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: $\text{trend} = \text{slope}(Q_1, Q_2, \ldots, Q_N)$
- 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

4.1 QoE Model Overview

Quality of Experience quantifies how well the current AP connection serves the station's needs. The model must balance multiple dimensions: a station with excellent signal but zero throughput has different needs than one with weak signal but active traffic.

The QoE score is defined as a weighted sum of five normalized component scores:

$$Q = w_s S + w_t T + w_r R + w_l L + w_a A \tag{4.1}$$

Where:

- S: Signal Quality Score (0-1)
- T: Throughput Score (0-1)
- R: Reliability Score (0-1)
- L: Latency Score (0-1)
- A: Activity Score (0-1)

The weights are normalized to sum to unity:

$$\sum_i w_i = 1 \tag{4.2}$$

The configured weights reflect relative importance:

$$w_s = 0.28, \quad w_t = 0.32, \quad w_r = 0.15, \quad w_l = 0.15, \quad w_a = 0.10 \tag{4.3}$$

Throughput receives the highest weight (0.32) because it most directly impacts user-perceived performance. Signal quality is second (0.28) as it constrains achievable throughput. Reliability and latency contribute equally (0.15 each) as indicators of connection stability. Activity has the lowest weight (0.10) since idle stations may have low activity but still maintain good connectivity.

4.2 Signal Quality Component

Signal quality measures RF link quality using RSSI from link measurements. The score maps RSSI to the range [0, 1] using linear interpolation:

$$S = \text{clamp}\left(\frac{\text{RSSI} - (-90)}{60}, 0, 1\right) \tag{4.4}$$

This formula maps:

- RSSI ≤ -90 dBm $\rightarrow$ score $= 0.0$ (unusable)

- RSSI ≥ -30 dBm $\rightarrow$ score = 1.0 (excellent)
- Intermediate values $\rightarrow$ linear interpolation

The -90 dBm lower bound represents the typical noise floor where reliable communication becomes impossible. The -30 dBm upper bound represents near-AP conditions where signal is no longer the limiting factor.

For example:

$$\text{RSSI} = -50 \text{ dBm} \rightarrow S = \frac{-50-(-90)}{60} = \frac{40}{60} = 0.67$$
$$\text{RSSI} = -70 \text{ dBm} \rightarrow S = \frac{-70-(-90)}{60} = \frac{20}{60} = 0.33$$

4.3 Throughput Component

Throughput score estimates the station's current data rate as a fraction of its theoretical maximum. It uses both TX (transmit to AP) and RX (receive from AP) bitrates:

$$T = \text{clamp}\left(\frac{\sqrt{\text{tx\_bitrate} \cdot \text{rx\_bitrate}}}{\text{phy\_peak}}, 0, 1\right) \tag{4.5}$$

The geometric mean $\sqrt{\text{tx} \cdot \text{rx}}$ ensures both directions contribute to the score. A connection with high TX but low RX (or vice versa) will score lower than one with balanced rates.

The `phy_peak` value represents the maximum theoretical bitrate for the station's capabilities (e.g., 866 Mbps for 2-stream 802.11ac, 1200 Mbps for 2-stream 802.11ax). This normalization accounts for device heterogeneity: a phone maxing out at 433 Mbps should score the same as a laptop maxing out at 1733 Mbps.

For example, a station with:

$$\text{tx\_bitrate} = 300 \text{ Mbps}$$
$$\text{rx\_bitrate} = 300 \text{ Mbps}$$
$$\text{phy\_peak} = 866 \text{ Mbps}$$

Yields:

$$T = \frac{\sqrt{300 \cdot 300}}{866} = \frac{300}{866} = 0.35 \tag{4.6}$$

4.4 Reliability Component

Reliability measures connection stability using retry rates and frame check sequence (FCS) errors. High retry rates indicate the AP must retransmit packets frequently due to poor RF conditions. FCS errors indicate corrupted frames that cannot be decoded.

The reliability score is computed as:

$$R = 1 - (\alpha \cdot \text{retry\_rate} + \beta \cdot \text{fcs\_rate}) \tag{4.7}$$

Where:

$$\alpha = 0.6, \quad \beta = 0.4 \tag{4.8}$$

The rates are computed from station statistics:

$$\begin{aligned} \text{retry\_rate} &= \frac{\text{retry\_count}}{\text{total\_tx\_packets}} \\ \text{fcs\_rate} &= \frac{\text{fcs\_error\_count}}{\text{total\_rx\_packets}} \end{aligned}$$

Retry rate receives higher weight ($\alpha = 0.6$) because it affects both throughput and latency. FCS errors ($\beta = 0.4$) primarily indicate poor receive sensitivity but may not impact interactive performance as severely.

For example, a station with 1% retry rate and 0.5% FCS error rate:

$$R = 1 - (0.6 \cdot 0.01 + 0.4 \cdot 0.005) = 1 - (0.006 + 0.002) = 0.992 \tag{4.9}$$

This scores highly (0.992) indicating excellent reliability. A station with 15% retries and 5% FCS errors would score:

$$R = 1 - (0.6 \cdot 0.15 + 0.4 \cdot 0.05) = 1 - (0.09 + 0.02) = 0.89 \tag{4.10}$$

4.5 Latency Component

The latency score estimates responsiveness using the inactive time counter. This counter tracks how long the station has been idle (no frames exchanged). Long inactive periods suggest the station is not actively communicating, which may indicate good performance (no backlog) or poor performance (unable to get airtime).

The score is computed as:

$$L = \text{clamp}\left(1 - \frac{\text{inactive\_msec}}{5000}, 0, 1\right) \tag{4.11}$$

The 5000 ms threshold represents the point at which a station is considered effectively disconnected. Stations with recent activity (low inactive time) score high. Completely idle stations score low.

Examples:

$$\begin{aligned} \text{inactive} = 100 \text{ ms} &\rightarrow L = 1 - \frac{100}{5000} = 0.98 \\ \text{inactive} = 2500 \text{ ms} &\rightarrow L = 1 - \frac{2500}{5000} = 0.50 \\ \text{inactive} = 5000 \text{ ms} &\rightarrow L = 1 - \frac{5000}{5000} = 0.00 \end{aligned}$$

4.6 Activity Component

Activity score measures how heavily the station is using the connection. It uses frame counts rather than byte counts to capture both bulk data transfers and interactive traffic:

$$A = \text{clamp}\left(\frac{\text{tx\_packets} + \text{rx\_packets}}{\text{max\_frames}}, 0, 1\right) \tag{4.12}$$

The `max_frames` parameter defines "full utilization" (e.g., 10,000 packets per measurement interval). Stations below this threshold score proportionally. Stations above it saturate at 1.0.

This component helps distinguish idle stations (which may have high signal/throughput scores but aren't actually using the network) from active ones (which need good QoE to maintain performance).

4.7 Historical Trend Analysis

Beyond the instantaneous QoE score, the system tracks historical behavior to identify degradation patterns. For each station, it maintains:

4.7.1 Moving Average

The moving average smooths short-term fluctuations:

$$\bar{Q}_t = \frac{1}{k}\sum_{i=0}^{k-1} Q_{t-i} \tag{4.13}$$

where k is the window size (typically 10 samples). This provides a stable baseline for comparison.

4.7.2 Trend Computation

The trend is the slope of a linear regression through the last k samples:

$$\text{trend} = \frac{k\sum t_i Q_i - (\sum t_i)(\sum Q_i)}{k\sum t_i^2 - (\sum t_i)^2} \tag{4.14}$$

Positive trends indicate improving conditions, negative trends indicate degradation. A station with QoE $= 0.6$ and strong negative trend is a better steering candidate than one with QoE $= 0.55$ and positive trend.

4.7.3 Volatility

Volatility is the sample variance:

$$\sigma^2 = \frac{1}{k}\sum_{i=0}^{k-1}(Q_{t-i} - \bar{Q}_t)^2 \tag{4.15}$$

High volatility suggests unstable conditions (interference, mobility) that may resolve without steering. Low volatility with poor QoE suggests persistent problems requiring intervention.

4.8 Missing Feature: Post-Roam QoE Delta Analysis

After the system steers a station from AP_1 to AP_2, it should measure the QoE improvement:

$$\Delta Q = Q_{\text{after}} - Q_{\text{before}} \tag{4.16}$$

This delta would inform future decisions: if AP_2 consistently produces poor ΔQ for certain stations, it should rank lower in future neighbor lists. Similarly, successful roams would reinforce AP_2's ranking.

Current Implementation Status: This analysis is not yet integrated. The system steers based on pre-roam QoE and neighbor rankings but does not adaptively learn from outcomes. Post-roam QoE is computed and stored, but not fed back into the decision logic.

Impact: The system cannot distinguish between APs that appear good (high neighbor RSSI) but perform poorly in practice. It also cannot detect when steering actually makes things worse.

Planned Implementation: Future releases will track per-AP success rates and incorporate them into the neighbor ranking formula, creating a feedback loop that improves steering accuracy over time.

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:

```
1: station ← next connected station
2: qoe ← fetch current QoE(station)
3: if qoe is None then
4:     skip {No QoE data yet}
5: end if
6: if qoe.score > 0.55 then
7:     skip {QoE acceptable}
8: end if
9: neighbors ← fetch ranked neighbors(station)
10: if neighbors is empty then
11:     skip {No steering candidates}
12: end if
13: request ← build_bss_tm_request(station, neighbors)
14: controller.send(request)
15: 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:

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

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. Future Enhancements

10.1 Post-Roam QoE Delta Analysis

The primary missing piece is the post-roam QoE delta model. Currently, the system measures QoE before and after steering but does not feed this information back into the decision logic.

10.1.1 Proposed Implementation

After a station roams from AP_{old} to AP_{new}:

1. Wait for the roaming process to complete (association, DHCP, etc.)
2. Measure QoE for a stabilization period (30-60 seconds)
3. Compute the delta:
$$\Delta Q = Q_{\text{after}} - Q_{\text{before}} \tag{10.1}$$
4. Record the delta in a per-AP performance database:

```
roam_history[(station, AP_old, AP_new)] = {
    "delta_qoe": ..Q,
    "timestamp": t,
    "before": Q_before,
    "after": Q_after
}
```

10.1.2 Incorporating Delta into Ranking

The neighbor ranking formula would be extended:

$$N_i = \gamma_1 \cdot \text{RSSI}_i + \gamma_2 \cdot \text{capacity}_i - \gamma_3 \cdot \text{load}_i + \gamma_4 \cdot \text{history}_i \tag{10.2}$$

Where `history`$_i$ is computed from past roaming outcomes:

$$\text{history}_i = \frac{1}{K} \sum_{k=1}^{K} \Delta Q_k \tag{10.3}$$

The sum is over the K most recent roams to AP_i by this station (or similar stations). APs that consistently produce positive ΔQ receive higher scores; APs that produce negative ΔQ are penalized.

10.1.3 Benefits

This enhancement would enable several improvements:

- **Adaptive Learning**: The system learns which APs perform better than their measurements suggest

- **Problem Detection**: APs with good RSSI but poor actual performance (e.g., due to hidden nodes, backend congestion) are automatically down-ranked
- **Personalization**: Per-station historical tracking accounts for device-specific behaviors
- **Feedback Loop**: Poor steering decisions self-correct over time

10.1.4 Implementation Challenges

Several technical challenges must be addressed:

1. **Roam Detection**: The system must reliably detect when a station has completed roaming (not just sent a reassociation frame)
2. **Causality**: The post-roam QoE may be affected by factors other than the AP change (user movement, interference)
3. **Stabilization Time**: How long to wait before measuring post-roam QoE? Too short and it captures transient effects; too long and other factors dominate
4. **Statistical Significance**: How many samples are needed before trusting the historical average?

10.1.5 Architectural Changes Required

Implementing post-roam delta analysis requires modifications to:

- **NeighborRankingDB**: Add historical performance storage and lookup methods
- **TransitionManagementEngine**: Track steering requests and match them with subsequent roaming events
- **QoE Scheduler**: Detect post-roam stabilization and trigger delta computation
- **Neighbor Ranking Algorithm**: Incorporate the new history term in the scoring formula

The changes are substantial but well-defined. The modular architecture supports these enhancements without requiring a complete redesign.

10.2 Possible Extentions

Beyond post-roam delta analysis, several other enhancements can be experimented with:

10.2.1 Load Balancing

Currently, steering is purely QoE-based. An AP with 50 stations and good QoE won't trigger steering even if a neighbor has zero stations. There can be enhancements that:

- Factor AP load into the neighbor ranking algorithm
- Proactively move stations from overloaded APs even if individual QoE is acceptable
- Implement fairness algorithms to balance load across the infrastructure

10.2.2 Predictive Steering

The current reactive model waits for QoE to degrade before steering. A predictive model would:

- Use trend analysis to forecast future QoE degradation
- Steer preemptively before user-perceived impact
- Reduce the frequency of severely degraded connections

This requires machine learning models trained on historical QoE time series data.

10.2.3 Client Capability Fingerprinting

The current system has limited knowledge of client capabilities. Future versions will:

- Extract supported PHY modes, spatial streams, and channel widths from association frames
- Use this information to avoid steering clients to APs they cannot fully utilize
- Optimize AP-client pairing based on capability matching

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.