units) of the SMS, stitches the message body together, and extracts the Sender ID (e.g., `VM-FAMPAY`).

C. The Email Scraper (`EmailForegroundService`)

- **Mechanism:** A specialized `Service` running in the foreground.
- **Protocol:** IMAP over SSL (Port 993).
- **State Management:**
  - The service maintains an open socket connection to `imap.gmail.com` .
  - It polls the `INBOX` every **15 seconds**.
  - It fetches headers strictly searching for `UNREAD` emails.
  - **Important:** Once a Fampay email is found and sent to the webhook, the app marks it as `READ` (Seen) on the server to prevent duplicate processing in the next loop.

3. The Logic Engine (`FampayParser.kt`)

The app does not send everything. It uses a "Strict Filter" to prevent spamming your server. A message is only forwarded if it meets the following criteria:

The "Sender" Check

The message source must match one of these signatures:

- **App Packages:** `com.fampay` , `com.fampay.in`
- **SMS Headers:** `FAMPAY` , `FMPAY` , `FAMAPP` , `FAMCRD` , `IDFCFB` (IDFC handles Fampay banking backend).
- **Email Senders:** Specific addresses containing "fampay" or "famcard".

The "Context" Check

If the sender is generic (like "GPay" sending a notification *about* Fampay), the body text **MUST** contain one of these strict keywords:

- `fampay` , `famcard` , `famx`

The "Transaction" Check

The message **MUST** contain a financial indicator:

- `received` , `credited` , `sent` , `debited` , `paid` , `rs.` , `inr` , `₹`

4. Persistence Strategy (How it runs 24/7)

Android aggressively kills background apps to save battery. PayHook uses four layers of defense to stay alive:

1. **Foreground Service Promotion:** The Email service creates a **Visible Notification** in the status bar ("Fampay Scraper Running"). This tells the Android Kernel that the app is "user-perceptible" and should not be killed to free up RAM.
2. `START_REDELIVER_INTENT` **:** If the system runs out of memory and *must* kill the service, this flag tells Android: *"Restart this service automatically as soon as memory is available, and give me the*

last intent back."

3. **Boot Receiver (`BootReceiver.kt`):** The app listens for `android.intent.action.BOOT_COMPLETED` . When the phone restarts, this receiver immediately launches the `EmailForegroundService` .
4. **Battery Optimization Whitelist:** The app explicitly requests `ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS` . This removes the app from Doze Mode, allowing it to use the network even when the screen has been off for hours.

5. Webhook Integration Guide (The Backend)

This is the section for your server developer. You need to create an API endpoint (e.g., using Node.js, Python/Flask, PHP, or Go).

Endpoint Requirements

- **Method:** `POST`
- **Content-Type:** `application/json`
- **Timeout:** The Android app does not wait for a response, but it expects a `200 OK` status code to confirm receipt (for future log upgrades).

JSON Payload Schema

The Android app sends the following JSON structure:

```
{
  "source": "string",
  "sender": "string",
  "message": "string",
  "timestamp": number
}
```

Field Definitions

Field	Type	Description
`source`	String	The origin of the data. Values: `SMS_FAMPAY` , `NOTIF_FAMPAY` , `EMAIL_FAMPAY` .
`sender`	String	**SMS:** The Sender ID (e.g., `VM-FMPAY`). **Notif:** Package Name (e.g., `com.fampay.in`). **Email:** Sender Address (e.g., `alerts@fampay.in`).
`message`	String	The full content. **SMS:** The message body.

		Notif: "Title: Description".
		Email: The Subject line.
timestamp	Long	Unix timestamp (milliseconds) of when the app captured the event.

Backend Implementation Examples

A. Node.js (Express)

```
const express = require('express');
const app = express();

app.use(express.json());

app.post('/webhook', (req, res) => {
    const { source, sender, message, timestamp } = req.body;

    console.log(`[${new Date(timestamp).toISOString()}] New Event from ${source}`);

    // 1. Regex to extract amount
    const amountRegex = /(?:Rs\.?|INR|₹)\s*(\d+(?:\.\d{1,2})?)/i;
    const match = message.match(amountRegex);

    if (match) {
        const amount = parseFloat(match[1]);
        console.log(`💰 Payment Detected: ₹${amount} from ${sender}`);

        // TODO: Update your database logic here
    } else {
        console.log("⚠️ Could not parse amount");
    }

    res.status(200).send('Received');
});

app.listen(3000, () => console.log('PayHook Server running on port 3000'));
```

B. Python (Flask)

```
from flask import Flask, request
import re

app = Flask(__name__)

@app.route('/webhook', methods=['POST'])
def handle_webhook():
    data = request.json

    source = data.get('source')
    message = data.get('message')

    print(f"Received from {source}: {message}")
    # Simple Parsing Logic
    amount_match = re.search(r"(?:Rs\.?|INR|₹)\s*(\d+(?:\.\d{1,2})?)", message, re.IGNO

    if amount_match:
        amount = amount_match.group(1)
        print(f"Detected Amount: {amount}")
        # Database update logic here

    return "OK", 200

if __name__ == '__main__':
    app.run(port=3000)
```