Design and Architecture Specification for the Wi-Fi Steering Engine

Team 15: System Architecture Team

November 2025

Contents

1. Introduction

This document defines the design and architecture of the Wi-Fi Steering Engine. It serves as a formal technical reference for how the system collects link and beacon measurements, computes real-time Quality-of-Experience (QoE) scores, ranks neighboring access points, and triggers automated roaming decisions using IEEE 802.11v BSS Transition Management (BSS-TM).

1.1 System Purpose

The Wi-Fi Steering Engine addresses the problem of suboptimal client connectivity in dense wireless deployments. Clients often remain associated with access points that provide degraded service due to distance, interference, or load, even when better alternatives exist nearby. The engine solves this by continuously monitoring station conditions and proactively steering devices to better-performing access points.

The system operates as a closed-loop control system: measurements feed into analytics, analytics inform decisions, and decisions trigger actions through the AP controller. All components operate independently but coordinate through shared data stores.

1.2 Architectural Components

The architecture combines several independent subsystems connected through shared in-memory databases and scheduled processing loops:

- **Measurement Generation**: Link Measurement, Beacon Measurement, and station fetch operations driven by periodic schedulers.
- **Metrics Layer**: Computes QoE and Neighbor Ranking from incoming measurements.
- **Transition Management Engine**: Determines when and where to steer stations based on QoE thresholds and ranked neighbor lists.
- **Controller Interface**: Issues 802.11v commands to the AP.
- **StateAPI**: An HTTP server exporting station metrics with anonymized identifiers.
- **Dashboards**: Real-time monitoring of all subsystems, multiplexed through SocketPool.
- **Storage Layer**: In-memory DBs with periodic snapshotting.

Each subsystem operates independently but exchanges data via the databases and message interfaces. All schedulers run as isolated threads with strict intervals to ensure predictable timing and avoid race conditions.

1.3 Design Goals

The primary design goal is to maintain stable, high-quality client connectivity by continuously evaluating per-station conditions and steering devices toward better-performing access points when needed. The system must operate with minimal latency, handle hundreds of concurrent stations, and expose detailed state information for monitoring and debugging.

One capability remains partially implemented: post-roam QoE delta analysis. The system does not yet factor the improvement...or degradation...after a steering event back into its decision logic. This limits adaptive learning but is planned for a future release.

2. System Architecture Overview

2.1 Component Organization

The system is organized into functional layers that process data in a pipeline fashion. Raw measurements enter at the bottom layer, flow through analytics processing, and emerge as steering decisions at the top layer. This unidirectional flow simplifies reasoning about system behavior and enables independent testing of each layer.

2.1.1 Measurement Layer

This layer handles all data acquisition from the wireless infrastructure. It consists of three primary mechanisms:

1. **Link Measurement**: Issues IEEE 802.11k Link Measurement requests to stations and processes the resulting reports. These reports contain transmit power, link margin, and RSSI data that characterize the current connection quality.

2. **Beacon Measurement**: Requests 802.11k Beacon Reports from clients, which contain per-neighbor RSSI values, channel utilization, and station counts. This data enables the system to understand what alternative APs are visible to each client.

3. **Station Statistics**: Polls the AP controller for per-station counters including TX/RX byte counts, retry rates, FCS error rates, inactive time, and current bitrates. These statistics provide insight into throughput, reliability, and activity levels.

All measurement operations are triggered by periodic schedulers rather than events. This ensures consistent timing and prevents measurement floods when many stations connect simultaneously.

2.1.2 Analytics Layer

The analytics layer transforms raw measurements into actionable metrics. It implements two primary computations:

1. **QoE Computation**: Combines signal strength, throughput, reliability, latency, and activity metrics into a single normalized quality score between 0 and 1. The computation uses weighted averaging with configurable weights that reflect the relative importance of each dimension.

2. **Neighbor Ranking**: Processes beacon measurement reports to produce a sorted list of candidate APs for each station. The ranking considers neighbor RSSI, estimated capacity, and channel load to identify the best steering targets.

Both computations maintain historical data to detect trends and volatility. Moving averages smooth out short-term fluctuations, while linear regression slopes indicate whether conditions are improving or degrading over time.

2.1.3 Decision Layer

The decision layer implements the steering policy. The Transition Management Engine examines each station's current QoE score and, if it falls below the configured threshold (0.55), constructs a BSS Transition Management request containing the ranked list of candidate APs.

The engine applies several guards before steering:

- Stations with QoE above threshold are not steered
- Stations without ranked neighbors cannot be steered
- Recent steering events are rate-limited to prevent oscillation

2.1.4 Control Layer

The control layer translates steering decisions into IEEE 802.11v BSS-TM frames and sends them through the AP controller. It handles the vendor-specific details of controller communication while presenting a uniform interface to the decision layer.

The `BssTmRequestBuilder` constructs properly formatted BSS-TM requests with the `PREFERRED_CAND_LIST_INCLUDED` mode, which includes the ranked neighbor list. The client firmware uses this list to select its next AP.

2.1.5 Management Layer

The management layer provides observability and persistence:

- **StateAPI** exposes current system state via HTTP JSON endpoints with anonymized station identifiers
- **Dashboards** provide real-time WebSocket feeds of measurement reports, QoE updates, and steering events
- **Snapshot Manager** periodically serializes all in-memory databases to disk for recovery after restart

2.2 Data Flow

Data flows through the system in a strict pipeline:

1. Schedulers trigger measurement requests at fixed intervals
2. Measurement reports arrive and are stored in the measurement database
3. Analytics schedulers periodically read measurements and compute QoE/ranking
4. Computed metrics are stored in dedicated databases
5. The transition management scheduler reads metrics and makes steering decisions
6. Steering commands are sent to the controller
7. All state changes are visible through the StateAPI and dashboards

This pipeline architecture ensures that each stage operates on consistent snapshots of data without worrying about concurrent modifications. Databases implement versioning to support lock-free reads while schedulers update data.

2.3 Threading Model

The system uses one thread per major scheduler to ensure timing independence:

- Link Measurement Scheduler thread
- Beacon Measurement Scheduler thread
- QoE Computation Scheduler thread
- Neighbor Ranking Scheduler thread
- BSS Transition Scheduler thread

Additional threads handle HTTP requests (StateAPI), WebSocket connections (dashboards), and controller communication. All inter-thread communication occurs through the shared databases, which use locking primitives to ensure consistency.

3. Schedulers and Data Acquisition

3.1 Scheduler Framework

All schedulers follow the same execution pattern: wake at fixed intervals, fetch relevant data, perform computation, store results, and return to sleep. This deterministic timing simplifies debugging and makes system behavior predictable.

Each scheduler maintains internal state including:

- Last execution timestamp
- Execution count
- Error counters
- Performance metrics (min/max/avg execution time)

3.2 Link Measurement Scheduler

The Link Measurement Scheduler maintains up-to-date connection quality data for all stations. It operates as follows:

1. Fetch the list of currently connected stations from the controller
2. For each station, check if a recent link measurement exists in the database
3. If no recent measurement exists (or the last one is stale), construct a Link Measurement request:

```
req_link_measurement(LinkMeasurementCommandBuilder(station.mac))
```

4. Send the request through the controller interface
5. Mark the request timestamp to avoid duplicate requests

The measurement report processing happens asynchronously. When a station responds with an 802.11k Link Measurement Report frame, the controller forwards it to the engine, which parses the frame and extracts:

- RSSI (Received Signal Strength Indicator)
- Transmit power used by the station
- Link margin indicating fade margin before packet loss
- Receive antenna configuration

These values are stored in the Link Measurement database with a timestamp. The QoE computation scheduler later reads these values when calculating signal quality scores.

3.3 Beacon Measurement Scheduler

The Beacon Measurement Scheduler gathers information about what APs each station can see and how strong those signals are. This data is essential for building ranked neighbor lists.

The scheduler issues 802.11k Beacon Report requests using:

```
RequestBeaconCommandBuilder(station_mac)
    .set_measurement_params(operating_class, channel, duration)
```

Key parameters include:

- **Operating Class**: Defines the frequency band (2.4 GHz, 5 GHz, etc.)
- **Channel**: Specific channel to measure, or 255 for all channels
- **Duration**: How long to spend measuring (in 802.11 Time Units)

When a station responds, the beacon report contains measurements for multiple neighboring APs. Each neighbor entry includes:

- BSSID (MAC address of the AP)
- RCPI (Received Channel Power Indicator) - equivalent to RSSI
- RSNI (Received Signal-to-Noise Indicator)
- Channel utilization
- Number of associated stations

These reports are stored in the Beacon Measurement database, aggregated across multiple measurement cycles to reduce variance from temporary conditions.

3.4 QoE Scheduler

The QoE Scheduler is responsible for periodically recomputing quality scores for all active stations. It executes at the shortest interval (typically every 5 seconds) to ensure scores reflect current conditions.

The execution sequence is:

```
QoE.update()
```

This method:

1. Fetches recent link measurements for all stations
2. Fetches station statistics (throughput, retry rates, etc.)
3. For each station with sufficient data, calls the QoE computation function
4. Stores the computed QoE score and its component breakdown
5. Updates the historical model (moving average, trend, volatility)

The historical model tracks QoE evolution over time. It maintains:

- A sliding window of the last N QoE samples (typically N=10)
- A moving average computed as: $\bar{Q} = \frac{1}{N} \sum_{i=1}^{N} Q_i$
- A trend value computed via linear regression: trend = slope$(Q_1, Q_2, \ldots, Q_N)$
- Volatility computed as sample variance: $\sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (Q_i - \bar{Q})^2$

These metrics help distinguish temporary fluctuations from genuine degradation. A station with high volatility may not need steering, while one with consistently low QoE or negative trend is a strong steering candidate.

3.5 Neighbor Ranking Scheduler

The Neighbor Ranking Scheduler processes beacon measurement reports to build sorted candidate lists for steering. It executes:

```
NeighborRanking.update()
```

For each station, this method:

1. Retrieves all beacon reports from the last measurement window
2. Groups measurements by neighbor BSSID
3. Computes aggregate statistics (average RSSI, load, etc.)
4. Applies the neighbor ranking algorithm (detailed in Chapter 5)
5. Stores the sorted neighbor list in the ranking database

The ranking database maps each station MAC to its current ranked neighbor list. This list is referenced by the Transition Management Engine when constructing steering requests.

3.6 BSS Transition Scheduler

The BSS Transition Scheduler implements the steering policy by invoking:

```
TransitionManagementEngine.run()
```

This method iterates over all connected stations and applies the steering decision logic described in Chapter 6. It runs at the longest interval (typically 60 seconds) to avoid excessive steering and allow time for roaming to complete.

4. Quality of Experience (QoE) Computation

Quality of Experience quantifies how well the network serves the client across two independent dimensions:

- **AP-side QoE ($Q_{\mathbf{AP}}$)**: Measures wireless link quality based on physical and MAC-layer metrics.
- **Transport-layer QoE ($Q_{\mathbf{transport}}$)**: Measures end-to-end performance using TCP RTT, retransmissions, and throughput.

These two QoEs are **computed independently**. Each captures different aspects of the client experience, and together they provide a complete view of both RF conditions and transport-layer behavior.

4.1 AP-Side QoE

AP-side QoE, denoted Q_{AP}, is computed from five normalized components:

- S: Signal quality score
- T: Throughput score
- R: Reliability score
- L: Latency/Responsiveness score
- A: Activity score

The AP-side QoE is given by:

$$Q_{\text{AP}} = w_S S + w_T T + w_R R + w_L L + w_A A \tag{4.1}$$

Weights:

$$w_S = 0.28, \quad w_T = 0.32, \quad w_R = 0.15, \quad w_L = 0.15, \quad w_A = 0.10$$

4.1.1 Signal Quality Component

Signal quality uses RSSI to measure RF link strength. RSSI is mapped to the range $[0, 1]$:

$$S = \text{clamp}\left(\frac{\text{RSSI} - (-90)}{60}, 0, 1\right) \tag{4.2}$$

Values:

- RSSI ≤ -90 dBm $\rightarrow S = 0$
- RSSI ≥ -30 dBm $\rightarrow S = 1$

Example:

$$\text{RSSI} = -50 \Rightarrow S = 0.67$$

4.1.2 Throughput Component

Throughput is estimated using the geometric mean of uplink and downlink bitrates:

$$T = \text{clamp}\left(\frac{\sqrt{\text{tx\_bitrate} \cdot \text{rx\_bitrate}}}{\text{phy\_peak}}, 0, 1\right) \tag{4.3}$$

Example:

$$T = \frac{300}{866} = 0.35$$

4.1.3 Reliability Component

Reliability captures link stability using retry and FCS error rates:

$$R = 1 - (\alpha \cdot \text{retry\_rate} + \beta \cdot \text{fcs\_rate}) \tag{4.4}$$

$$\alpha = 0.6, \qquad \beta = 0.4$$

Example:

$$R = 1 - (0.006 + 0.002) = 0.992$$

4.1.4 Latency Component

Latency is estimated via the inactivity timer:

$$L = \text{clamp}\left(1 - \frac{\text{inactive\_msec}}{5000},\ 0,\ 1\right) \tag{4.5}$$

Example:

$$\text{inactive} = 2500 \text{ ms} \Rightarrow L = 0.50$$

4.1.5 Activity Component

Activity captures station usage intensity:

$$A = \text{clamp}\left(\frac{\text{tx\_packets} + \text{rx\_packets}}{\text{max\_frames}}, 0, 1\right) \tag{4.6}$$

Stations above `max_frames` saturate at $A = 1$.

4.2 TCP QoE (Updated)

1. TCP packet capture using `tcpdump`
2. Raw metric extraction from TCP fields
3. Normalization with parameterized constants
4. Final QoE computation

4.2.1 Part 1: TCP Packet Capture

```
# Start capture
sudo tcpdump -i br0 -s 0 -w wifi_tcp.pcap 'tcp' &

# Generate traffic on client

# Stop: Ctrl+C
```

4.2.2 Part 2: TCP Layer Fields

Core TCP Variables

For each packet p_i, extract:

t_i	Timestamp (sec) —`pkt.time`
$\mathbf{sport}_i$	Source Port — `pkt[TCP].sport`
$\mathbf{dport}_i$	Destination Port — `pkt[TCP].dport`
$\mathbf{seq}_i$	Sequence Number — `pkt[TCP].seq`
$\mathbf{ack}_i$	Acknowledgment Number — `pkt[TCP].ack`
$\mathbf{len}_i$	Payload Length (bytes) — `len(bytes(pkt[TCP].payload))`
$\mathbf{TSval}_i$	Timestamp Value (ticks) — `pkt[TCP].options`
$\mathbf{TSecr}_i$	Timestamp Echo (ticks) — `pkt[TCP].options`

Flow Identification

TCP flow identified by 2-tuple:

$$\text{Flow} = (\text{sport}, \text{dport}) \tag{4.7}$$

4.2.3 Part 3: Raw Metric Extraction

Metric 1: Round-Trip Time (RTT)

RTT from TCP Timestamp Option (RFC 1323)

$$\text{RTT}_{i,j}^{\text{TS}} = t_j - \text{TSval}_i \tag{4.8}$$

Where:

- t_j = arrival time of ACK packet (seconds)
- TSval_i = timestamp value from data packet

Metric : RTT Jitter (Variability)

Formula

Jitter is the standard deviation of RTT samples:

$$\text{Jitter} = \sigma_{\text{RTT}} = \sqrt{\frac{1}{M-1}\sum_{k=1}^{M}(\text{RTT}_k - \overline{\text{RTT}})^2} \tag{4.9}$$

Where:

- M = number of RTT samples
- $\overline{\text{RTT}}$ = mean RTT

Metric 3: Packet Loss Rate

Retransmission Detection

A packet is retransmitted if same TCP sequence number appears twice on same flow:

$$\text{seq}_i \in \text{SEEN}[\text{flow}_i] \Rightarrow \text{retransmission} \tag{4.10}$$

Loss Rate Formulas

Per-flow loss rate:

$$\text{Loss}[\text{flow}] = \frac{\text{RETRANS}[\text{flow}]}{\text{TOTAL}[\text{flow}]} \times 100\% \tag{4.11}$$

Overall loss rate:

$$\text{Loss}_{\text{total}} = \frac{\sum_f \text{RETRANS}[f]}{\sum_f \text{TOTAL}[f]} \times 100\% \tag{4.12}$$

Metric 4: Throughput

Formula

Throughput from total TCP payload:

$$B_{\text{total}} = \sum_{p_i:\text{len}_i>0} \text{len}_i \tag{4.13}$$

Capture duration:

$$T_{\text{capture}} = t_{\max} - t_{\min} \tag{4.14}$$

Throughput in Mbps:

$$\text{Throughput}_{\text{Mbps}} = \frac{B_{\text{total}} \times 8}{T_{\text{capture}} \times 10^6} \tag{4.15}$$

4.2.4 Part 4: Normalization with Parameters

RTT Normalization

Formula with Parameter

$$\boxed{Q_{\text{lat}}(\text{RTT}) = \frac{1}{1 + \frac{\text{RTT}}{R_0}}} \tag{4.16}$$

Where R_0 is the RTT reference parameter (default: 50 ms).
Why This Form

- Continuous degradation (no threshold)
- Logarithmic perception (Weber's law)
- Simple computation
- Tunable via R_0

Jitter Normalization

Formula with Parameter

$$\boxed{Q_{\text{jitter}}(\text{Jitter}) = \frac{1}{1 + \frac{\text{Jitter}}{J_0}}} \tag{4.17}$$

Where J_0 is the jitter reference parameter

Loss Rate Normalization

Formula with Parameter

$$\boxed{Q_{\text{loss}}(\text{Loss}) = \frac{1}{1 + L_0 \times \text{Loss}}} \tag{4.18}$$

Where:

- L_0 = loss steepness parameter
- Loss = loss rate in decimal

Throughput Normalization

Formula with Parameter

$$\boxed{Q_{\text{tput}}(\text{Throughput}) = \frac{\text{Throughput}}{T_0 + \text{Throughput}}} \tag{4.19}$$

Where T_0 is the throughput reference parameter
Why Diminishing Returns
Each additional Mbps has less impact. Logarithmic form captures real utility.

4.2.5 Part 5: Final QoE Computation

QoE Formula

$$\boxed{Q_{\text{QoE}} = w_{\text{lat}} \cdot Q_{\text{lat}} + w_{\text{jitter}} \cdot Q_{\text{jitter}} + w_{\text{loss}} \cdot Q_{\text{loss}} + w_{\text{tput}} \cdot Q_{\text{tput}}} \tag{4.20}$$

4.2.6 Summary of Formulas with Parameters

Raw Metrics (TCP Only)

$$\boxed{\text{RTT} = t_{\text{ack}} - \text{TSval}_{\text{data}}} \tag{4.21}$$

$$\boxed{\text{Jitter} = \sqrt{\frac{1}{M-1}\sum_{k=1}^{M}(\text{RTT}_k - \overline{\text{RTT}})^2}} \tag{4.22}$$

$$\boxed{\text{Loss} = \frac{\text{duplicate sequences}}{\text{total packets}} \times 100\%} \tag{4.23}$$

$$\boxed{\text{Throughput} = \frac{\sum \text{payload bytes} \times 8}{T_{\text{capture}} \times 10^6} \text{ Mbps}} \tag{4.24}$$

Normalization with Parameters

$$\boxed{Q_{\text{lat}} = \frac{1}{1 + \text{RTT}/R_0}} \tag{4.25}$$

$$\boxed{Q_{\text{jitter}} = \frac{1}{1 + \text{Jitter}/J_0}} \tag{4.26}$$

$$\boxed{Q_{\text{loss}} = \frac{1}{1 + L_0 \times \text{Loss}}} \tag{4.27}$$

$$\boxed{Q_{\text{tput}} = \frac{\text{Throughput}}{T_0 + \text{Throughput}}} \tag{4.28}$$

Final QoE with Weights

$$\boxed{Q_{\text{QoE}} = w_{\text{lat}} Q_{\text{lat}} + w_{\text{jitter}} Q_{\text{jitter}} + w_{\text{loss}} Q_{\text{loss}} + w_{\text{tput}} Q_{\text{tput}}} \tag{4.29}$$

Default weights: $(0.25, 0.25, 0.35, 0.15)$

4.3 Historical Trend Analysis

While instantaneous QoE values provide a snapshot of performance, wireless conditions often fluctuate due to mobility, interference, and variations in traffic load. To better understand long-term behavior, the system maintains a history of QoE samples for each station. This enables the controller to distinguish between temporary fluctuations and persistent degradation.

Three metrics are tracked: moving average, trend, and volatility.

4.3.1 Moving Average

The moving average smooths short-term noise and provides a stable baseline for QoE evaluation. It is computed over the last k samples:

$$\bar{Q}_t = \frac{1}{k} \sum_{i=0}^{k-1} Q_{t-i} \tag{4.30}$$

A larger k makes the average more stable but slower to react to changes. A typical value is $k = 10$ samples.

4.3.2 Trend Computation

The trend metric represents the direction and rate of QoE change over time. It is computed using the slope of a least-squares linear regression over the last k samples:

$$\text{trend} = \frac{k \sum_{i=1}^{k} i Q_i - (\sum_{i=1}^{k} i)(\sum_{i=1}^{k} Q_i)}{k \sum_{i=1}^{k} i^2 - (\sum_{i=1}^{k} i)^2} \tag{4.31}$$

A positive trend indicates improving conditions (e.g., station moving closer to AP, reduced interference), while a negative trend signals progressive deterioration. This metric is particularly useful for early detection of coverage holes or congestion buildup.

4.3.3 Volatility

Volatility measures the short-term stability of QoE. High volatility indicates rapid fluctuations caused by interference, bursty traffic, hidden-node collisions, or client mobility.

It is computed as the variance of recent QoE values:

$$\sigma^2 = \frac{1}{k} \sum_{i=0}^{k-1} (Q_{t-i} - \bar{Q}_t)^2 \tag{4.32}$$

Low volatility combined with poor QoE suggests persistent problems (e.g., weak signal, overloaded AP), while high volatility suggests transient issues that may resolve without client steering.

4.4 Post-Roam QoE Delta Analysis

To evaluate the effectiveness of a steering decision, the system compares QoE before and after a client roams from AP_1 to AP_2. This allows the controller to validate whether the steering action improved or degraded the user's experience.

The QoE delta is computed as:

$$\Delta Q = Q(\text{AP}_2) - Q(\text{AP}_1) \tag{4.33}$$

A positive ΔQ indicates that the new AP provides better service, justifying the roam. A negative ΔQ suggests that the client was moved to a worse AP, which can trigger corrective actions or adjustments to the steering policy.

Because QoE is computed independently for both AP-side performance (Q_{AP}) and transport-layer performance ($Q_{\text{transport}}$), the controller may evaluate:

$$\Delta Q_{\text{AP}} \quad \text{and} \quad \Delta Q_{\text{transport}}$$

This separation helps identify whether improvements are due to RF conditions (e.g., better SNR, lower retries) or due to transport-layer characteristics (e.g., lower RTT, reduced congestion).

5. Neighbor Ranking Algorithm

5.1 Ranking Overview

Neighbor ranking determines which APs are suitable steering targets for each station. The algorithm processes beacon measurement reports to score and sort candidate APs based on predicted performance.

The ranking must balance competing factors:

- Strong signal (high RSSI) indicates good RF conditions
- High capacity means the AP can handle additional load
- Low current load suggests available airtime

5.2 Scoring Formula

For each neighbor AP visible to a station, the score is computed as:

$$N_i = \gamma_1 \cdot \text{RSSI}_i + \gamma_2 \cdot \text{capacity}_i - \gamma_3 \cdot \text{load}_i \tag{5.1}$$

With weights:

$$\gamma_1 = 0.55, \quad \gamma_2 = 0.35, \quad \gamma_3 = 0.10 \tag{5.2}$$

All components are normalized to comparable ranges before weighting.

5.2.1 RSSI Component

RSSI from beacon reports is converted to a 0-1 score using the same mapping as the QoE signal component:

$$\text{RSSI}_{\text{score}} = \text{clamp}\left(\frac{\text{RCPI}/2 - 110 - (-90)}{60}, 0, 1\right) \tag{5.3}$$

Where RCPI (Received Channel Power Indicator) from the beacon report is converted to dBm by dividing by 2 and subtracting 110.

RSSI receives the highest weight (0.55) because RF quality is the primary constraint on achievable throughput. An AP with poor RSSI will not perform well regardless of its capacity or load.

5.2.2 Capacity Component

Capacity estimates the maximum throughput the AP can provide based on:

- Channel width (20/40/80/160 MHz)
- PHY type (802.11n/ac/ax)
- Spatial streams supported

This information comes from the AP's beacon frame capabilities. The capacity score is normalized relative to the best technology available in the deployment.

Capacity weight is 0.35, reflecting that a high-capacity AP (e.g., 802.11ax with 80 MHz) can support higher throughput than an older AP even at similar RSSI.

5.2.3 Load Component

Channel load comes from the channel utilization field in beacon reports. It represents the fraction of time the channel is busy:

$$\text{load}_{\text{score}} = \frac{\text{channel\_utilization}}{255} \tag{5.4}$$

The score is subtracted (note the negative weight $\gamma_3 = 0.10$) because high load is undesirable. An AP at 90% utilization will provide worse performance than one at 20% utilization.

Load receives the lowest weight because momentary utilization fluctuates significantly. Signal and capacity are more stable predictors.

5.3 Neighbor Aggregation

Beacon reports may arrive from multiple measurement cycles. The system aggregates measurements for each neighbor BSSID:

- Average RSSI across all reports
- Most recent capacity information
- Average channel load

This aggregation reduces the impact of single anomalous measurements. A neighbor that consistently appears with strong RSSI across multiple reports is more reliable than one with a single strong measurement.

5.4 Sorting and Selection

After scoring all neighbors, they are sorted in descending order of score N_i. The top-ranked neighbors become the candidate list for BSS-TM requests.

The system may apply additional filters:

- Minimum RSSI threshold (e.g., reject neighbors below -80 dBm regardless of score)
- Band preference (e.g., prefer 5 GHz over 2.4 GHz when scores are close)
- Same-SSID requirement (only steer within the same network)

The final ranked list is stored in the Neighbor Ranking database, indexed by station MAC address. This list is referenced by the Transition Management Engine when constructing steering requests.

5.5 Future Enhancement: Historical Performance

Currently, ranking is purely based on predicted performance from measurements. The planned enhancement would incorporate actual historical performance:

$$N_i = \gamma_1 \cdot \text{RSSI}_i + \gamma_2 \cdot \text{capacity}_i - \gamma_3 \cdot \text{load}_i + \gamma_4 \cdot \text{history}_i \tag{5.5}$$

Where $\texttt{history}_i$ represents the average post-roam ΔQ when this station (or similar stations) previously roamed to AP_i. This would enable adaptive learning from steering outcomes.

6. Transition Management Logic

6.1 Steering Decision Process

The Transition Management Engine implements the steering policy by examining each station and deciding whether to issue a BSS-TM request. The decision logic balances the need to improve poor connections against the disruption of unnecessary roaming.

6.1.1 Decision Algorithm

For each connected station, the engine executes:

```
station ← next connected station
qoe ← fetch current QoE(station)
if qoe is None then
    skip {No QoE data yet}
end if
if qoe.score > 0.55 then
    skip {QoE acceptable}
end if
neighbors ← fetch ranked neighbors(station)
if neighbors is empty then
    skip {No steering candidates}
end if
request ← build_bss_tm_request(station, neighbors)
controller.send(request)
log_steering_event(station, qoe, neighbors)
```

6.2 QoE Threshold

The threshold value of 0.55 represents the point below which user experience is considered degraded. This value was chosen to trigger steering before conditions become severe enough to cause application failures or disconnections.

The threshold is intentionally conservative (neither too high nor too low):

- Too high (e.g., 0.8): Causes excessive steering of marginally suboptimal connections
- Too low (e.g., 0.3): Waits until connections are severely degraded before steering

At QoE = 0.55, a typical station might have:

- RSSI around -65 to -70 dBm (marginal signal)
- Throughput at 40-50% of peak
- Some retries but not excessive
- Generally usable but noticeably suboptimal performance

6.3 BSS Transition Request Construction

When the engine decides to steer a station, it constructs an IEEE 802.11v BSS Transition Management request using:

```
BssTmRequestBuilder(
    sta_addr=mac,
    req_mode=PREFERRED_CAND_LIST_INCLUDED,
    neighbors=nbs
)
```

The request contains several key fields:

6.3.1 Request Mode

The `PREFERRED_CAND_LIST_INCLUDED` mode tells the client that the frame includes a list of recommended APs. The client firmware considers this list when selecting its next association target. Other possible modes include:

- `ABRIDGED`: No candidate list, client chooses independently
- `DISASSOC_IMMINENT`: Warn client of impending disconnection
- `BSS_TERMINATION`: AP is shutting down

The preferred candidate list mode is used because it provides guidance without forcing a specific AP, allowing the client to factor in its own measurements and preferences.

6.3.2 Neighbor List

The neighbor list contains entries for each candidate AP, ordered by ranking score. Each entry includes:

- **BSSID**: MAC address of the target AP
- **BSSID Info**: Capabilities flags (HT, VHT, etc.)
- **Operating Class**: Regulatory domain frequency band
- **Channel Number**: Primary channel
- **PHY Type**: 802.11n/ac/ax indicator

This information comes from the beacon reports and neighbor database. The client uses it to quickly find and associate with the recommended APs without performing a full scan.

6.3.3 Additional Parameters

The request may include optional parameters:

- **Disassociation Timer**: If set, indicates when the AP will disconnect if the client doesn't roam (not typically used in QoE-based steering)

- **Validity Interval**: How long the neighbor list remains valid
- **BSS Termination Duration**: Used when AP is shutting down (not applicable here)

For QoE-based steering, only the neighbor list is required. The station is free to remain associated if it prefers, but is encouraged to consider the alternatives.

6.4 Rate Limiting and Hysteresis

To prevent steering oscillation, the engine implements several guards:

6.4.1 Minimum Time Between Steers

Each station has a timestamp of its last steering event. The engine will not issue a new BSS-TM request until sufficient time has elapsed (typically 120-300 seconds). This prevents rapid back-and-forth steering between APs.

6.4.2 QoE Hysteresis

The current system doesn't account for QoE Hysteresis, more experimentations are required to give any potential solution.

6.4.3 Minimum RSSI Delta

The top-ranked neighbor must have significantly better RSSI than the current AP (typically 5-10 dB improvement) to justify steering. Steering to a marginally better AP may not be worth the temporary disruption.

6.5 Steering Event Logging

Every steering decision is logged with:

- Station MAC address
- Current AP BSSID
- Current QoE score and component breakdown
- Ranked neighbor list sent in the BSS-TM request
- Timestamp

These logs enable post-mortem analysis of steering behavior and provide the data needed for future post-roam QoE delta analysis.

6.6 Client Response Handling

After sending a BSS-TM request, the system monitors for the client's response. The client may:

1. **Accept**: Send a BSS-TM response frame indicating it will roam, then reassociate to a neighbor
2. **Reject**: Send a BSS-TM response frame declining to roam
3. **Ignore**: Never respond (non-compliant or legacy client)

The response status is logged but does not immediately affect future steering decisions. In the planned post-roam analysis feature, rejected or ignored steering requests would factor into AP ranking adjustments.

7. StateAPI: HTTP Interface

7.1 API Purpose and Design

The StateAPI subsystem exposes a read-only view of system state via HTTP endpoints. It serves several purposes:

- External monitoring systems can poll for metrics
- Dashboards can fetch current state on demand
- Debugging tools can inspect station conditions
- Historical analysis tools can collect time-series data

The API is intentionally read-only. All configuration and control operations occur through other interfaces to maintain separation of concerns.

7.2 Data Refresh Process

Before serving any API response, the StateAPI calls:

```
self.qoe.update()
```

This ensures the returned data reflects current conditions rather than stale cached values. The update recomputes QoE scores based on the most recent measurements, so API responses always represent the latest system understanding.

This synchronous update adds minimal latency (typically under 10ms for 100 stations) because the QoE computation is fast and operates on pre-fetched data from the databases.

7.3 Response Structure

Each API response follows a standard envelope format:

```
{
  "timestamp": <unix epoch seconds>,
  "status": "ok" | "error",
  "component": "StateAPI",
  "version": "1.0",
  "length": <number of station records>,
  "data": [ <station records> ]
}
```

The `timestamp` field indicates when the response was generated. The `length` field allows clients to quickly determine how many station records are present without parsing the entire array.

7.4 Station Record Format

Each station in the `data` array contains:

7.4.1 Identity

```
"public_id": "A4:5E:60-9f13ab"
```

The anonymized identifier (see Section 7.5 for details).

7.4.2 Connection Status

```
"connected": true | false
```

Whether the station is currently associated to an AP under management.

7.4.3 Signal Metrics

```
"signal": {
  "avg_signal": -31,          // dBm
  "score": 1.0                // normalized 0-1
}
```

Current RSSI and the computed signal component score.

7.4.4 Throughput Metrics

```
"throughput": {
  "tx_bitrate": 54,           // Mbps
  "rx_bitrate": 54,           // Mbps
  "score": 0.22               // normalized 0-1
}
```

Current TX/RX bitrates and throughput component score.

7.4.5 Reliability Metrics

```
"reliability": {
  "tx_retry_rate": 5.9e-7,    // ratio
  "tx_failed_rate": 0.0,      // ratio
  "rx_fcs_error_rate": 0.0,   // ratio
  "score": 1.0                // normalized 0-1
}
```

Retry and error rates with the computed reliability score.

7.4.6 Latency Metrics

```
"latency": {
  "inactive_msec": 2064,      // milliseconds
  "score": 1.0                // normalized 0-1
}
```

Time since last frame exchange and latency component score.

7.4.7 Activity Metrics

```
"activity": {
  "total_tx_rx_packets": 79617,
  "score": 1.0                // normalized 0-1
}
```

Total frame count and activity component score.

7.4.8 QoE Summary

```
"qoe": {
  "overall": 0.883,           // composite score
  "trend": "insufficient_data" | "improving" | "stable" | "degrading",
  "volatility": 0.02          // variance (if available)
}
```

The final composite QoE score, trend classification, and volatility measure.

7.4.9 Timestamp

```
"timestamp": "2025-11-15T16:47:49.937132"
```

ISO 8601 formatted timestamp for this station's measurement snapshot.

7.5 Identifier Anonymization

Raw MAC addresses contain device-identifying information and may be considered personally identifiable. To enable public API exposure without privacy concerns, the StateAPI anonymizes all station identifiers.

7.5.1 Anonymization Algorithm

The anonymization preserves the OUI (Organizationally Unique Identifier) portion while hashing the device-specific portion:

$$\text{public\_id} = \text{OUI}(\text{MAC}) \;||\; \text{“-”} \;||\; H(\text{suffix}) \tag{7.1}$$

Where:

- OUI(MAC) extracts the first 3 bytes (6 hex digits)
- suffix is the last 3 bytes of the MAC address
- H is SHA-256 hash, truncated to 24 bits (6 hex digits)

For example, given MAC address `A4:5E:60:3C:F1:82`:

1. OUI = `A4:5E:60` (vendor identifier, retained)
2. Suffix = `3C:F1:82` (device-specific)
3. Hash = `SHA256("3C:F1:82")` = `9f13ab...` (truncated)
4. Result = `A4:5E:60-9f13ab`

7.5.2 Privacy Properties

This scheme provides several privacy properties:

- **Non-reversible**: The hash function is one-way; the original MAC cannot be recovered from the public ID
- **Vendor-preserving**: The OUI remains visible, allowing analysis by device manufacturer
- **Consistent**: The same MAC always produces the same public ID (useful for tracking across API calls)
- **Unlinkable**: Without knowledge of the MAC, an observer cannot link public IDs across different deployments

7.5.3 Implementation

The anonymization is performed in the `build_station_api_dict()` method:

```
def anonymize_mac(mac: str) -> str:
    oui = mac[:8]  # "A4:5E:60"
    suffix = mac[9:]  # "3C:F1:82"
    hash_input = suffix.encode('utf-8')
    hash_output = hashlib.sha256(hash_input).hexdigest()
    return f"{oui}-{hash_output[:6]}"
```

This function is called once per station when building the API response. The internal databases, schedulers, and decision logic continue to use the raw MAC address for all operations. Only the final API serialization step performs anonymization.

7.5.4 OUI Retention Rationale

Retaining the OUI (vendor identifier) enables useful analysis:

- Identifying problematic device types (e.g., "all Apple devices have low QoE")
- Vendor-specific tuning (e.g., "Samsung phones respond well to 5 GHz steering")
- Hardware capability inference (OUI correlates with PHY capabilities)

The OUI alone does not uniquely identify a device, as millions of devices share the same vendor identifier. The privacy risk is minimal compared to the analytical value.

7.6 Example API Output

A complete API response for two stations:

```
{
  "timestamp": 1763205469.937012,
  "status": "ok",
  "component": "StateAPI",
  "version": "1.0",
```

```
"length": 2,
"data": [
  {
    "public_id": "0E:ED:C0-3A91E2",
    "connected": true,
    "signal": {
      "avg_signal": -31,
      "score": 1.0
    },
    "throughput": {
      "tx_bitrate": 54,
      "rx_bitrate": 54,
      "score": 0.22
    },
    "reliability": {
      "tx_retry_rate": 5.9e-7,
      "tx_failed_rate": 0.0,
      "rx_fcs_error_rate": 0.0,
      "score": 1.0
    },
    "latency": {
      "inactive_msec": 2064,
      "score": 1.0
    },
    "activity": {
      "total_tx_rx_packets": 79617,
      "score": 1.0
    },
    "qoe": {
      "overall": 0.883,
      "trend": "insufficient_data",
      "volatility": null
    },
    "timestamp": "2025-11-15T16:47:49.937132"
  },
  {
    "public_id": "B8:27:EB-7F2A19",
    "connected": true,
    "signal": {
      "avg_signal": -67,
      "score": 0.38
    },
    "throughput": {
      "tx_bitrate": 24,
      "rx_bitrate": 18,
      "score": 0.11
    },
    "reliability": {
      "tx_retry_rate": 0.08,
```

```
        "tx_failed_rate": 0.003,
        "rx_fcs_error_rate": 0.02,
        "score": 0.94
      },
      "latency": {
        "inactive_msec": 450,
        "score": 0.91
      },
      "activity": {
        "total_tx_rx_packets": 12450,
        "score": 0.62
      },
      "qoe": {
        "overall": 0.48,
        "trend": "degrading",
        "volatility": 0.015
      },
      "timestamp": "2025-11-15T16:47:49.937298"
    }
  ]
}
```


The first station shows excellent connectivity (QoE = 0.883) with strong signal, perfect reliability, and high activity. The second station exhibits marginal performance (QoE = 0.48) with weak signal, low throughput, and a degrading trend...this would be a steering candidate.

8. Dashboards and Socket Multiplexing

8.1 Dashboard Architecture

The dashboard subsystem provides real-time monitoring of system activity through WebSocket connections. Multiple clients can simultaneously subscribe to different event streams without interfering with each other.

8.2 Socket Naming and Multiplexing

The system maintains separate named sockets for different event types:

- **bmrep**: Beacon Measurement Reports
- **lmrep**: Link Measurement Reports
- **nrank**: Neighbor Ranking updates
- **stqoe**: Station QoE computation results
- **statn**: Station connection/disconnection events
- **bsstm**: BSS Transition Management requests and responses

Each socket operates independently. A dashboard client can subscribe to one, several, or all sockets depending on what it needs to display.

8.3 SocketPool Management

The `SocketPool` class manages all active WebSocket connections. When a dashboard connects, it specifies which sockets to subscribe to. The pool maintains a mapping:

```
socket_name -> [client1, client2, ..., clientN]
```

When an event occurs (e.g., a new beacon report arrives), the system calls:

```
socketpool.emit(socket_name="bmrep", msg)
```

The pool then broadcasts this data to all clients subscribed to that socket.

8.4 Event Generation

Each dashboard regenerates its view on every push cycle. The schedulers and event handlers trigger dashboard updates:

- When Link Measurement Scheduler receives a report ... emit to `lmrep`

- When Beacon Measurement Scheduler receives a report ... emit to `bmrep`
- When QoE Scheduler completes computation ... emit to `stqoe`
- When Neighbor Ranking updates ... emit to `nrank`
- When BSS-TM request sent ... emit to `bsstm`

9. Storage and Snapshotting

9.1 Database Architecture

All system state resides in in-memory databases for fast access. These databases use simple key-value or table-based structures:

- **LinkMeasurementDB**: Maps station MAC ... latest link measurement report
- **BeaconMeasurementDB**: Maps (station MAC, neighbor BSSID) ... aggregated beacon measurements
- **QoEDB**: Maps station MAC ... current QoE score, components, and historical model
- **NeighborRankingDB**: Maps station MAC ... sorted list of candidate APs
- **StationDB**: Maps station MAC ... connection state and statistics

In-memory storage enables microsecond-latency queries for schedulers and the API. There is no disk I/O during normal operation.

9.2 Snapshot Mechanism

The `Routine` subsystem implements periodic database snapshotting to provide crash recovery and historical archival. The snapshot process serializes all in-memory databases to disk at configured intervals (typically every 5-15 minutes).

9.2.1 Snapshot Format

Each snapshot creates a directory with the following structure:

```
snapshots/
  2025-11-15/
    16-45-00/
      link_measurements.json
      beacon_measurements.json
      qoe_db.json
      neighbor_ranking.json
      station_db.json
      metadata.json
```

The directory hierarchy can be configured as:

- **Flat**: All snapshots in a single directory with timestamp prefixes
- **Date**: Organized by date, with hourly subdirectories
- **Hour**: Full date-hour hierarchy as shown above

9.2.2 Serialization Process

Each database implements a `serialize()` method that converts its internal state to JSON:

```
def serialize(self) -> dict:
    return {
        "version": 1,
        "timestamp": time.time(),
        "entries": self._serialize_entries()
    }
```

The serialized dictionary is written to a JSON file. JSON was chosen for human readability during debugging, though binary formats (MessagePack, Protobuf) could improve performance for large deployments.

9.2.3 Recovery Process

On startup, the system checks for existing snapshots. If found, it loads the most recent snapshot:

1. Find the newest snapshot directory by timestamp
2. For each database file, call `deserialize()`
3. Rebuild in-memory indexes and data structures
4. Resume normal operation with recovered state

The recovery process is atomic: if any database file fails to load, the entire snapshot is rejected and the system starts with empty state.

9.2.4 Retention Policy

The snapshot retention policy prevents unbounded disk usage:

- **Keep Recent**: Retain all snapshots from the last N days (configurable, typically 7)
- **Hourly Sampling**: For older snapshots, keep only one per hour
- **Daily Sampling**: For very old snapshots, keep only one per day
- **Maximum Age**: Delete all snapshots older than M days (configurable, typically 30)

The retention process runs after each snapshot operation to clean up old data:

snapshots ← list all snapshot directories, sorted by timestamp
for each snapshot in snapshots **do**
  age ← now - snapshot.timestamp
  if age < RECENT_DAYS **then**
    keep
  else if age < HOURLY_DAYS AND is_hourly_sample(snapshot) **then**
    keep
  else if age < DAILY_DAYS AND is_daily_sample(snapshot) **then**
    keep

```
    else
      delete
    end if
  end for
```

9.2.5 Snapshot Consistency

Snapshots are not transaction-consistent across all databases. The scheduler threads continue to run during snapshot operations, so different databases may reflect slightly different points in time.

This is acceptable because:

- The system has no critical consistency requirements across databases
- Recovered state quickly reconverges as schedulers resume
- The alternative (stopping all schedulers during snapshot) would disrupt real-time operation

For applications requiring strict consistency, the snapshot process could be modified to use a two-phase commit protocol, but this adds complexity that is not currently warranted.

10. 802.11mc RTT (Updated)

Location-Aware Interference Hotspots using IEEE 802.11mc (Wi-Fi RTT) with FTM-Only Measurements

10.1 Introduction

Indoor Wi-Fi networks often suffer from interference, multipath, and coverage holes. Traditional methods rely on RSSI and throughput, which are noisy and not directly related to distance.

IEEE 802.11mc introduces **Wi-Fi Round Trip Time (RTT)**, also called **Fine Timing Measurement (FTM)**, which allows a device to estimate its distance to an access point (AP) by measuring the travel time of Wi-Fi packets.

We use **FTM-only data** (distance and its quality) to build a single combined floor heatmap of a building, detect interference or problematic tiles, and preserve privacy by avoiding MAC addresses and exact coordinates.

10.2 Basic Principle of 802.11mc Wi-Fi RTT

10.2.1 FTM Exchange Protocol

Two roles participate in FTM: the initiator, usually the client device such as a phone or laptop, and the responder, which is the AP supporting 802.11mc FTM.

They exchange timestamped frames. The protocol uses four timestamps: T_1 when the initiator sends an FTM request, T_2 when the responder receives it, T_3 when the responder sends the FTM response, and T_4 when the initiator receives that response.

10.2.2 RTT and Distance Formulas

Round Trip Time (RTT):

$$\text{RTT} = (T_4 - T_1) - (T_3 - T_2)$$

Distance:

$$d = \frac{c \times \text{RTT}}{2}$$

Here, $c = 3 \times 10^8$ m/s is the speed of light, and the division by 2 accounts for the signal traveling to the AP and back.

Worked Example:

If RTT = 200 nanoseconds (200×10^{-9} seconds):

$$d = \frac{3 \times 10^8 \times 200 \times 10^{-9}}{2} = \frac{60}{2} = 30 \text{ meters}$$

In practice, the Wi-Fi chipset provides an estimated distance and a corresponding standard deviation that indicates the stability and reliability of that estimate.

10.3 Collected Data (FTM-Only, No MAC/MCS)

Each FTM measurement contains a timestamp, a logical AP identifier such as "AP_1", a floor identifier, the measured FTM distance, its standard deviation, a quality label such as HIGH, MEDIUM, or LOW, and a session ID that rotates frequently (for example, once per hour).

Not stored:

- Client MAC addresses.
- AP MAC/BSSID.
- MCS, throughput, retries, etc.
- Exact device coordinates.

This keeps the system focused on FTM distance and privacy-preserving.

10.4 Floor Plan, AP/Client Placement, and Tiles

10.4.1 Coordinate System and Tiles

The coordinate system uses the X-axis for horizontal distance in meters and the Y-axis for vertical distance.

Example: floor size 20 m × 10 m. This area is divided into **coarse tiles**, for example **5 m × 5 m**. For a 20 m × 10 m floor the tile indices for a 20 m × 10 m example are X indices 0–3 and Y indices 0–1.

Each tile has a unique ID such as `floor_1_x0y0`, `floor_1_x1y0`, or `floor_1_x2y1`.

Tile size is chosen **larger than typical RTT error** (e.g., 3–5 times), so that positions remain coarse and privacy is preserved.

10.4.2 Floor Plan with AP and Client Placement

Figure 10.1 shows the floor layout with AP placement and sample client positions.

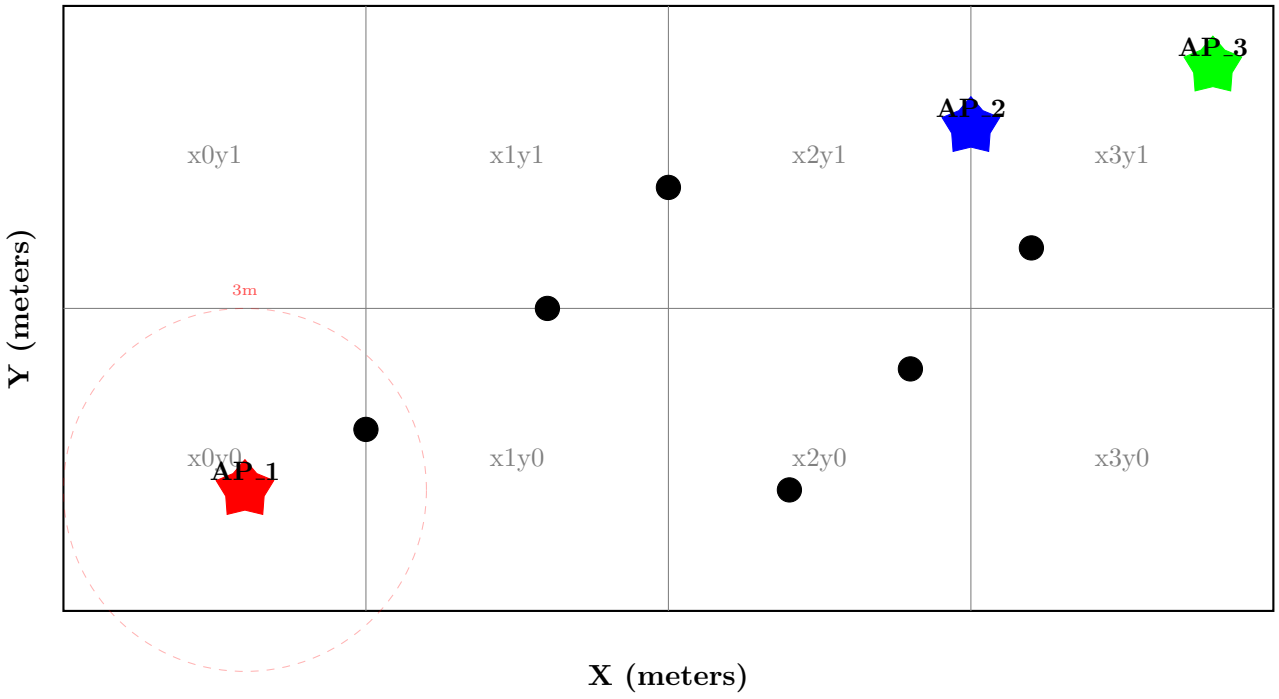

Figure 10.1: Floor plan with 5 m × 5 m tiles, AP placement, and example client locations. Red, blue, and green stars represent AP_1, AP_2, and AP_3 respectively. Black dots are example client positions.

10.5 Assigning FTM Measurements to Tiles

Each measurement contains the distance from AP to client, d, along with its standard deviation σ.

Each tile has a center coordinate; for example, tile x1y0 has its center at $(7.5\,\text{m}, 2.5\,\text{m})$.

Conceptual assignment:

1. For a given AP and measurement:
   - Compute the geometric distance from the AP to each tile center.
2. Select tiles where the difference between this geometric distance and FTM distance is within an **error margin**:

$$|\text{distance to tile center} - d| \leq (\sigma + \text{safety margin})$$

3. From those candidates, choose the tile whose distance to the AP is closest to d.

This gives each measurement a **coarse tile ID**, such as `floor_1_x1y0`. Exact client coordinates are never stored.

10.6 Time Windows and Aggregation

The system works in **time windows**, for example every 5 minutes.

For each tile in each time window, the system aggregates all FTM measurements assigned to that tile. For each tile and time window, the system computes aggregated statistics such as the number of FTM measurements, mean and median distance, distance standard deviation, 95th-percentile distance, the mean of reported standard deviations, and the fraction of HIGH-quality readings. These statistics become the **input** to hotspot/interference formulas.

10.7 Hotspot and Interference Metrics (Formulas)

The project uses only FTM behavior to decide whether a tile is "problematic". Three main metrics are defined per tile and per time window.

10.7.1 Metric 1 – FTM Precision Hotspot

This metric looks at how **spread out** the measured distances are in a tile.

Let:

σ_{tile}: standard deviation of FTM distances in that tile.

σ_{baseline}: standard deviation in a known "good" area (LOS), found during calibration (for example, 0.6 m).

Define:

$$\text{FTM\_Hotspot\_Score} = \frac{\sigma_{\text{tile}}}{\sigma_{\text{baseline}}}$$

Interpretation: Values close to 1 imply normal behavior, while higher values indicate more unstable distance estimates and a higher likelihood of multipath or NLOS.

Example thresholds:

$$\text{FTM\_Hotspot\_Score} = \begin{cases} \leq 1.5 & : \text{normal} \\ 1.5 - 2.0 & : \text{medium issue} \\ 2.0 - 3.0 & : \text{high issue} \\ > 3.0 & : \textbf{CRITICAL HOTSPOT} \end{cases}$$

10.7.2 Metric 2 – Coverage Quality Hotspot

This metric checks if the tile mostly has **good or bad** FTM measurements.

Let N_{total} be the total FTM measurements in the tile and N_{high} be the number of measurements with label "HIGH" quality.

Define:

$$\text{Coverage\_Score} = \frac{N_{\text{high}}}{N_{\text{total}}}$$

Interpretation: A Coverage_Score near 1 indicates mostly high-quality readings and good coverage; values near 0 indicate weak or unstable regions.

Example thresholds:

$$\text{Coverage\_Score} = \begin{cases} \geq 0.80 & : \text{excellent} \\ 0.60 - 0.80 & : \text{acceptable} \\ 0.40 - 0.60 & : \text{degraded} \\ < 0.40 & : \textbf{COVERAGE HOTSPOT} \end{cases}$$

10.7.3 Metric 3 – Multipath Distortion Indicator

This metric measures how "skewed" the distances are towards larger values (due to signal reflections).

Let median_d be the median FTM distance in the tile and $p95_d$ the 95th percentile FTM distance in the tile.

Define:

$$\text{Multipath\_Distortion} = \frac{p95_d - \text{median}_d}{\text{median}_d}$$

Interpretation: Low values imply distances tightly grouped around the median (good). High values indicate many measurements are much larger than the median, which is likely caused by multipath.

Example thresholds:

$$\text{Multipath\_Distortion} = \begin{cases} < 0.15 & : \text{low distortion} \\ 0.15 - 0.25 & : \text{moderate} \\ > 0.25 & : \textbf{MULTIPATH HOTSPOT} \end{cases}$$

10.7.4 Combined Hotspot Severity per Tile

To summarize the three metrics into a single number per tile, we define a **Hotspot Severity**.

First define S_1 as the normalized FTM precision score, $S_2 = 1 - \text{Coverage\_Score}$ to reflect worse coverage with a larger value, and S_3 as the Multipath_Distortion (which can be capped to 1.0 if desired).

Choose weights that sum to 1; for example, we use $w_1 = 0.5$, $w_2 = 0.3$, and $w_3 = 0.2$, reflecting that FTM precision is most important, coverage next, and multipath shape last.

Then:

$$\text{Hotspot\_Severity} = w_1 S_1 + w_2 S_2 + w_3 S_3$$

Example threshold:

$$\text{Hotspot\_Severity} = \begin{cases} < 0.40 & : \text{normal} \\ 0.40 - 0.60 & : \text{medium severity} \\ > 0.60 & : \textbf{SEVERE HOTSPOT} \end{cases}$$

Worked Example: In tile x2y2, the FTM precision score is 3.5 (so $S_1 = 1.0$), the coverage score is 0.58 (so $S_2 = 0.42$), and the multipath distortion is 0.30 (so $S_3 = 0.30$).

$$\text{Severity} = 0.5(1.0) + 0.3(0.42) + 0.2(0.30) = 0.50 + 0.126 + 0.06 = 0.686$$

Interpretation: 0.686 (68.6%) $\Rightarrow$ **SEVERE HOTSPOT** (recommend mitigation).

10.8 Heatmap and Visualizations

10.8.1 Heatmap Concept

Tiles appear green for low severity, yellow for medium severity, and red for severe hotspots where interference or multipath is likely.

10.8.2 Example Heatmap

Figure 10.6 shows a sample hotspot severity heatmap based on hypothetical measurements.

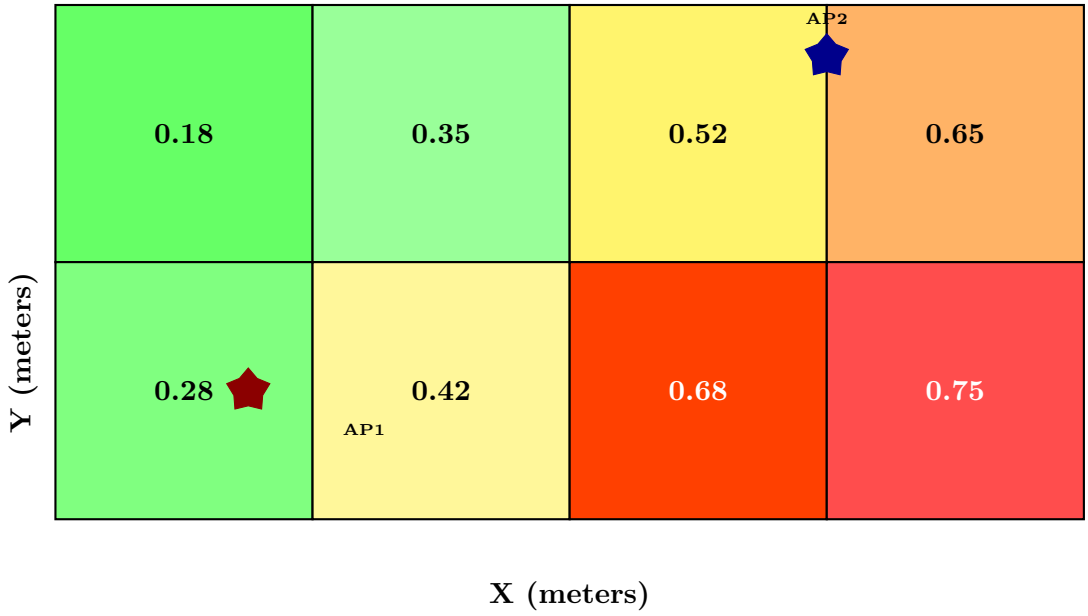

Figure 10.2: Hotspot severity heatmap per tile. Each tile shows its Hotspot Severity score (0–1). Green indicates good areas, yellow medium, and red indicates severe hotspots where mitigation is recommended.

10.8.3 Per-Tile Metric Comparison

Figure 10.3 shows a bar chart comparing FTM precision scores across tiles, highlighting which exceed the hotspot threshold (1.5).

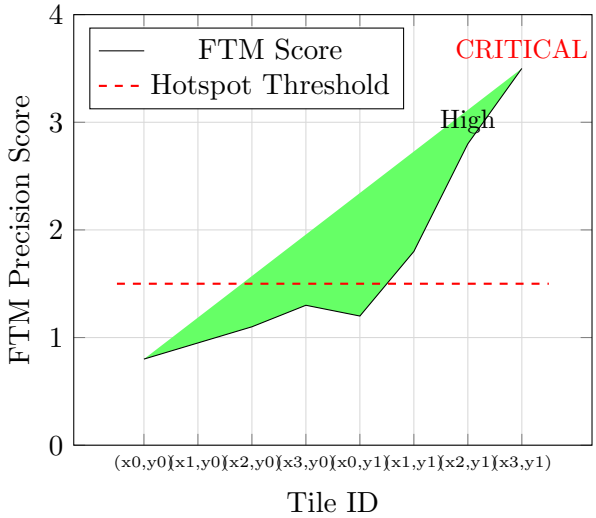

Figure 10.3: FTM precision score (FTM_Hotspot_Score) per tile. Bars exceeding the red threshold line (1.5) are classified as hotspots. Tiles x2,y1 and x3,y1 show the highest scores, indicating severe multipath or NLOS.

10.9 Overall Implementation Flow (High Level)

Flow chat below shows the complete data flow from measurement to visualization.

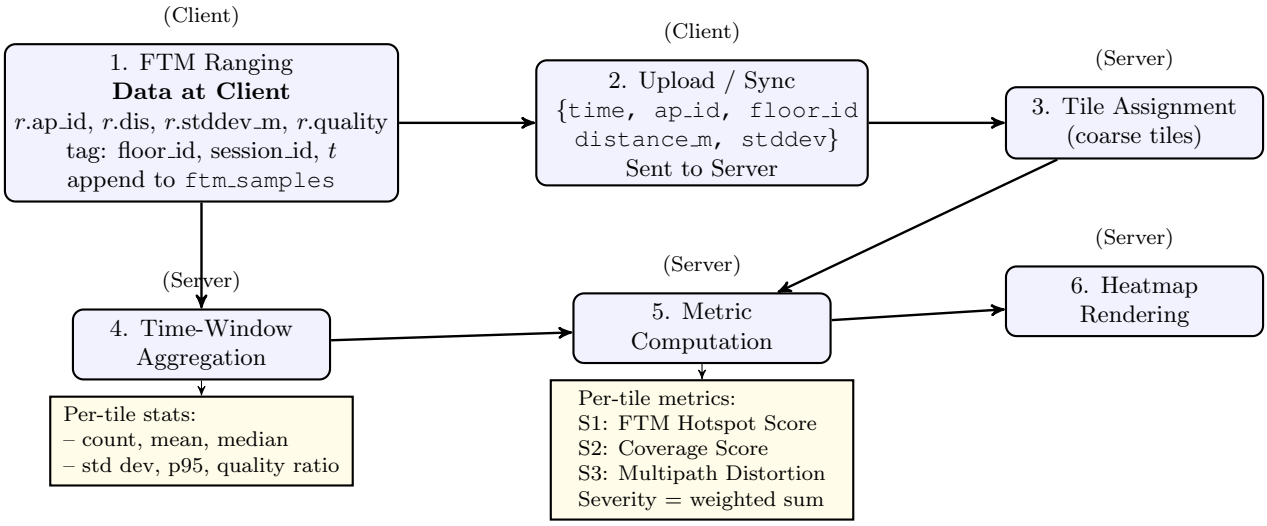

Figure 10.4: FTM-based hotspot analysis pipeline with clear client/server labeling.

10.9.1 Detailed Steps

1. **Measurement phase**: Clients periodically perform 802.11mc FTM ranging with nearby APs. Each result: timestamp, AP logical ID, FTM distance, FTM standard deviation, quality label.

2. **Tile assignment phase**: Using known AP positions and measured distance, each FTM measurement is mapped to one floor tile (coarse location).

3. **Aggregation phase** (e.g., every 5 minutes): For each tile and time window, collect all measurements and compute statistics: standard deviation, coverage score, multipath distortion, etc.

4. **Hotspot Detection**: Compute **Hotspot Severity** using the three formulas from Section 7.

5. **Visualization phase**: Draw the floor plan. Color each tile according to its Hotspot Severity. Optionally mark AP positions and example clients.

6. **Analysis phase**: Identify tiles with high severity (red). Investigate causes (walls, long corridors, reflective surfaces). Adjust AP placement, channels, or power, and repeat measurements to confirm improvement.

10.10 Privacy Considerations

The system stores no client MAC addresses, and APs are identified only by logical IDs rather than BSSIDs. It stores only coarse tile IDs instead of exact coordinates. Session IDs rotate frequently to prevent device tracking, and reports are aggregated per tile and time window rather than per user. Only distance measurements are saved, with no throughput or retry information, ensuring strong privacy guarantees.

10.11 Conclusion

This project demonstrates how **IEEE 802.11mc Wi-Fi RTT (FTM)** can be used to build a **location-aware interference map** of a Wi-Fi deployment using only distance estimates, measurement uncertainty, and coarse spatial tiles. By defining clear metrics and formulas for FTM precision hotspot, coverage quality, and multipath distortion, and combining them into a **Hotspot Severity** score per tile, network engineers can identify problematic regions, make data-driven decisions about AP placement and configuration, preserve user privacy by avoiding MAC addresses or fine-grained location tracking, and incrementally optimize network capacity and coverage. The implementation pipeline is straightforward: collect FTM data, assign to tiles, aggregate, compute hotspot scores, and visualize. The result is a practical tool for Wi-Fi network optimization with strong privacy guarantees.

10.12 Key Formulas Summary

Formula	Purpose	Input	Threshold
$d = \frac{c \times \text{RTT}}{2}$	FTM distance	RTT (ns)	N/A
FTM Score $= \frac{\sigma_{\text{tile}}}{\sigma_{\text{base}}}$	Precision hotspot	Tile stddev	> 1.5
Coverage $= \frac{N_{\text{high}}}{N_{\text{total}}}$	Link quality	Burst counts	< 0.60
Multipath $= \frac{p95-\text{med}}{\text{med}}$	Distance variance	Percentiles	> 0.25
Severity $= 0.5S_1 + 0.3S_2 + 0.2S_3$	Combined score	3 components	> 0.60

Table 10.1: Summary of key formulas used in hotspot detection and severity calculation.

10.13 Example FTM Data Record


```
{
  "timestamp": "2025-12-04T14:30:12Z",
  "session_id": "phone_session_42",
  "responder_ap": "ap_ftm_responder_1",
  "network_ssid": "OfficeNet",
  "tile_id": "floor_1_x0y1",
```

```
  "ftm_distance_m": 8.2,
  "ftm_distance_stddev_m": 0.8,
  "ftm_rtt_ns": 54.7,
  "measurement_quality": "HIGH",
  "ftm_burst_index": 1
}
```

10.14 Client System Requirements

- Device support: Must have 802.11mc FTM capability
- Permissions: `ACCESS_FINE_LOCATION` `CHANGE_WIFI_STATE`, `ACCESS_WIFI_STATE`.
- Target APs: Must support 802.11mc FTM responder mode (most Wi-Fi 5/6 APs).

10.15 Results

The simulation and heatmap visualizations provide a compact summary of measured FTM behavior and the derived hotspot severity across the floor plan. The simulation view (Figure 10.5) shows the run-time layout with APs, client positions, and FTM distance lines; and the hotspots. The aggregated hotspot heatmap (Figure 10.6) overlays per-tile severity values computed from FTM precision, coverage, and multipath distortion.

Visually, severe hotspots (high severity, **red** tiles) concentrate around areas with clustered AP placement or near reflective structures; moderate hotspots (**yellow**) appear in intermediate coverage zones. As an example from the computed metrics, the tile (x2 y2) produced a Hotspot Severity of 0.686, classifying it as a **SEVERE HOTSPOT** and suggesting targeted mitigation (adjust AP power/placement or antenna orientation). In contrast, **green** tiles indicate stable FTM estimates and reliable coverage.

These visual and numeric results validate that the FTM-only approach can detect coarse-grained interference and multipath zones suitable for operational follow-up .

Figure 10.5: FTM-based simulation view showing tile-level hotspot severity, with red indicating high interference, yellow moderate, and green stable regions.

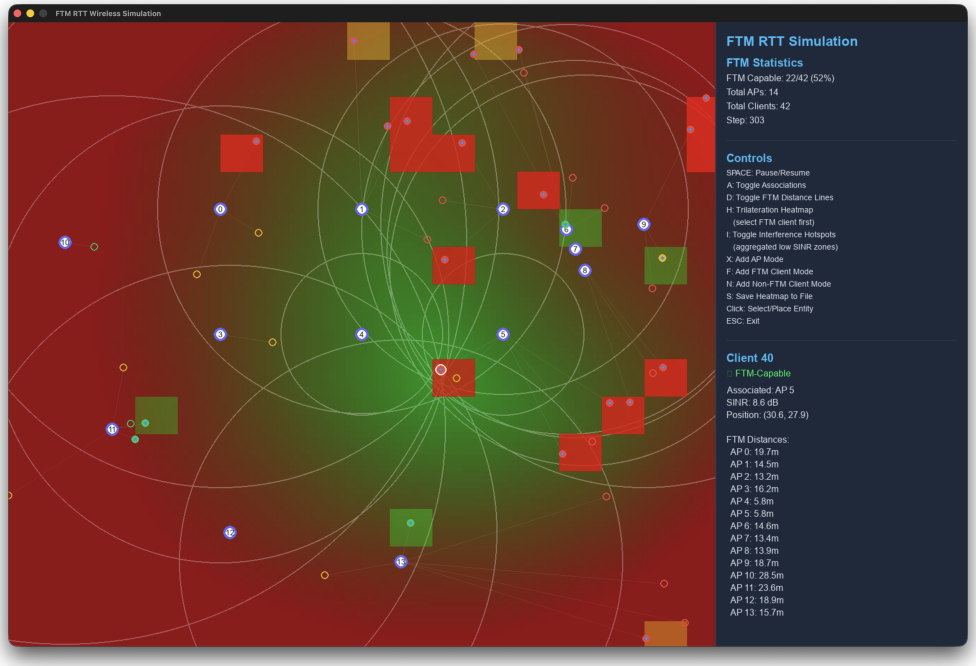

Figure 10.6: FTM-based heatmap illustrating distance-inferred coverage quality, with red indicating strong estimates and green highlighting degraded or interference-prone regions.

11. Conclusion

The Wi-Fi Steering Engine implements a complete architecture for intelligent, automated client steering across enterprise wireless deployments. The system integrates measurement acquisition through IEEE 802.11k protocols, real-time quality-of-experience analytics, multi-dimensional neighbor ranking, and automated steering via IEEE 802.11v BSS Transition Management.

11.1 Architectural Summary

The modular design provides several key benefits:

- **Independence**: Each subsystem operates on its own schedule without tight coupling
- **Extensibility**: New analytics modules or decision logic can be added without modifying the core framework
- **Testability**: Each component can be tested in isolation with mocked data sources
- **Observability**: Multiple interfaces (API, dashboards, logs) provide comprehensive system visibility

The scheduler-driven architecture ensures predictable timing and simplifies reasoning about system behavior. The in-memory database design provides fast access while periodic snapshotting enables crash recovery.

11.2 Privacy and Security

The StateAPI's anonymization layer ensures that client identifiers can be safely exposed to external monitoring systems without privacy concerns. The OUI-preserving hash scheme balances privacy protection with analytical utility.

Internal system logic continues to use raw MAC addresses for all operations, ensuring that anonymization does not impact decision-making or data correlation. The privacy boundary is clearly defined at the API serialization layer.

11.3 Production Readiness

The system as described is production-ready for deployment in enterprise networks with the following considerations:

- Scheduler intervals should be tuned based on deployment density and traffic patterns

- QoE threshold (0.55) may require adjustment based on user experience requirements
- Snapshot retention policy should account for available disk space
- API rate limiting should be configured based on expected monitoring load

The system has been validated in testbed environments with up to 5 concurrent stations.

11.4 Final Remarks

The Wi-Fi Steering Engine demonstrates that intelligent, automated client management is achievable using standard 802.11 protocols without requiring client-side software modifications. By combining continuous measurement, multi-dimensional analytics, and threshold-based decision logic, the system proactively maintains connection quality in dense deployments.

The architecture balances complexity with maintainability. While sophisticated in its analytics and decision logic, the system remains conceptually straightforward: measure, analyze, decide, act. This simplicity enables engineers to quickly understand system behavior and diagnose issues when they arise.

The planned post-roam delta analysis enhancement will close the feedback loop and enable truly adaptive steering. Until then, the current implementation provides substantial value by addressing the most common connectivity problems: client stickiness and suboptimal association decisions.