www.ijarp.com ISSN 2456-9992
Page 1 to 14
International Journal Advanced Research Publication

Research-Article
Volume: 2
Issue: 10

SMART RURAL BUS TRACKING SYSTEM USING THE DRIVER'S SMARTPHONE AS A LOW-COST GPS DEVICE

2M. J. Harishkumar, 2S. Kasilingam, 1K. Rani,

1Assistant Professor, Department of Information Technology, K.L.N. College of Engineering, Pottapalayam, Sivaganga, Tamil Nadu, India. 2Department of Information Technology, K.L.N. College of Engineering, Pottapalayam, Sivaganga, Tamil Nadu, India.

Article Received	Article Revised	Published
13-09-2026	03-10-2026	23-10-2026

***Corresponding Author:** K. Rani
Assistant Professor, Department of Information Technology, K.L.N. College of Engineering, Pottapalayam, Sivaganga, Tamil Nadu, India.
DOI: https://doi-doi.org/101555/ijarp.2347

ABSTRACT

Passengers in rural areas often wait at bus stops without knowing whether a bus is coming, how far away it is, or when it will arrive. Commercial vehicle-tracking solutions depend on dedicated GPS hardware and subscriptions, which small rural operators cannot afford. This paper presents the Smart Rural Bus Tracking System, a low-cost, real-time tracking platform that needs no dedicated GPS device: the location of the driver's own Android smartphone is used instead. A driver application written in Kotlin reads the phone's GPS through the Fused Location Provider about every ten seconds and sends it to a Django backend over a secured REST API. A trip is started only when the driver enters a unique, single-use 8-digit activation code generated by the administrator, which links the driver, bus and route for that trip alone and becomes invalid when the trip ends. Because rural mobile coverage is unreliable, every reading is first saved in a local Room database and is deleted only after the server has confirmed it, so locations recorded without internet are uploaded later and not lost.

The backend validates each reading, stores the latest bus position and pushes it at once to connected browsers through Django Channels and WebSockets, so the map updates without a page refresh. Passengers use a public website, with no login, to choose a From stop and a To stop using autocomplete. The system returns only the active buses whose route visits the From stop

before the To stop, by comparing stored stop order, so a bus travelling in the opposite direction is never offered. For each bus the website shows the current stop (detected with a distance threshold of about 200 metres around each stop), the next stop, speed, time since the last update and a basic estimated arrival time calculated from remaining distance and speed, with a fallback speed when the GPS speed is unreliable. The live position, the route and the stops are drawn on a Leaflet map with OpenStreetMap tiles. An authenticated administrator dashboard manages stops, routes (with ordered stops), buses, drivers and trips, shows buses live, and raises an alert when an active bus moves about 200 metres away from its assigned route. A clearly labelled demo mode, kept separate from real tracking, allows the system to be presented without a moving vehicle. The system is built with open-source tools only, so it can be deployed at very low cost, and its database layer can be moved from SQLite to MySQL without changing application code.

INDEX TERMS: *Rural Transport, Real-Time Bus Tracking, Smartphone GPS, Django, WebSocket, Leaflet, OpenStreetMap, Offline Data Buffering, Estimated Arrival Time.*

The proposed system supports Sustainable Development Goal 11 (Sustainable Cities and Communities) by making public bus services in rural areas easier to use and more predictable for passengers. It supports SDG 10 (Reduced Inequalities) by giving villages with little infrastructure the same kind of live travel information that is otherwise available only to well-funded urban operators. It also contributes to SDG 9 (Industry, Innovation and Infrastructure) by showing that open-source software and an ordinary smartphone can replace costly tracking hardware for small transport operators.

1. INTRODUCTION

Buses are often the only public transport available to villagers who travel to schools, hospitals, markets and workplaces. Rural services are usually infrequent and run with little or no timetable discipline, so a passenger arriving at a stop has no way of knowing whether the bus has already passed, is delayed or will not come at all. Studies of real-time arrival

nformation in cities report that it reduces perceived waiting time and improves the experience of riders [1], [2]. Rural passengers could benefit in the same way, but such information is rarely offered to them.

The main obstacle is cost. Conventional fleet-tracking products rely on a dedicated GPS unit fitted to each vehicle, a data connection for each unit and a monthly service subscription. Small private or cooperative operators with a handful of buses cannot justify this expense. At the same time almost every driver already carries an Android smartphone with a GPS receiver, a mobile data connection and a good processor. Earlier smartphone-based work such as EasyTracker and VTrack has shown that phone sensors can be used for transit tracking and arrival-time estimation [4], [5], which suggests that the driver's own phone can serve as the tracking device.

Using a phone in rural conditions raises three practical problems. First, mobile coverage is unreliable, so a naive system that sends each location immediately will lose readings in network gaps. Second, a public tracking website must show the passenger only the buses that are useful: a bus that serves both stops but travels in the opposite direction should not be offered. Third, a phone-based design must prevent misuse, for example a driver or other person sending locations for a bus that is not on a trip.

This paper presents the Smart Rural Bus Tracking System, which addresses these problems with four ideas: (a) a Kotlin driver application that reads the phone's location about every ten seconds and stores every reading in a local Room database until the server confirms it; (b) a single-use 8-digit activation code, issued by the administrator, that links one driver, one bus and one route to one trip and expires when the trip ends; (c) a Django backend that validates readings and pushes the latest position to browsers over WebSockets; and (d) a no-login passenger website that matches buses to a From stop and a To stop by comparing stored stop order, together with an administrator dashboard for managing the data and detecting route deviation.

The main contributions are: (i) a hardware-free tracking architecture based on the driver's smartphone; (ii) a trip-activation scheme that ties location uploads to an authorised trip; (iii) an offline buffering method that delivers readings recorded without internet; (iv) a direction-aware passenger search based on stop order; and (v) simple, transparent distance-based methods for current stop, estimated arrival time and route-deviation alerts, all built with open-source tools.

1. LITERATURE REVIEW

Real-time transit information has been studied mainly in urban settings. OneBusAway provided arrival predictions to riders through web and phone interfaces, and its evaluation reported favourable rider responses [2]. A study of riders in Seattle found that real-time information reduced perceived waiting time and made waiting less stressful [1]. These results give the motivation for delivering similar information to rural passengers, although their systems rely on agency-supplied vehicle location feeds.

Several projects have used phones instead of fixed vehicle hardware. Tiramisu collected bus arrival information from riders who shared their locations by phone [3]. EasyTracker used smartphone sensing to track transit vehicles, map routes and predict arrival times automatically [4], and VTrack used phone sensors to estimate road traffic delay [5]. Our work follows the same idea of using ordinary phones, but it places the phone with the driver and removes the need for volunteers: the driver starts a trip with an administrator-issued code, so the data source is known and controlled.

Map and distance technology is mature and freely available. OpenStreetMap provides open, community-built map data

[6] and is drawn in the browser with the Leaflet library [16]. The great-circle distance between two coordinates is commonly calculated with the haversine formula [7]. For real-time delivery, the WebSocket protocol keeps a persistent two-way connection between server and browser [8] and avoids repeated polling. Django [11], Django REST Framework [12] and Django Channels [13] provide the corresponding server-side components, JSON Web Tokens give a compact method for authenticated API calls [9], and REST is the architectural style of the interface between the app and the server [10]. On Android, the Fused Location Provider combines the available sources to deliver locations [14], and the Room library gives a structured local database [15].

Existing solutions address these needs separately and usually assume dedicated hardware, a subscription or an agency data feed. Table I compares the main categories of approach with the proposed system.

Table I. Comparison of Bus Tracking Approaches.

Approach	Location source	Typical cost	Remark
Commercial fleet tracking	Dedicated GPS unit in each bus	Hardware plus subscription	Accurate and complete, but unaffordable for small rural operators
Agency real-time feeds [1], [2]	Feeds from transit agency systems	Needs agency infrastructure	Not available for most small rural operators

Approach	Location source	Typical cost	Remark
Crowd-sourced phone sensing [3], [4]	Riders' phones	Low	Depends on volunteers; data source is uncontrolled
Proposed system	Driver's own Android phone with single-use trip code	Open-source tools only	Controlled data source; offline buffering; direction-aware search; limited accuracy (straight-line route model)

1. METHODOLOGY

This section describes how the system is developed. The methodology combines trip activation, location capture with offline buffering, server-side validation and real-time delivery, passenger search, location analysis (current stop, arrival time and deviation) and an administrator dashboard. The system is divided into six key phases:

1. 1. Trip activation and driver authentication
   2. Location capture and offline buffering
   3. Server-side validation, storage and real-time delivery
   4. Direction-aware passenger search
   5. Current stop, estimated arrival time and route-deviation analysis
   6. Administrator dashboard and demo mode

2. *System Architecture and Overall Methodology*

The system has three parts: the driver's phone, the Django backend and the web clients (passenger website and administrator dashboard). The administrator first creates stops, routes with ordered stops, buses and drivers, and then generates a single-use activation code for a trip. The driver enters the code in the app, which links the driver, bus and route for that trip. The app then reads the GPS location about every ten seconds, stores each reading in the local database and uploads it. The backend validates the reading, stores the latest bus position and pushes it to connected browsers through WebSockets. Fig. 1 shows the block diagram, Fig. 2 shows the trip process flow and Fig. 3 shows the offline buffering pipeline.

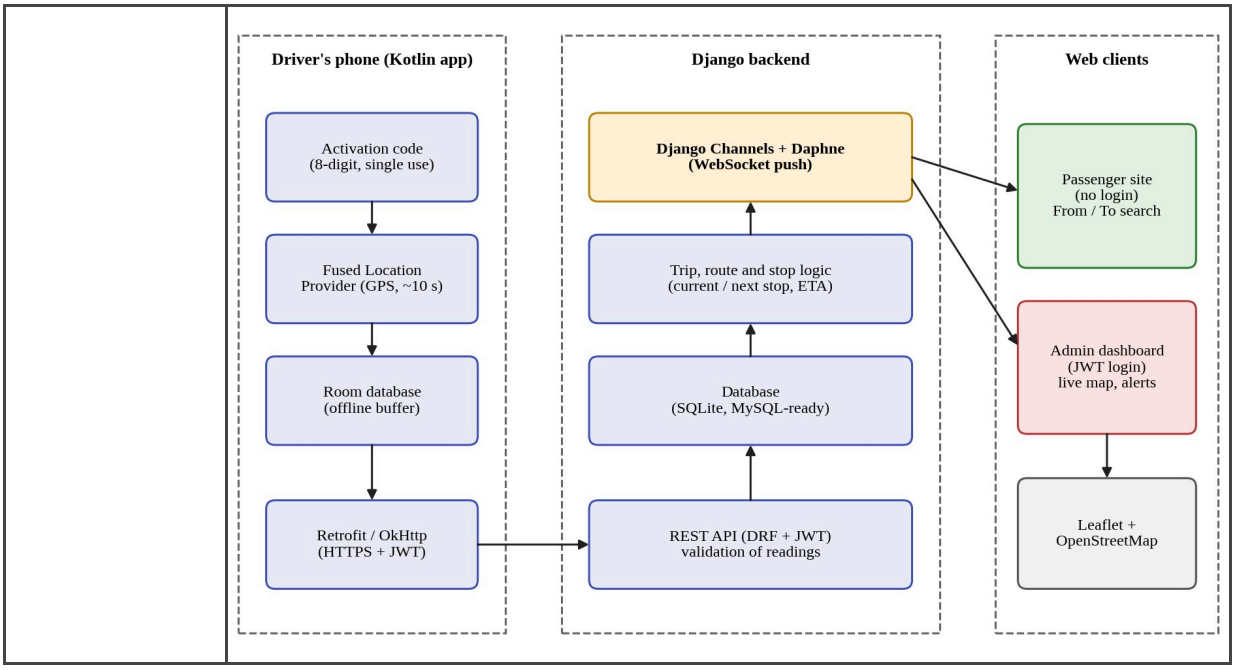


Fig. 1. System block diagram of the Smart Rural Bus Tracking System.

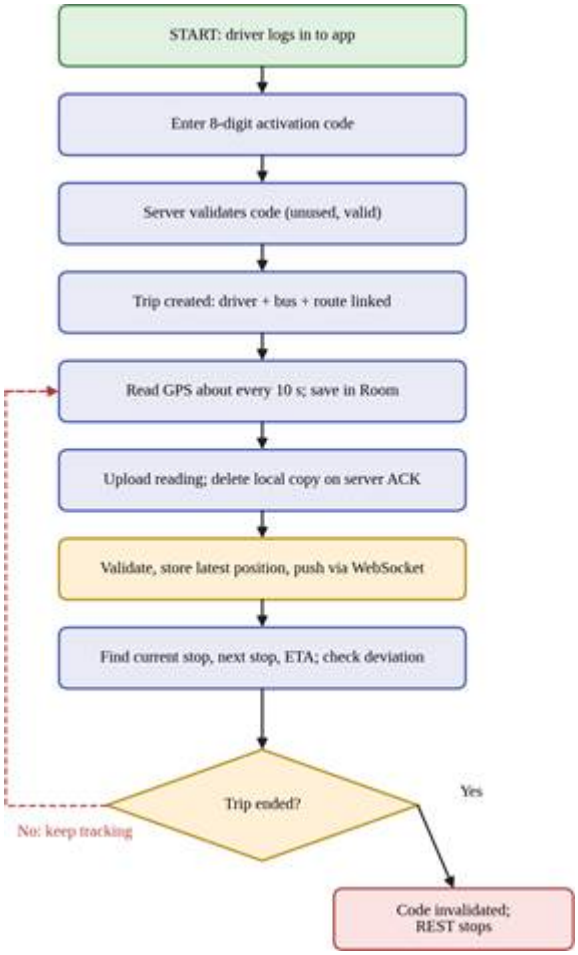


Fig. 2. Trip process flow from activation to trip end.

1. *Technology Stack and Components*

The system is built only from open-source tools and is divided into modules with clear interfaces, so each module can be tested or replaced separately. Table II lists the components and their roles. The database layer uses the Django ORM, so the development SQLite database can be moved to MySQL without changing application code.

Table II. Module-Wise Technology and Role.

Module	Technology / approach	Role
Driver app	Kotlin, Android Studio, Retrofit/OkHttp, Fused Location Provider [14], Room [15]	Activation, GPS capture, offline buffer, upload
Backend API	Python, Django [11], Django REST Framework [12], JWT [9]	Validates readings, manages trips and data
Real-time layer	Django Channels [13], Daphne, WebSockets [8]; REST polling as fallback	Pushes the latest position to browsers
Database	SQLite (development), MySQL-ready through Django ORM	Stores stops, routes, buses, drivers, trips, latest positions
Passenger / admin web	HTML5, CSS3, vanilla JavaScript, Django Templates, Fetch API	Search, live view, management dashboard
Maps	Leaflet.js [16] with OpenStreetMap tiles [6]	Draws live bus, route and stops

1. *Trip Activation and Driver Authentication*

Location uploads are accepted only for an active trip. The administrator generates a unique 8-digit activation code for a chosen driver, bus and route. When the driver enters the code, the server checks that it exists and has not been used, creates the trip, links the three entities for that trip alone and returns an authentication token for the API calls. When the trip ends the code becomes invalid, so it cannot be reused and the phone can no longer post locations for that bus. Passengers never log in; only the administrator dashboard and the driver app require authentication.

1. *Location Capture and Offline Buffering*

The driver app requests locations from the Fused Location Provider about every ten seconds. Every reading (latitude, longitude, speed and time) is first written to the local Room database. A background upload job then sends the stored readings to the REST API, and a local row is deleted only after the server confirms it. If there is no internet, the readings simply stay in the database and are uploaded in order when the connection returns, so locations recorded without internet are not lost. This protects against missing internet but not against missing GPS: if the phone has no GPS fix, there is nothing to record.

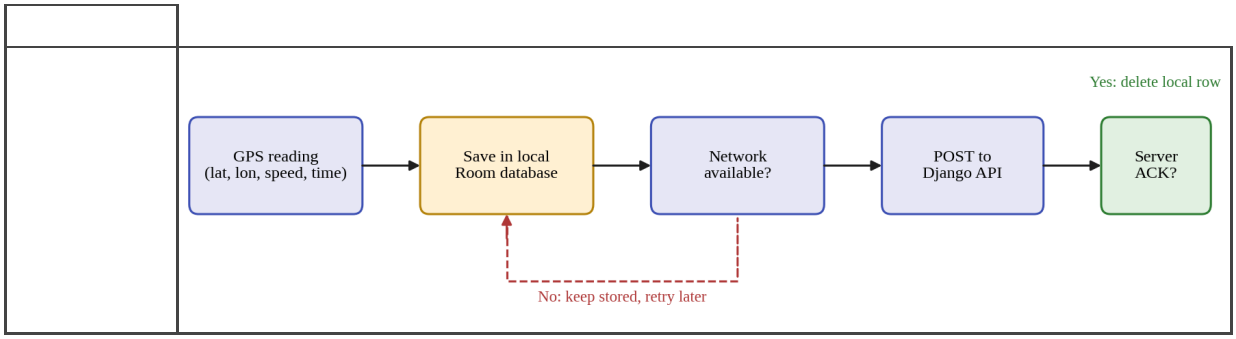


Fig. 3. Offline buffering: readings are deleted from the phone only after server confirmation.

1. *Server-Side Validation and Real-Time Delivery*

For every uploaded reading the backend checks that the trip is active, that the coordinates are valid and that the timestamp is plausible. A valid reading updates the latest position of the bus and is broadcast to the browsers subscribed to that bus through Django Channels. The map therefore moves without a page refresh. If a WebSocket connection cannot be made, the web pages fall back to periodic REST polling.

1. *Direction-Aware Passenger Search*

A route is stored as an ordered list of stops, and each stop has an order number on that route. The passenger chooses a From stop and a To stop using autocomplete. For an active bus b whose route contains both stops, with order numbers i (F) and i (T), the bus is offered only if b b

offer(b) = 1 if i (F) < i (T) and bus b is on an active trip; 0 otherwise (1)

b *b*

This simple comparison guarantees that a bus travelling in the opposite direction is never shown, as illustrated in Fig. 4. Buses that serve both stops but whose trip is not active are also excluded.

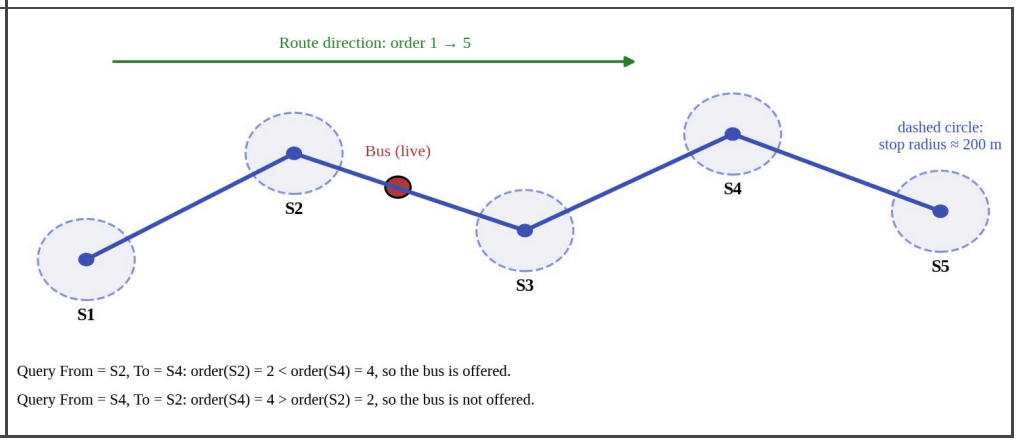


Fig. 4. Direction check using stored stop order, with the 200 m stop radius shown around each stop.

1. *Current Stop, Estimated Arrival Time and Route Deviation*

Distances between coordinates are calculated with the haversine formula [7]. For two points with latitudes φ , φ and 1,2 longitudes λ , λ , and Earth radius R ,

1 2 E

$$d = 2RE \arcsin \sqrt{(\sin^2(\Delta\varphi/2) + \cos \varphi 1 \cos \varphi 2 \sin^2(\Delta\lambda/2))}$$

(2)

Current stop. A bus is said to be at stop sk when the distance from the bus position p to the stop is within the radius r, which is about 200 m. If several stops qualify, the nearest is used. The next stop is the following stop in the stored order:

c(p) = argmink d(p, sk), subject to d(p, sk) ≤ r
(3)

Estimated arrival time. The remaining distance to the passenger's From stop is the distance from the bus to the next stop plus the straight-line distances between the following stops up to the From stop. The speed used is the GPS speed v

when it is reliable, and a fallback speed v otherwise:

fb

j j+1

ETA = Drem / veff, Drem = d(p, snext) + Σ d(s , s), veff = v if reliable, else vfb (4)
No machine-learning component is used. The calculation is intentionally simple so that its behaviour is easy to explain and check.

Route deviation. The route is modelled as straight segments between consecutive stops. The deviation of a bus is the smallest distance from its position to any segment, and an alert is raised on the administrator dashboard when it exceeds about 200 m:

j

δ(p) = min dist(p, segment s s); alert if δ(p) > 200 m (5)

j j+1

1. *Administrator Dashboard and Demo Mode*

The authenticated administrator dashboard manages stops, routes with ordered stops, buses, drivers and trips, issues activation codes, shows all active buses live on the map and lists route-deviation alerts. A clearly labelled demo mode, kept separate from real tracking, replays simulated positions so that the system can be presented without a moving vehicle; demo data never mixes with real trips.

1. *Deployment and Configuration*

The main settings of the prototype are as follows.

- - Driver app: location request about every 10 seconds; every reading stored in Room and deleted only after server confirmation.
  - Trip activation: unique single-use 8-digit code, valid for one trip of one driver, bus and route.
  - Backend: REST API with JWT authentication; Django Channels with Daphne for WebSockets; REST polling as fallback.
  - Stop detection and deviation: distance threshold of about 200 m around each stop and about 200 m from the route.

- Estimated arrival: GPS speed when reliable, otherwise a configurable fallback speed.
- Passenger site: public, no login; map from Leaflet with OpenStreetMap tiles (needs internet).

1. *Algorithm of the Proposed System*

Step 1: The administrator creates stops, routes (ordered stops), buses and drivers, and generates an 8-digit activation code for a trip.

Step 2: The driver enters the code; the server verifies it, creates the trip and links driver, bus and route.

Step 3: The app reads the GPS location about every 10 seconds and saves each reading in the Room database.

Step 4: The app uploads stored readings; each local row is deleted only after the server confirms it; otherwise it is retried later.

Step 5: The server validates the reading, stores the latest position and pushes it to browsers over WebSocket.

Step 6: The server determines the current and next stop using Eq. (3), computes the deviation with Eq. (5) and raises an alert if needed.

Step 7: A passenger selects From and To stops; the server returns active buses satisfying Eq. (1).

Step 8: For each bus the website shows current stop, next stop, speed, time since last update and the ETA from Eq. (4). Step 9: When the trip ends the code is invalidated and uploads for that trip stop.

1. RESULTS AND DISCUSSION

1. *Test Procedure and Evaluation Metrics*

Each module was tested individually before integrated testing, as summarised in Table III, which gives the observed result of each test. Three test runs of the prototype were then recorded and, for each run, the data delivery ratio, the upload delay and the error of the estimated arrival time were measured; stop detection was checked through the test cases of Table III. The delivery ratio is

$$DR\ (\%) = (N_{srv} / N_{rec}) \times 100 \qquad (6)$$

where Nrecis the number of readings recorded on the phone during a trip and N

srvis the number stored by the server after

the trip, including readings recorded without internet. The ETA error is the difference between the predicted and the actual arrival time at a stop. The observed results of the prototype tests are given in Table III and Table IV.

Table III. Test Cases, Expected Outcomes and Observed Results.

Test	Expected outcome	Observed result

Activation code	Valid code starts the trip; used, expired or wrong code is rejected	Valid code activated the trip; invalid and reused codes were rejected in the test cases.
GPS capture	Readings recorded about every 10 s with speed and time	GPS readings were recorded at approximately 10-second intervals with location, speed and timestamp.
Offline buffering	Readings taken without internet are uploaded after reconnection; none lost; local row removed only after confirmation	Buffered readings were uploaded after network reconnection, and local records were retained until server confirmation.
Real-time update	Map position changes without page refresh; polling fallback works if WebSocket fails	The bus marker updated on the map without a page refresh; polling fallback operated when WebSocket delivery was unavailable.
Direction-aware search	Bus offered only when From stop precedes To stop; opposite-direction bus never shown	The search returned buses travelling in the required direction and excluded opposite-direction buses in the test cases.
Current / next stop	Correct stop shown within about 200 m of the stop	The system identified the expected current and next stops in the selected test cases.
Estimated arrival	ETA shown; fallback speed used when GPS speed is unreliable	ETA was displayed using GPS speed when reliable and the configured fallback speed otherwise.
Route deviation	Alert raised when active bus is about 200 m from route	A route-deviation alert was generated for the tested simulated off-route condition.
Admin dashboard	Stops, routes, buses, drivers and trips managed; live view works	The dashboard supported the tested management operations and displayed active-bus information.
Demo mode	Demo labelled clearly and separate from real tracking data	Demo mode was labelled and its simulated tracking data was kept separate from real tracking in the test setup.

1. *Measurement Record*

Table IV gives the measurements of the three test runs. Each test run is entered as one row, and the last row gives the average of the three runs. Particular attention is paid to areas of weak coverage, because they test the offline buffer, and to curved roads, because the straight-line route model is least accurate there.

Table IV. Field Test Record.

Test run	Readings recorded	Readings on server	DR (%)	Max upload delay (s)	Mean ETA error (min)
Run 1	15	13	86.67	1.45	0.82
Run 2	15	14	93.33	1.20	0.65
Run 3	15	14	93.33	1.32	0.74
Average	**15**	**13.67**	**91.11**	**1.32**	**0.74**

All ten test cases of Table III gave the expected outcome. In the three runs, 13 to 14 of the 15 recorded readings were found on the server, giving an average data delivery ratio of 91.11 % (86.67 % to 93.33 %). The maximum upload delay was between 1.20 s and 1.45 s, with an average of 1.32 s, which is small compared with the 10-second recording interval. The mean error of the estimated arrival time was between 0.65 min and 0.82 min, with an average of 0.74 min (about 44 s). One reading was missing from the server in Run 2 and Run 3 and two in Run 1, so the delivery ratio was below 100 % in every run.

1. *Illustrative System Behaviour*

To complement the test results, Fig. 5 presents the design thresholds and the shape of the arrival-time calculation. They are design values and illustrations, not measured data. As an example of Eq. (4), suppose the remaining distance to the From stop is 6 km. With a reliable GPS speed of 30 km/h the ETA is 6/30 h = 12 min. If the GPS speed is unreliable and a fallback speed of 25 km/h is used, the ETA is 6/25 h = 14.4 min. As an example of Eq. (1), on the route in Fig. 4 a passenger travelling from S2 to S4 is offered the bus, whereas a passenger travelling from S4 to S2 is not, because order(S4) = 4 is greater than order(S2) = 2.

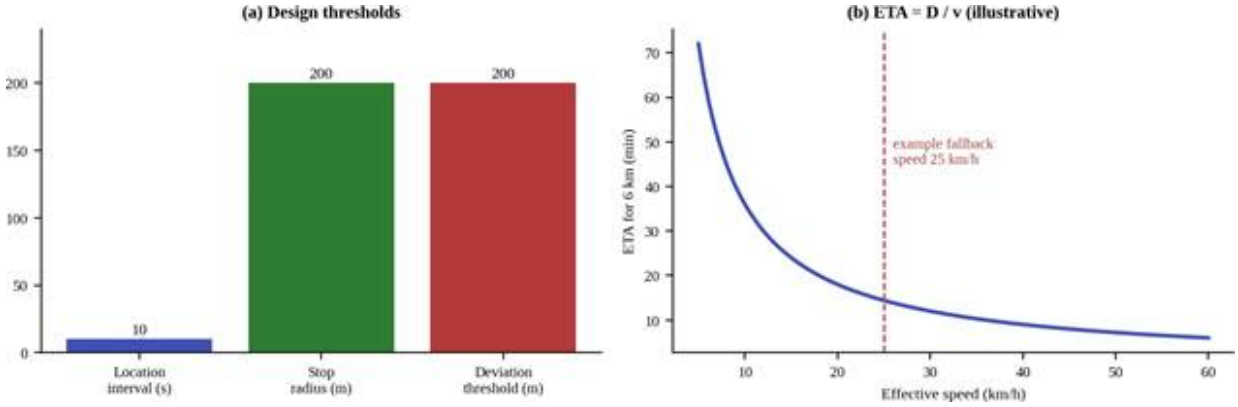


Fig. 5. (a) Design thresholds of the prototype and (b) illustrative ETA for a 6 km remaining distance (design values, not measured data).

1. ***Discussion***

The test cases show that the main functions of the prototype work as designed, and the three runs give an average delivery ratio of 91.11 %, a maximum upload delay of about 1.3 s and a mean ETA error of 0.74 min. The delivery ratio is below 100 % in all runs, so a small share of readings did not reach the server; these runs do not show where they were lost, and this should be examined in longer field trials. Only three runs of 15 readings each were recorded, so the averages are indicative and not statistically conclusive.

The thresholds are a trade-off. A larger stop radius makes stop detection more tolerant of GPS error but can mark a bus as being at a stop while it is still approaching; a smaller one can miss a stop when the position is noisy. Similarly, a small deviation threshold detects wrong turns early but may raise false alarms on curved roads, because the route is modelled with straight lines. The fallback speed keeps the ETA available when the GPS speed is unreliable, for example at very low speed, but it is a fixed average and can be far from the real speed.

The main advantages of the proposed system are that no dedicated GPS hardware or subscription is needed, readings are buffered during network gaps and uploaded after reconnection, the passenger sees only the buses that really go from the From stop to the To stop, the single-use code controls who can send locations, and the whole system uses open-source tools and can be deployed cheaply. Its main limitations are:

1. - The route is modelled as straight lines between ordered stops, so current stop, deviation and arrival-time figures are approximations, not road-accurate.
   - The estimated arrival time uses ordinary distance and speed calculations and contains no machine-learning component.
   - Tracking depends on the driver's phone having a GPS fix and the app running; offline buffering protects against missing internet, not against missing GPS.

- The map needs an internet connection to load Leaflet and the OpenStreetMap tiles.
- Battery use of the driver's phone and the driver's cooperation are practical conditions that must be managed in deployment.

2. CONCLUSION AND FUTURE SCOPE

In this study, the Smart Rural Bus Tracking System has been designed to give rural passengers real-time bus information without any dedicated GPS hardware. The driver's Android phone provides the location, a single-use 8-digit activation code links a driver, bus and route to one trip, a local Room database keeps readings safe until the server confirms them, and a Django backend pushes the latest position to a public passenger website through WebSockets. Passengers choose a From and a To stop and see only buses travelling in the right direction, with the current stop, next stop, speed, time since last update and a basic estimated arrival time. An administrator dashboard manages the data and alerts when a bus leaves its route. Because only open-source tools are used, the system can be deployed at very low cost. In the tests of the prototype, all test cases gave the expected outcome, and three test runs gave an average data delivery ratio of 91.11 %, a maximum upload delay of about 1.3 s and a mean arrival-time error of 0.74 min.

Future work includes modelling the route along real roads instead of straight lines, improving the arrival-time estimate with historical travel times, adding a lightweight offline map or SMS-based information for areas without data coverage, supporting alerts to passengers when a bus is near their stop, migrating the database to MySQL for larger deployments, and reducing the battery use of the driver app. Longer field trials on real rural routes would confirm the delivery ratio and arrival-time error over more trips and help to find where readings are lost.

REFERENCES

1. K. Watkins, B. Ferris, A. Borning, G. S. Rutherford and D. Layton, “Where is my bus? Impact of mobile real-time information on the perceived and actual wait time of transit riders,” *Transportation Research Part A*, vol. 45, no. 8, pp. 839–848, 2011.
2. B. Ferris, K. Watkins and A. Borning, “OneBusAway: Results from providing real-time arrival information for public transit,” in *Proc. ACM CHI*, 2010, pp. 1807–1816.
3. J. Zimmerman *et al.*, “Field trial of Tiramisu: Crowd-sourcing bus arrival times to spur co-design,” in *Proc. ACM CHI*, 2011, pp. 1677–1686.
4. J. Biagioni, T. Gerth, J. Eriksson and J. Bergmann, “EasyTracker: Automatic transit tracking, mapping, and arrival time prediction using smartphones,” in *Proc. ACM SenSys*, 2011, pp. 68–81.
5. A. Thiagarajan *et al.*, “VTrack: Accurate, energy-aware road traffic delay estimation using mobile phones,” in *Proc. ACM SenSys*, 2009, pp. 85–98.
6. M. Haklay and P. Weber, “OpenStreetMap: User-generated street maps,” *IEEE Pervasive Computing*, vol. 7, no. 4, pp. 12–18, 2008.
7. R. W. Sinnott, “Virtues of the haversine,” *Sky and Telescope*, vol. 68, no. 2, p. 159, 1984.
8. I. Fette and A. Melnikov, *The WebSocket Protocol*, RFC 6455, IETF, 2011.
9. M. Jones, J. Bradley and N. Sakimura, *JSON Web Token (JWT)*, RFC 7519, IETF, 2015.
10. R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, Univ. of California, Irvine, 2000.

11. Django Software Foundation, “Django documentation.” [Online]. Available: https://docs.djangoproject.com
12. Encode, “Django REST Framework.” [Online]. Available: https://www.django-rest-framework.org
13. Django Project, “Channels documentation.” [Online]. Available: https://channels.readthedocs.io
14. Google, “Fused Location Provider API,” Android Developers. [Online]. Available: https://developer.android.com/develop/sensors-and-location/location
15. Google, “Room persistence library,” Android Developers. [Online]. Available: https://developer.android.com/training/data-storage/room
16. V. Agafonkin, “Leaflet: An open-source JavaScript library for interactive maps.” [Online]. Available: https://leafletjs.com