Wi-Fi Aware Architecture

This document details the architecture for Wi-Fi Aware in Android 17 and higher, which mitigates limitations inherent in the firmware-centric architecture in Android 16 or lower.

The Wi-Fi Aware implementation in Android 16 or lower relies on a firmware-centric design (see Figure 1), where critical control functions reside within the firmware. This approach poses challenges for chip manufacturers, leading to an elevated memory footprint, inconsistent support for optimization techniques, protracted feature deployment timelines, and suboptimal data throughput. Furthermore, this existing model complicates harmonization with other similar Android technologies, such as Wi-Fi Direct, Station/AP, and Unsynchronized Service Discovery (USD), hindering potential future efforts to share foundational elements across these connectivity options.

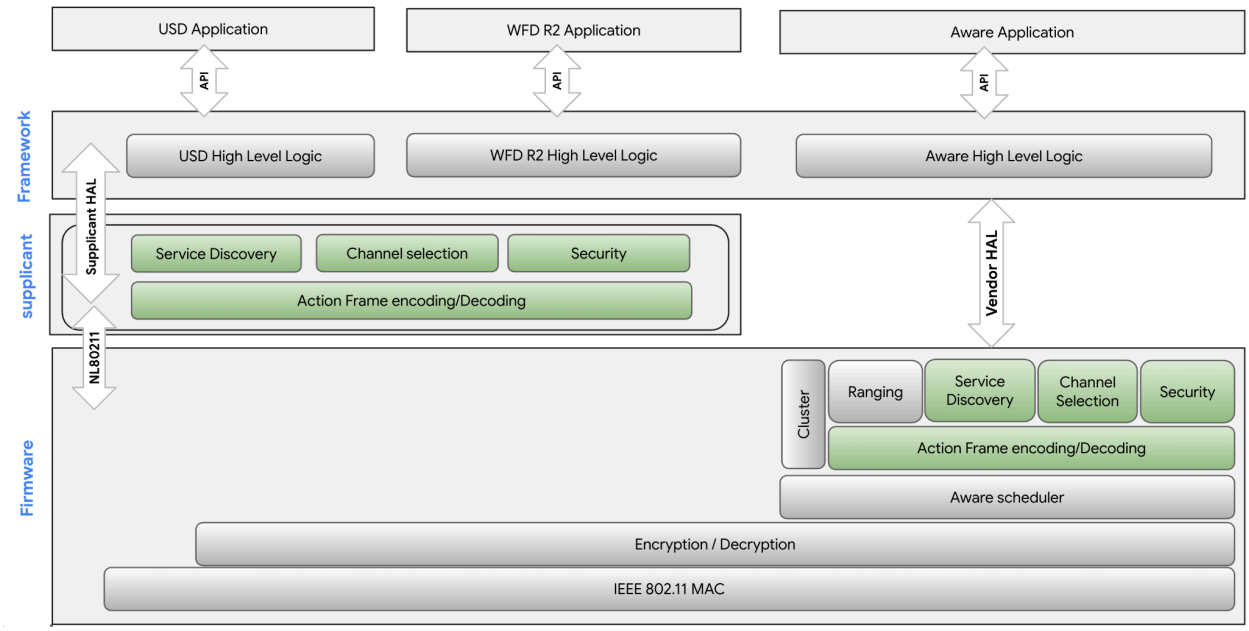

Figure 1. Wi-Fi Aware architecture in Android 16.

High-level architecture

The following diagram shows the high-level view of the changes in the Aware architecture for Android 17 and higher compared to the architecture in Android 16 or lower.

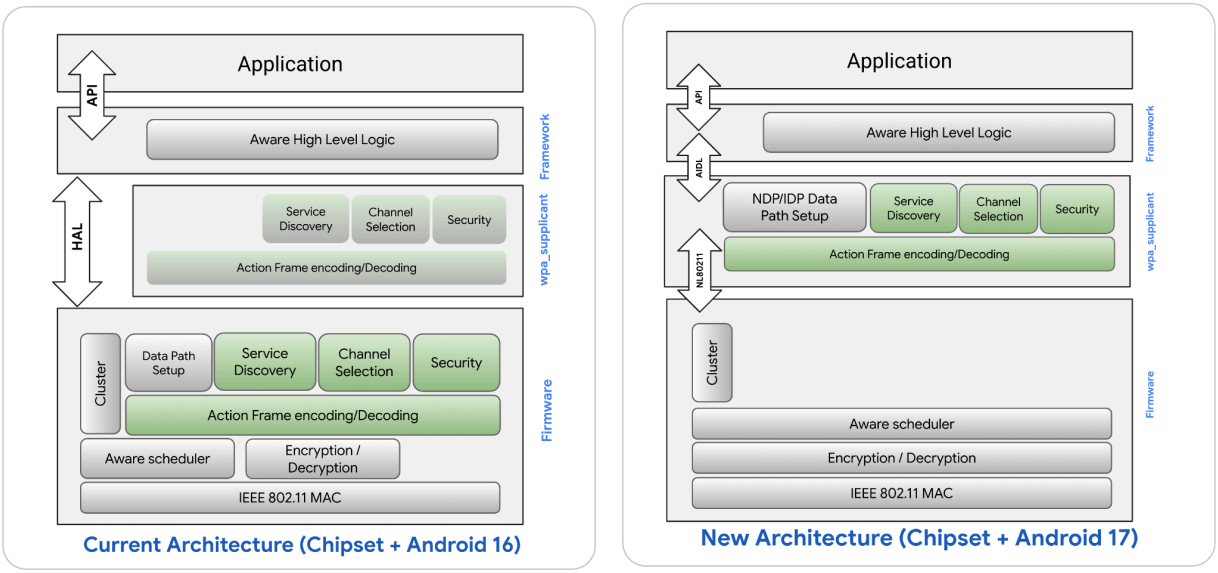

Figure 2. Wi-Fi Aware architecture in Android 17.

The architectural approach for Android 17 and higher signifies a fundamental restructuring of Wi-Fi Aware, transferring essential control functions to the `wpa_supplicant` and Android framework from firmware.

The Wi-Fi Aware Android framework manages the SDK API abstraction and key high-level functionalities such as NIK (NAN Identity Key) generation, NPKSA (NAN Pairwise Key Security Association) caching, MAC address randomization and data path schedule negotiation.

The specific control functionalities being relocated to the `wpa_supplicant` include a broad spectrum of operations. Mechanisms for service discovery, critical for device identification and service accessibility, are managed by the `wpa_supplicant`. The handling of follow-up action frames, which are pivotal for initiating subsequent interactions and actions after discovery, is also centralized here. Additionally, security protocols, vital for maintaining secure communication channels, and the complex procedures for establishing data paths, which facilitate the connection between devices, are under the control of the `wpa_supplicant`. The setup of data path and ranging functions are likewise managed by the supplicant.

Conversely, the firmware retains accountability for the foundational physical layer and low-level protocol operations of the Wi-Fi Aware functionality. This encompasses the oversight of the NAN Cluster, a core element for device coordination and time synchronization in Wi-Fi Aware.

The firmware also maintains the responsibility for the periodic broadcasting of beacons, which publicize device presence and capabilities within the network. Role switching mechanisms, allowing devices to dynamically alter roles within the network, and cluster merge processes, which unify multiple NAN Clusters, are also managed by the firmware. The transmission and reception of management and data frames, the basic units of communication, as well as data encryption and decryption for security, and the execution of Round-Trip Time (RTT) measurements, for precise distance assessment, remain the firmware's responsibilities.

The architecture replaces the vendor HAL with the Supplicant AIDL.

Several Wi-Fi Aware applications use a separate, energy-efficient method for device discovery (such as BLE). Upon discovery, a Wi-Fi Aware connection is established to facilitate data transfer. After the data exchange is completed, the Wi-Fi Aware connection is terminated. So a firmware offload is not necessary for most of the applications.

Although this design doesn't prevent the offloading of Aware operations, which often wakes up the application processor, the supplicant retains the flexibility to transfer various stages to the firmware if the chipset allows. This is achieved through the exchange of capabilities between the supplicant and firmware.

The architecture in Android 17 and higher offers distinct advantages, including reduced implementation complexity for chip manufacturers, a diminished memory footprint, streamlined incorporation of optimization strategies, and enhanced future compatibility with other technologies.

Key architecture changes in Android 17

The following are key architecture changes in Android 17.

- All `wpa_supplicant` changes are upstreamable to be part of the standard `hostap` open source project.
- The architecture uses upstreamed standard `nl80211` commands and events.
- Because this architecture is adopted by legacy devices, the `nl80211` commands will be back-ported with vendor NL commands (OUI = Android).
- Vendor NL commands required by `wpa_supplicant` can be added to `external/wpa_supplicant_8/src/common/android-vendor.h`. The Android framework keeps a master copy of this file.
- Usage of mainline supplicant removes the dependency on HAL freeze.

Detailed architecture

This section outlines the key elements and workflows of Wi-Fi Aware operation. It encompasses critical phases from device capability setup and cluster creation to service discovery, interaction, pairing, and data path establishment.

In particular, this details how device capabilities are acquired and stored, how applications connect and form clusters, the fundamental publish/subscribe model for service discovery, the notification process for discovered services, the role of follow-up messages in enabling further communication and activities, the device pairing steps, and the procedures for setting up secure data paths for subsequent interactions.

Initialization

As part of the existing framework codebase, upon starting `wpa_supplicant`, the framework calls into the supplicant requesting driver capabilities (similar to the `getWpaDriverCapabilities` HAL API used with the vendor supplicant), which returns the capabilities set in the supplicant at startup. This informs the framework whether the chip supports the supplicant-based Aware architecture so it can act accordingly.

As outlined in High-level architecture, parts of the Wi-Fi Aware stack are implemented by `wpa_supplicant` and the firmware. The supplicant and firmware inherently have their own features and limitations, such as the maximum number of concurrent publish/subscribe sessions, maximum service name length, supported frequencies, and other technical specifications.

During initialization, once the framework establishes a NAN interface through the Vendor HAL `IWifiChip.createNanIface()` and registers it with the supplicant using `IMainlineSupplicant.addNanInterface()`, the supplicant proceeds to query the firmware or driver through the `NL80211_CMD_GET_WIPHY` command (provided `CONFIG_NAN` is enabled) and caches the firmware-defined capabilities.

When the Android framework's Wi-Fi Aware service starts (controlled by `PackageManager.FEATURE_WIFI_AWARE`), the framework registers an event callback and triggers a request to the supplicant to retrieve the device's Wi-Fi Aware capabilities through `ISupplicantNanIface.getCapabilitiesRequest()`.

The supplicant relays the capability data back to the framework using the `ISupplicantNanIfaceEventCallback.notifyCapabilitiesResponse()` callback.

Upon receipt, the framework caches this information in memory. This cached record serves as the framework's source for responding to subsequent application queries (for example, when an application calls `WifiAwareManager.getCharacteristics()`) without requiring continuous communication with the supplicant. To conclude the flow, the framework tears down the temporary NAN interface by calling `IMainlineSupplicant.removeNanInterface()` and `IWifiChip.removeNanIface()`.

For a given device and firmware version, the Wi-Fi Aware capabilities are static. They don't change dynamically during device operation. Therefore, caching the capabilities in framework memory after the initial query is efficient.

Here is the call flow for caching Aware capabilities in the framework as part of initialization. The capabilities queried by the framework are documented in the AIDL interface section.

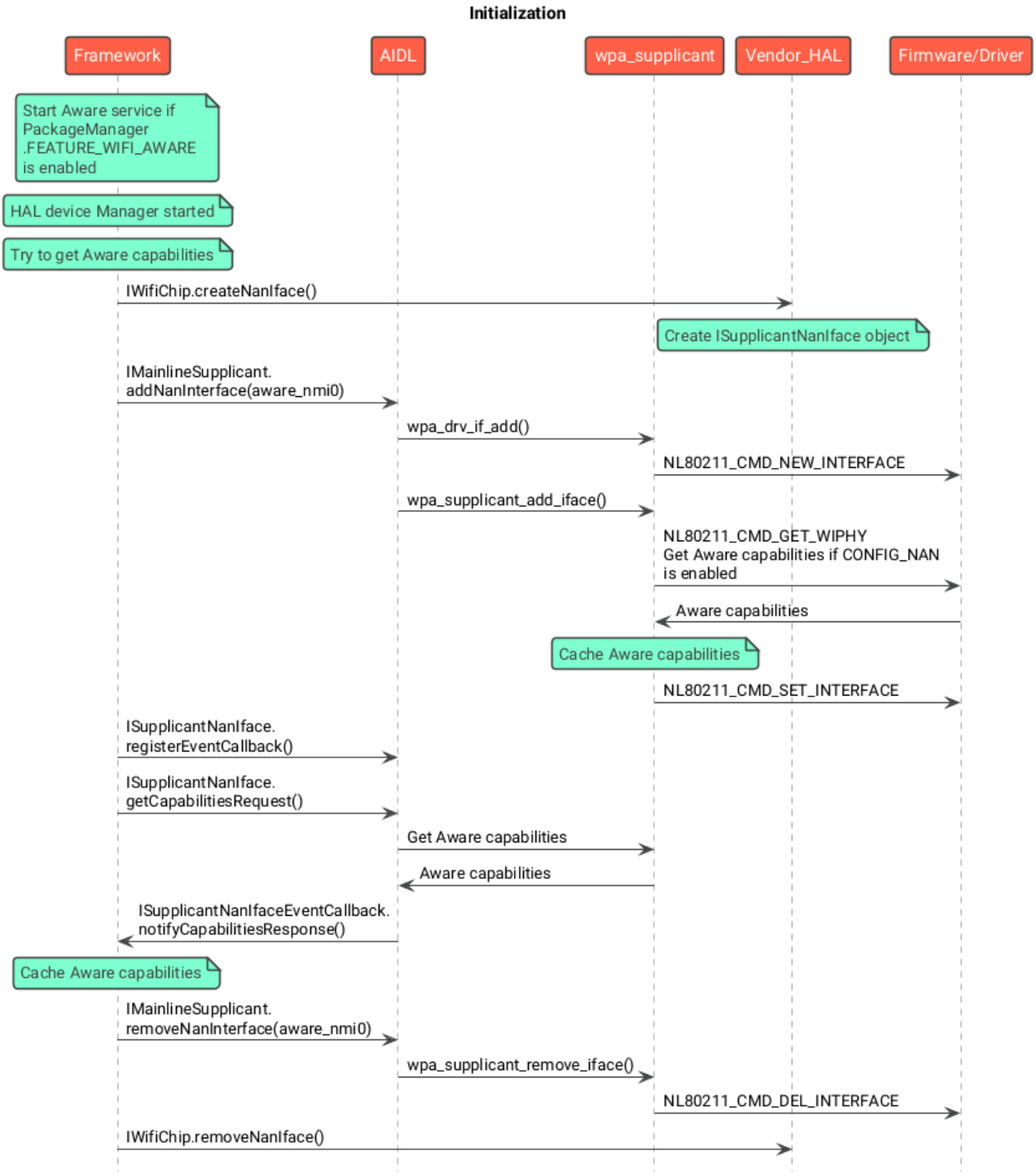

Figure 3. Wi-Fi Aware initialization flow.

AIDL

`IWifiChip` has vendor HAL APIs to create the Aware control interface (`aware_nmi0`), which is statically created by vendor software. The interface name is saved in the Android property `wifi.aware.interface`. See `getWlanIfaceName()` in `hardware/interfaces/wifi/aidl/default/wifi_chip.cpp.`

Note: The creation and activation of the interface are located in the supplicant, similar to the process used for Wi-Fi Direct. However, the `IWifiChip` APIs, which are utilized by the HAL device manager within the framework, are still required.

Vendor HAL methods to create and remove the NAN interface are defined in `IWifiChip.aidl`. Interface management for `wpa_supplicant` is provided by `IMainlineSupplicant.aidl`.

The Aware interface (`ISupplicantNanIface`) enables supplicant AIDL methods to request and receive Aware capabilities through callback responses from the supplicant.

The callbacks are defined in `ISupplicantNanIfaceEventCallback.aidl`.

Wi-Fi Aware capabilities are defined in `NanCapabilities.aidl`.

Attach (cluster building)

The Wi-Fi Aware attach process is the initial step for an Android app to enable and utilize Wi-Fi Aware functionality on a device. For SDK API usage details, see Obtain a session.

A core aspect of the attach phase is the formation of clusters of neighboring Aware devices. When an application invokes the `attach()` API, the underlying Wi-Fi Aware system (managed by the firmware) attempts to:

- **Framework setup:** On the initial attach request, the framework creates a NAN interface through the Vendor HAL and triggers interface addition in the supplicant through the `IMainlineSupplicant.addNanInterface()` AIDL call.
- **Interface creation:** `wpa_supplicant` executes `wpa_drv_if_add()` and `wpa_supplicant_add_iface()`, dispatching `NL80211_CMD_NEW_INTERFACE` and `NL80211_CMD_SET_INTERFACE` to the kernel to instantiate and configure the `aware_nmi0` management interface.
- **Enable Aware:** The framework registers for event callbacks and initiates

`ISupplicantNanIface.enableRequest()`. This triggers `wpas_nan_start()` in the supplicant, which sends `NL80211_CMD_START_NAN` to the firmware. The firmware activates the Aware engine and returns a status response.

- **Data path initialization:** The framework iteratively generates required NAN Data Interfaces (NDIs), such as `aware_dataX`, through `createDataInterfaceRequest()`. Each call prompts the supplicant to add a virtual network interface using `NL80211_CMD_NEW_INTERFACE`.
- **Session established:** Following management and data interface readiness, the framework notifies the app through `AttachCallback.onAttached()`, successfully establishing a `WifiAwareSession` session.
- **Join an existing cluster:** If other Wi-Fi Aware devices are operating in the vicinity and forming a cluster, the device attempts to join that cluster using the existing cluster ID.
- **Create a new cluster:** If the device is the first one enabling Aware in the area, it initiates the formation of a new cluster with a newly generated cluster ID. This clustering behavior is managed by the firmware. The framework and privileged apps can configure key characteristics of the cluster; regular apps have no direct control over configuration parameters.

Apps can also listen for cluster ID and discovery interface MAC address (identity) changes (see `attach()`).

From the app's perspective, the outcome of the attach operation is a `WifiAwareSession` session.

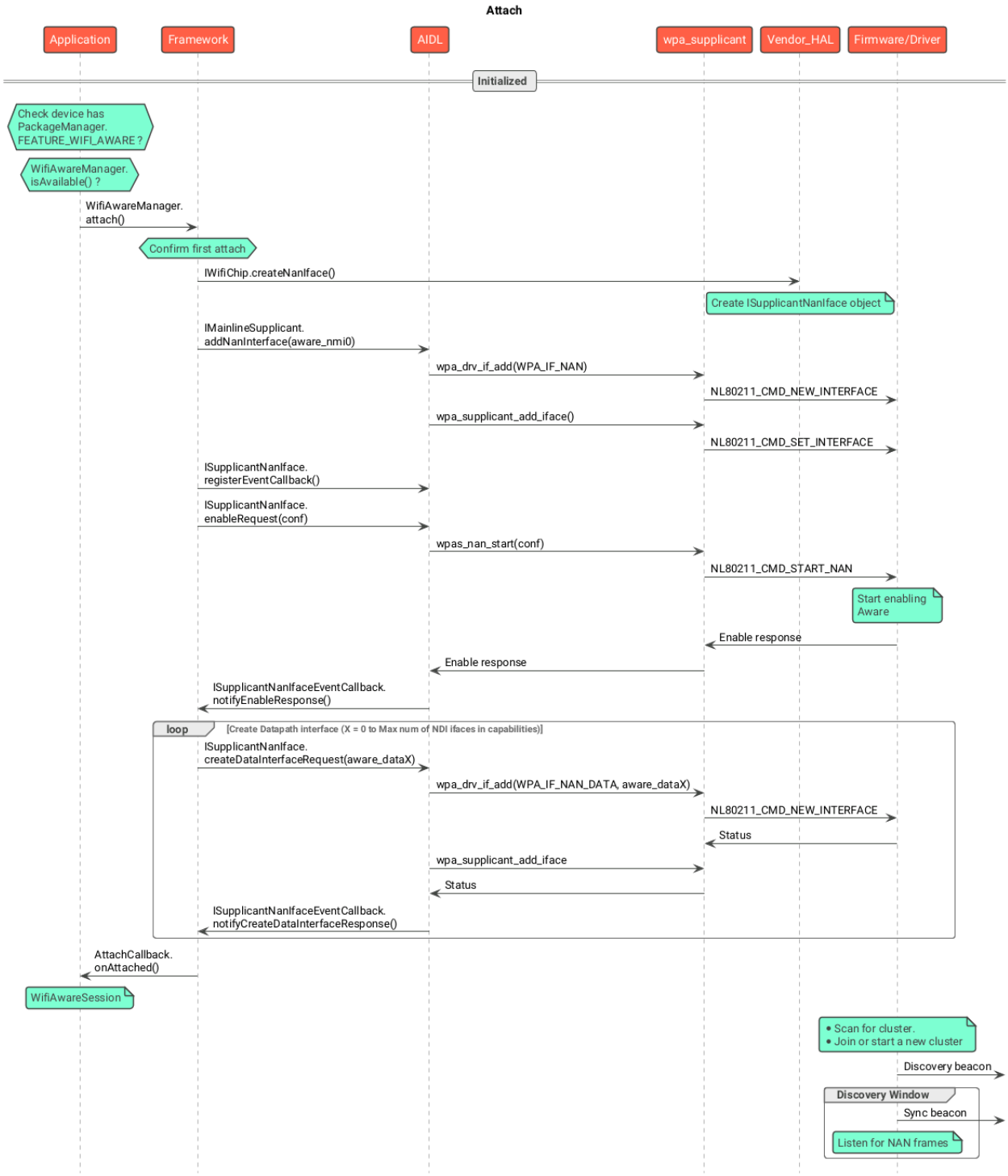
Attach
Application
Framework
AIDL
wpa_supplicant
Vendor_HAL
Firmware/Driver
Initialized
Check device has PackageManager. FEATURE_WIFI_AWARE ?
WifiAwareManager. isAvailable() ?
WifiAwareManager. attach()
Confirm first attach
IWifiChip.createNanIface()
Create ISupplicantNanIface object
IMainlineSupplicant. addNanInterface(aware_nmi0)
wpa_drv_if_add(WPA_IF_NAN)
NL80211_CMD_NEW_INTERFACE
wpa_supplicant_add_iface()
NL80211_CMD_SET_INTERFACE
ISupplicantNanIface. registerEventCallback()
ISupplicantNanIface. enableRequest(conf)
wpas_nan_start(conf)
NL80211_CMD_START_NAN
Start enabling Aware
Enable response
Enable response
ISupplicantNanIfaceEventCallback. notifyEnableResponse()
loop
[Create Datapath interface (X = 0 to Max num of NDI ifaces in capabilities)]
ISupplicantNanIface. createDataInterfaceRequest(aware_dataX)
wpa_drv_if_add(WPA_IF_NAN_DATA, aware_dataX)
NL80211_CMD_NEW_INTERFACE
Status
wpa_supplicant_add_iface
Status
ISupplicantNanIfaceEventCallback. notifyCreateDataInterfaceResponse()
AttachCallback. onAttached()
WifiAwareSession
• Scan for cluster.
• Join or start a new cluster
Discovery beacon
Discovery Window
Sync beacon
Listen for NAN frames

Figure 4. Wi-Fi Aware attach operation flow.

An Aware cluster is a shared resource. If the device is already attached to a cluster and a subsequent request is initiated, the framework manages the overlapping request as follows:

- **Configuration check:** Upon detecting concurrent attachment requests, the framework evaluates session parameters to determine if a reconfiguration of global cluster state is warranted.
- **Reconfiguration (if needed):** If an update is required, the framework dispatches the modified configuration through `ISupplicantNanIface.configRequest()`. The supplicant then signals the firmware to alter cluster behavior using `NL80211_CMD_CHANGE_NAN_CONFIG`, confirming status through the `ISupplicantNanIfaceEventCallback.notifyConfigResponse()` callback.
- **Session establishment:** Following successful configuration (or immediately if no changes are necessary), the framework notifies the application through `AttachCallback.onAttached()`, providing a new `WifiAwareSession` instance.

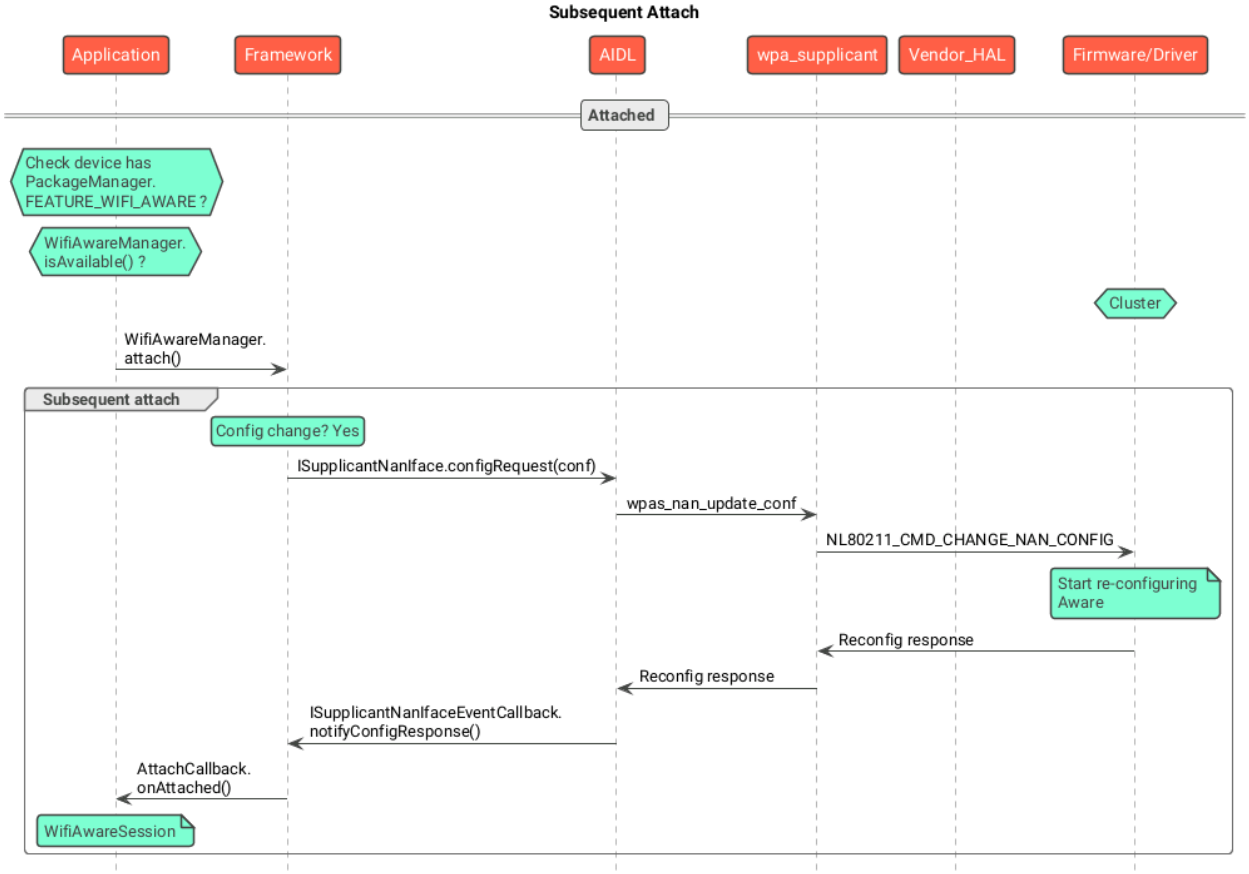

Figure 5. Wi-Fi Aware subsequent attach operation flow.

If an app is registered for cluster events, it receives notifications when a cluster is started or joined, or when an interface address changes. MAC addresses of the Wi-Fi Aware discovery (NMI) and data interfaces (NDP) are randomized (see MAC address randomization). The following events are supported:

- Cluster ID changes triggered by either a cluster started event or cluster joined event through the `NL80211_CMD_NAN_CLUSTER_JOINED` Netlink command, containing the 6-byte Cluster ID.

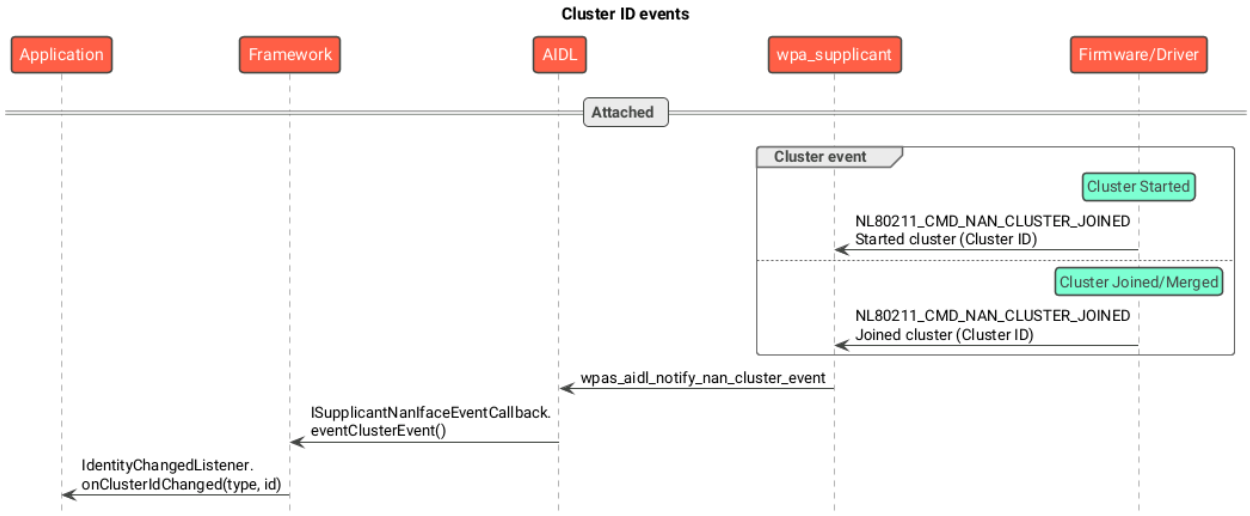

Figure 6. Wi-Fi Aware cluster ID events flow.

- Identity changes resulting from the device joining a cluster, starting a cluster, or discovery interface address changes (NMI addresses are randomized at regular intervals).

MAC address randomization is controlled by firmware in Android 16 or lower. In Android 17 and higher, MAC address randomization is managed by the framework (see MAC address randomization.

Apps with elevated privileges have the ability to configure settings that control Wi-Fi Aware device operation after joining or starting a cluster.

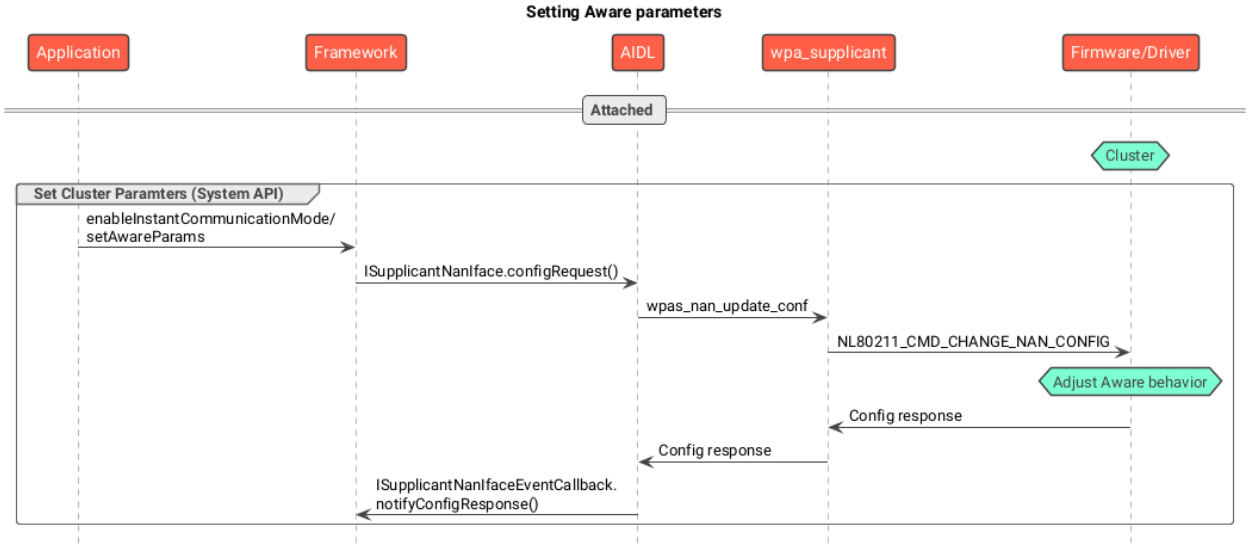

Figure 7. Setting Wi-Fi Aware parameters flow.

The `WifiAwareSession.close()` method terminates a Wi-Fi Aware session. If this was the last active session on the device, the framework proceeds to fully disable the Aware subsystem and clean up all associated interfaces:

- **Disabling the engine:** The framework sends `IMainlineSupplicant.disableRequest()` to the supplicant, which triggers the `NL80211_CMD_STOP_NAN` Netlink command to the firmware. This stops the Discovery Engine (DE), ceases all beaconing, and causes the device to leave the cluster. The supplicant confirms this through the `notifyDisableResponse()` callback.
- **Data interface cleanup:** The framework iterates through all active data interfaces (`aware_dataX`) and requests their removal through `deleteDataInterfaceRequest()`, resulting in the deletion of virtual network interfaces in the kernel.
- **Management interface removal:** The framework removes the primary management Aware interface through `IMainlineSupplicant.removeNanInterface()` and the vendor HAL's `IWifiChip.removeNanIface()`, fully deallocating hardware resources.

Note: This action cancels all ongoing operations, including active publish and subscribe sessions, and data links.

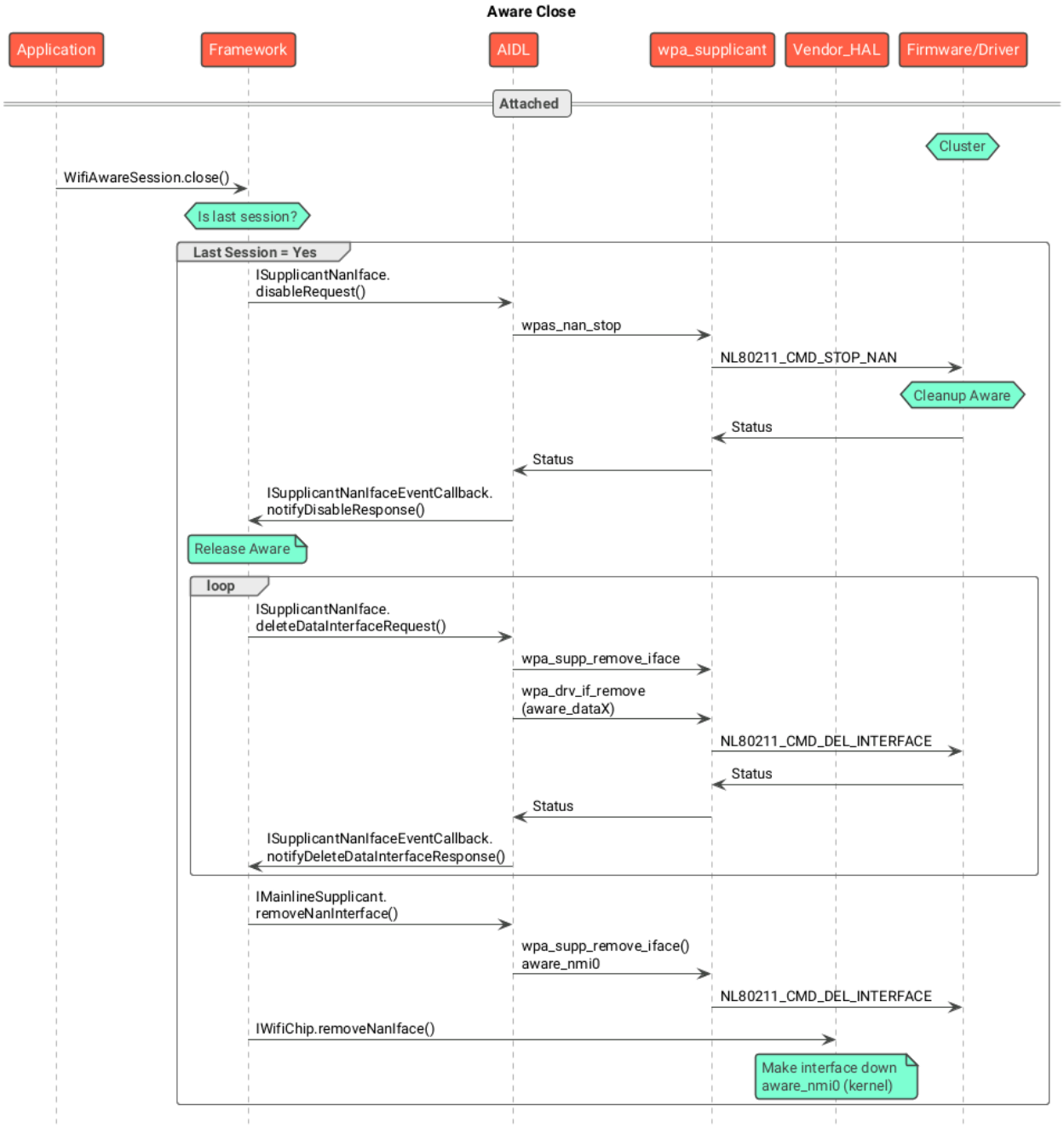

Figure 8. Wi-Fi Aware close flow.

AIDL

The NAN interface supports the following AIDL APIs to enable and configure the Aware device:

- ISupplicantNanIface.aidl
- NanBandSpecificConfig.aidl
- NanClusterEventInd.aidl
- NanClusterEventType.aidl
- NanConfigRequest.aidl
- NanEnableRequest.aidl

Callbacks for enable, config, and cluster events (start, join, mac address change) are defined in ISupplicantNanIfaceEventCallback.aidl.

Publish and subscribe

Wi-Fi Aware uses a publish-subscribe model for service discovery.

Publisher: A device that advertises a service. It publishes information about the service it offers.

Here is the call flow for a publisher:

1. **App publishes:** The application requests to publish a service using `WifiAwareManager.publish()`.
2. **Framework request:** The framework sends the publish configuration to the supplicant through the `ISupplicantNanIface.startPublishRequest()` AIDL call.
3. **Supplicant configures:** `wpa_supplicant` (specifically `wpas_nan_publish`) processes the request locally. In this userspace-driven flow, the supplicant registers the service in its internal discovery table but doesn't send an immediate Netlink command to the firmware at this stage.
4. **Response to app:** The supplicant returns a status, and the framework notifies the app that the publish session has started through `DiscoverySessionCallback.onPublishStarted()`.
5. **Discovery window loop:**
    - **DW notification:** The firmware or driver periodically notifies `wpa_supplicant` of upcoming Discovery Window (DW) timings using `NL80211_CMD_NAN_NEXT_DW_NOTIFICATION`.
    - **Unsolicited path:** For unsolicited publishers, the supplicant builds a Service

Discovery Frame (SDF) and sends it to the firmware for transmission using `NL80211_CMD_FRAME`.

- **Solicited path:** When the firmware receives a peer's subscribe frame, it passes it to the supplicant through `NL80211_CMD_FRAME` (RX). The supplicant checks match criteria; if matched, it sends a solicited publish response frame back to the firmware through `NL80211_CMD_FRAME` (TX). The firmware then provides transmission confirmation through `NL80211_CMD_FRAME_TX_STATUS`.

6. **Reconfiguration:** Application-initiated updates to Aware parameters flow through `ISupplicantNanIface.configRequest()`. This triggers `wpas_nan_update_conf` in the supplicant, which sends the `NL80211_CMD_CHANGE_NAN_CONFIG` Netlink command to the firmware. The firmware adjusts its Aware behavior and returns a response, which the supplicant forwards back to the framework through `ISupplicantNanIfaceEventCallback.notifyConfigResponse()`.

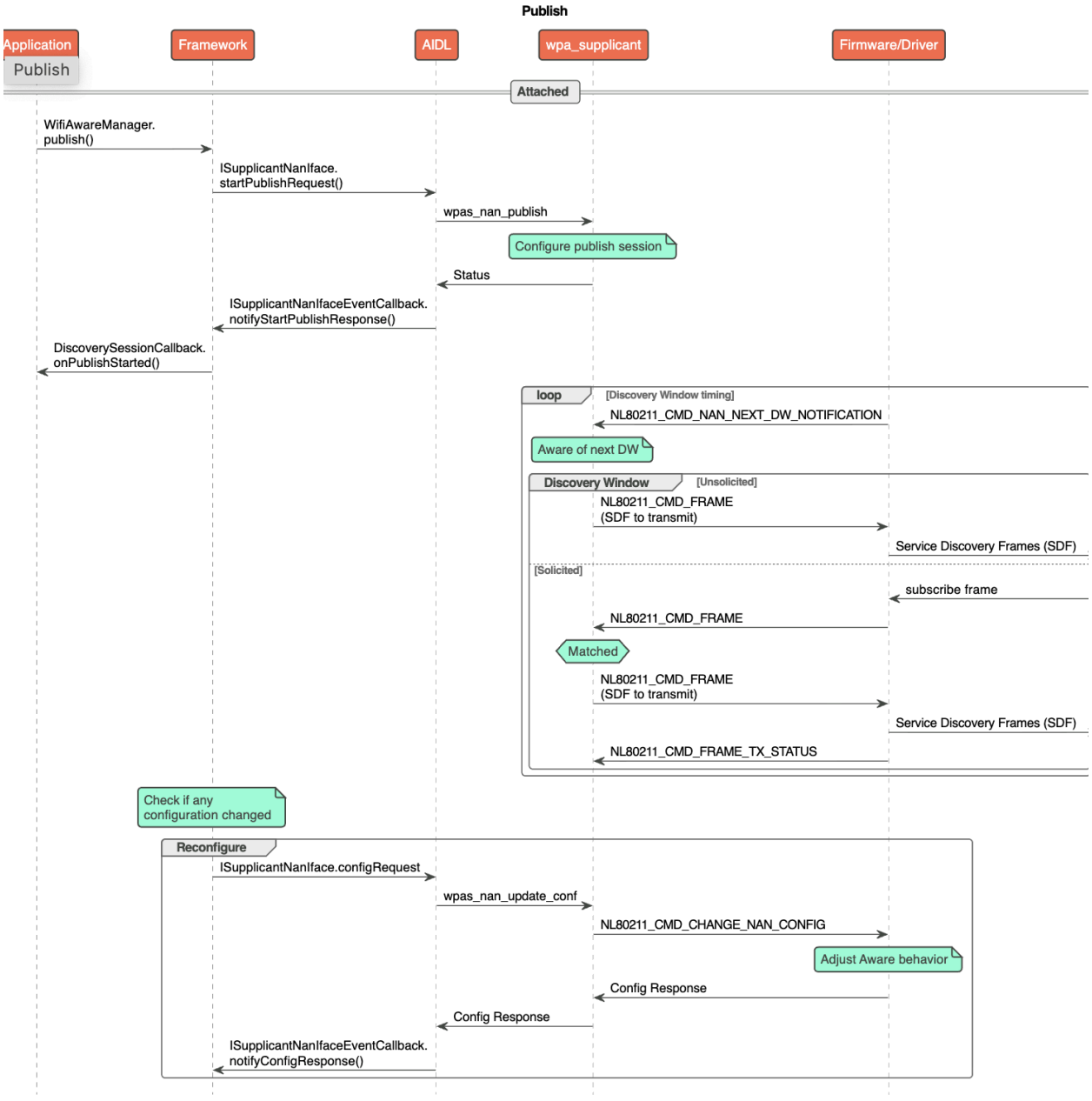

Figure 9. Wi-Fi Aware publish flow.

Subscriber: A device looking for a specific service. It subscribes to the service it is interested in. When a subscriber discovers a publisher offering a matching service within range, the

subscriber receives a notification.

Here is the call flow for a subscriber:

1. **App subscribes:** The app requests to subscribe to a service using `WifiAwareManager.subscribe()`.
2. **Framework request:** The framework sends the subscribe configuration to the supplicant through the `ISupplicantNanIface.startSubscribeRequest()` AIDL call.
3. **Supplicant configures:** `wpa_supplicant` (specifically `wpas_nan_subscribe`) processes the request locally. In this userspace-driven flow, the supplicant registers the subscribe session in its internal discovery table but doesn't send an immediate Netlink command to the firmware at this stage.
4. **Response to app:** The supplicant returns a status, and the framework notifies the app that the subscribe session has started through `DiscoverySessionCallback.onSubscribeStarted()`.
5. **Discovery window loop:**
    - **DW notification:** The firmware or driver periodically notifies `wpa_supplicant` of upcoming Discovery Window timings using `NL80211_CMD_NAN_NEXT_DW_NOTIFICATION`.
    - **Active path:** For active subscribers, the supplicant builds a Subscribe Service Discovery Frame (SDF) and sends it to the firmware for transmission using `NL80211_CMD_FRAME`.
    - **Passive path:** In passive mode, the firmware listens for incoming frames. When it receives a peer's unsolicited publish frame, it passes it to the supplicant through `NL80211_CMD_FRAME` (RX). The supplicant checks match criteria; if matched, it notifies the framework of the discovery result without transmitting a frame back to the peer.
6. **Reconfiguration:** If configuration changes are required, the framework calls `ISupplicantNanIface.configRequest()`. This triggers `wpas_nan_update_conf` in the supplicant, which sends the `NL80211_CMD_CHANGE_NAN_CONFIG` Netlink command to the firmware. The firmware adjusts its Aware behavior and returns a response, which is forwarded back to the framework through the `notifyConfigResponse()` callback.

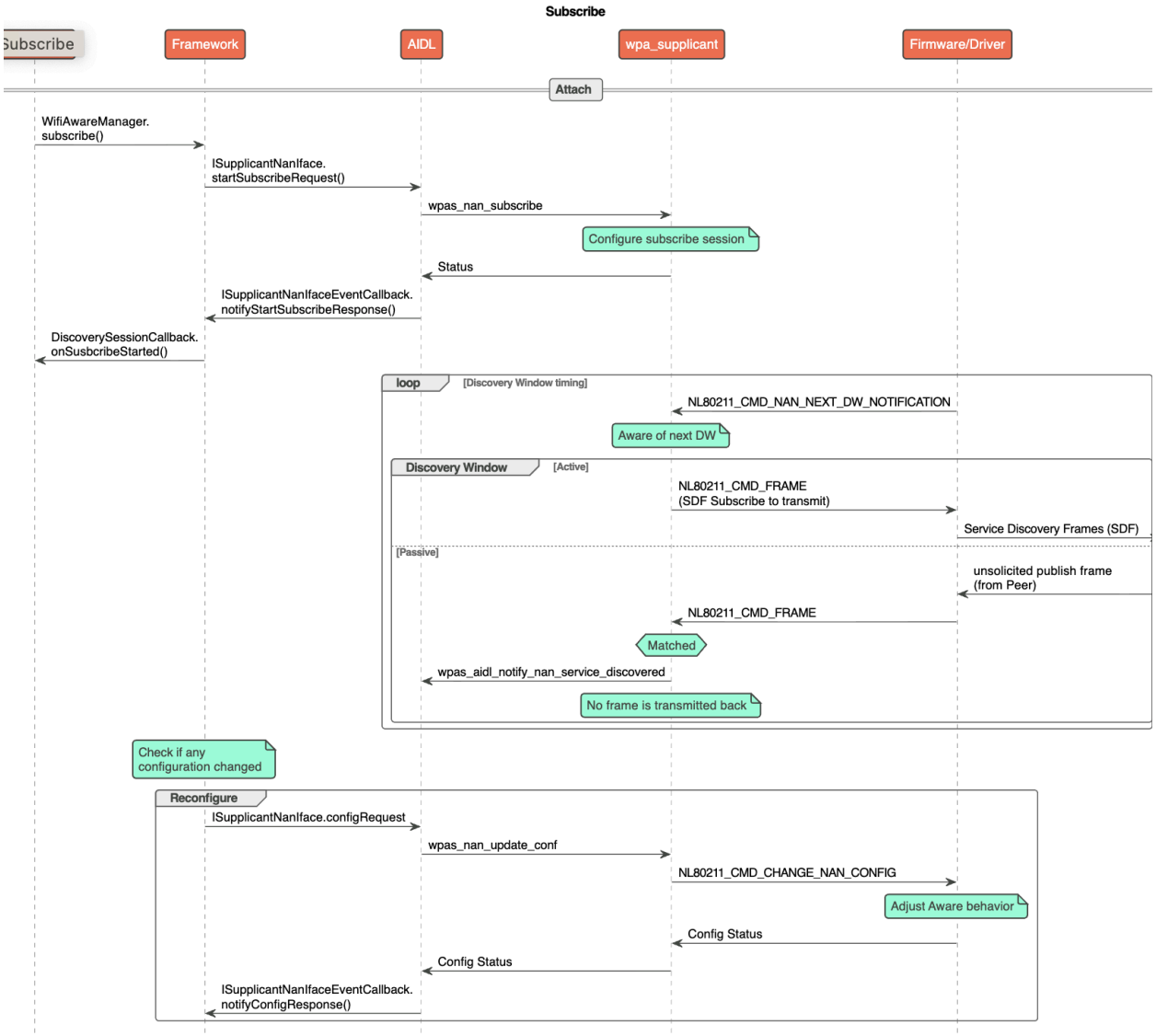

Figure 10. Wi-Fi Aware subscribe flow.

For more details on SDK API usage, see:

- Publish a service
- Subscribe to a service

Both publish and subscribe can be offloaded to firmware for periodic transmission of frames to reduce application processor wake up. Only required frames after a match need to be forwarded to the supplicant. For more information, see Service discovery.

AIDL

These are the AIDL interfaces for the publish-subscribe model for service discovery.

- ISupplicantNanIface.aidl
- ISupplicantNanIfaceEventCallback.aidl
- NanConfigRequest.aidl
- NanDataPathSecurityConfig.aidl
- NanDiscoveryCommonConfig.aidl
- NanPairingConfig.aidl
- NanPublishRequest.aidl
- NanSubscribeRequest.aidl

Service discovery

When all applicable conditions (service ID match, matching filter match, discovery range match, and service response filter criteria) are met for a received subscribe message, the publisher's trigger condition is met. This causes the publisher to transmit a solicited publish message back to the subscriber. Firmware also forwards the matched subscribe frames to the supplicant for further processing. Any subsequent matched frames, optionally, are forwarded to supplicant by the firmware.

The following diagram shows both publisher and subscriber's behavior if firmware supports offloading the discovery.

Note: Filters can be programmed in hardware, firmware, or `wpa_supplicant` depending on vendor software design.

Here is the call flow for service discovery (publisher's view, offloaded to firmware):

1. **App publishes:** The app requests to publish a service using `WifiAwareManager.publish()`.
2. **Framework request:** The Framework sends the publish configuration to the supplicant through the `ISupplicantNanIface.startPublishRequest()` AIDL call.
3. **Supplicant configures (offload):** `wpa_supplicant` (specifically `wpas_nan_publish`) receives the request. In this firmware-offloaded flow, the supplicant doesn't manage the

discovery table locally; instead, it sends the `NL80211_CMD_VENDOR` command to the driver or firmware to program the hardware's DE (Discovery Engine).

4. **Discovery window loop (autonomous):**
    - **Unsolicited path:** For unsolicited publishers, the firmware autonomously broadcasts publish frames during discovery windows. Unlike the userspace-driven flow, no per-frame `NL80211_CMD_FRAME` commands are sent from the supplicant.
    - **Solicited path (hardware matching):** When the firmware receives a peer's subscribe frame, it performs matching directly in hardware based on service ID, match filters, and the service response filter (SRF).
    - **Case 1 (no match):** If the incoming frame doesn't meet the response criteria, the firmware discards the frame silently without waking the host processor.
    - **Case 2 (match):** If a match is found, the firmware automatically generates and transmits a solicited publish response frame. The host CPU and `wpa_supplicant` remain uninvolved in this frame-exchange cycle, significantly reducing power consumption.

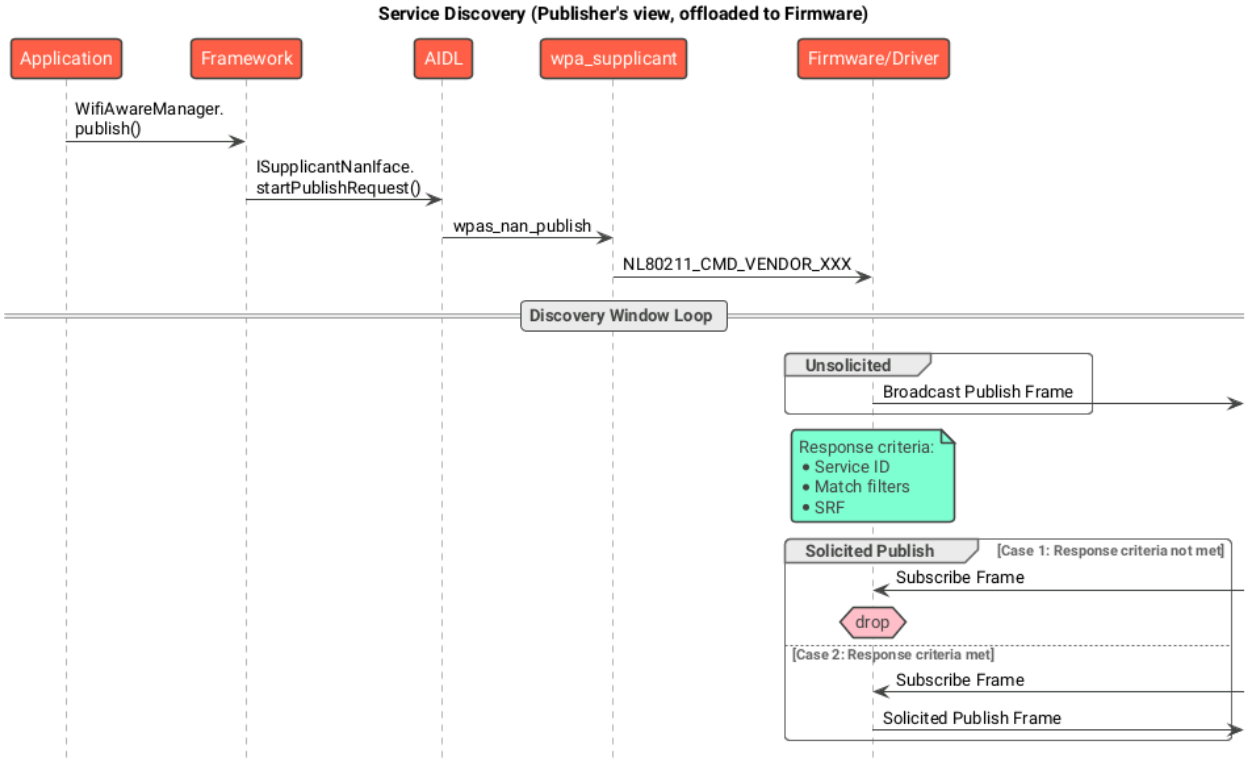

Figure 11. Wi-Fi Aware service discovery flow (publisher view, offloaded to firmware).

The call flow for service discovery (subscriber's view, offloaded to firmware):

1. **App subscribes:** The app requests to subscribe to a service using `WifiAwareManager.subscribe()`.
2. **Framework request:** The framework sends the subscribe configuration to the supplicant through the `ISupplicantNanIface.startSubscribeRequest()` AIDL call.
3. **Supplicant configures (offload):** In this firmware-offloaded flow, `wpa_supplicant` (through `wpas_nan_subscribe`) programs the hardware's discovery engine by sending the `NL80211_CMD_VENDOR` command. The supplicant delegates matching and frame management to the firmware to reduce host CPU overhead.
4. **Discovery window loop (autonomous):**
    - **Active path:** For active subscribers, the firmware autonomously generates and transmits Subscribe Service Discovery Frames (SDF) during discovery windows.
    - **Hardware matching:** The firmware monitors incoming publish frames from peers and evaluates them against the programmed service ID and match filters directly

in the radio hardware.

- **Discovery result:**
  - **Match failed:** If an incoming frame doesn't satisfy the match criteria, the firmware discards it immediately without notifying the host.
  - **Matched:** Upon the first successful match, the firmware passes the discovery frame up to `wpa_supplicant` through the `NL80211_CMD_FRAME` Netlink event.
  - **Filtering redundancy:** To conserve power, the firmware can optionally drop subsequent identical publish frames from the same peer once a match has already been reported to the host.

5. **Notification to app:** The supplicant processes the matched frame and triggers the `wpas_aidl_notify_nan_service_discovered` notification. The AIDL layer then calls `ISupplicantNanIfaceEventCallback.eventMatch()`, which the framework forwards to the app through `DiscoverySessionCallback.onServiceDiscovered()`, providing a `peerHandle` for future message exchanges.

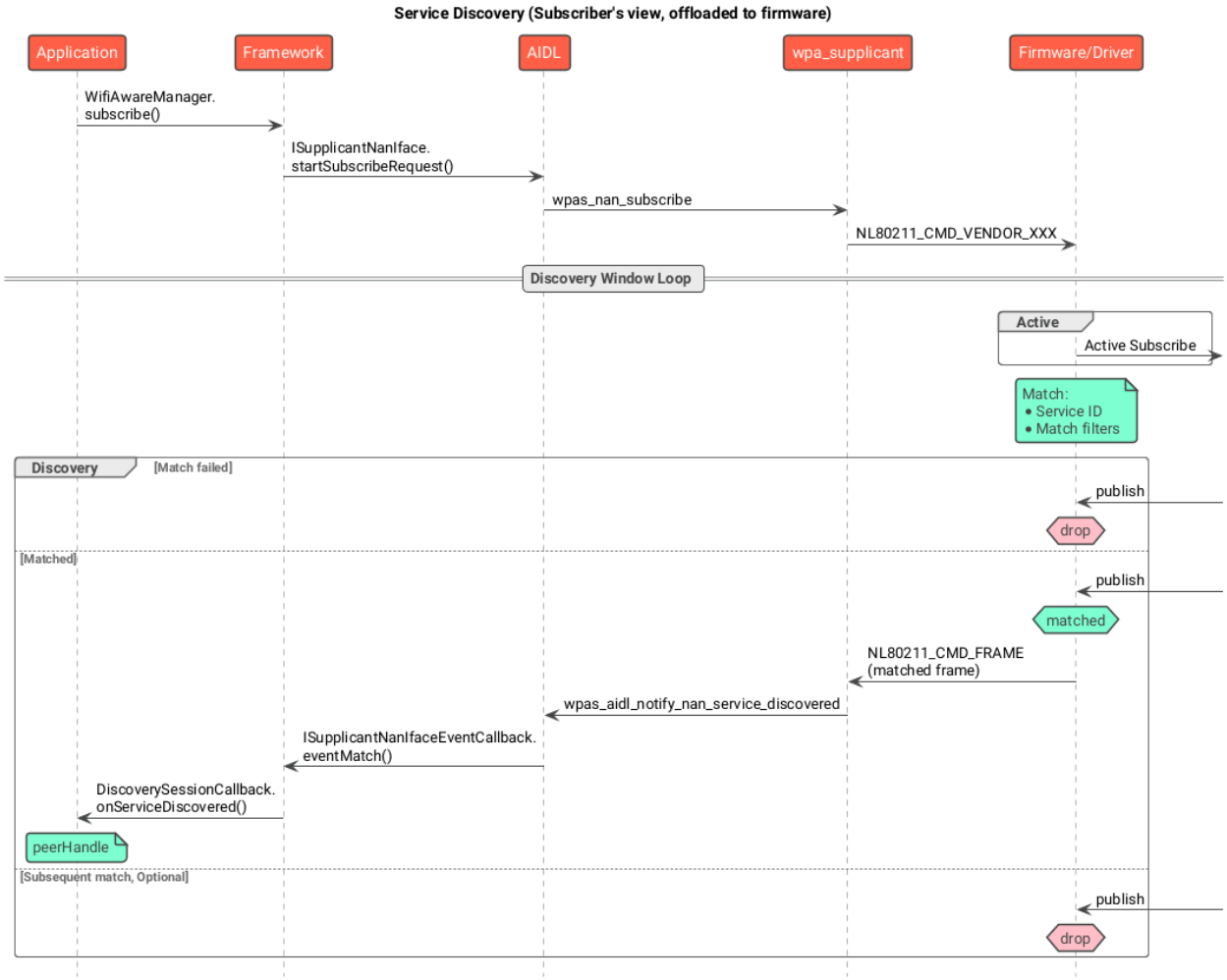

Figure 12. Wi-Fi Aware service discovery flow (subscriber view, offloaded to firmware).

For API usage details, see Subscribe to a service.

Supplicant-based discovery

The supplicant-based discovery approach is executed in `wpa_supplicant` depending on whether the firmware supports discovery offloading. Firmware capabilities are queried during Initialization. Matching criteria can be implemented in the supplicant, potentially reducing firmware processing. However, this trade-off increases power consumption due to frequent application processor wake-ups.

Here is the call flow for service discovery (publisher's view, supplicant based):

1. **App publishes:** The app requests to publish a service using `WifiAwareManager.publish()`.
2. **Framework request:** The framework sends the publish configuration to the supplicant through the `ISupplicantNanIface.startPublishRequest()` AIDL call.
3. **Supplicant configures:** `wpa_supplicant` (specifically `wpas_nan_publish`) processes the request locally. In this userspace-driven flow, the supplicant registers the service in its internal discovery table but doesn't send an immediate Netlink command to the firmware at this stage.
4. **Discovery window loop:**
    - **DW notification:** The firmware or driver periodically notifies `wpa_supplicant` of upcoming discovery window (DW) timings using `NL80211_CMD_NAN_NEXT_DW_NOTIFICATION`.
    - **Unsolicited path:** For unsolicited publishers, the supplicant builds the Publish Service Discovery Frame (SDF) and sends it to the firmware for transmission using `NL80211_CMD_FRAME`.
    - **Solicited path (userspace matching):** When the firmware receives a peer's subscribe frame (broadcast or unicast), it passes it up to the supplicant through `NL80211_CMD_FRAME` (RX).
    - **Matching logic:** The supplicant performs the matching logic in userspace, verifying the service ID and match filters. If a match is found, the supplicant generates a unicast publish response frame and sends it to the firmware through `NL80211_CMD_FRAME` (TX).
    - **Transmission confirmation:** The firmware transmits the solicited publish frame and notifies the supplicant of the completion through `NL80211_CMD_FRAME_TX_STATUS`.

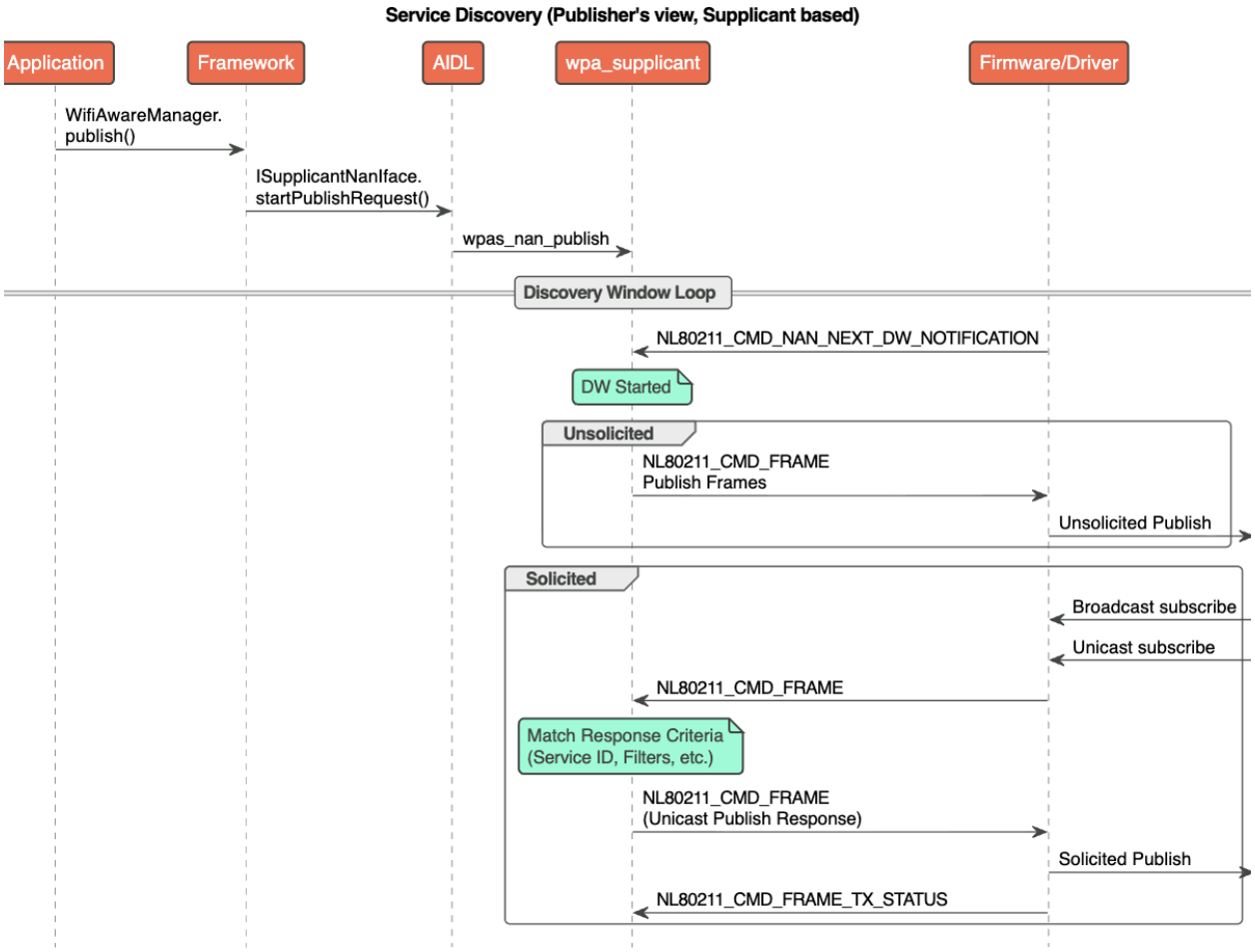

Figure 13. Wi-Fi Aware service discovery flow (publisher view, supplicant-based).

The call flow for service discovery (subscriber's view, supplicant-based):

1. **App subscribes:** The app requests to subscribe to a service using `WifiAwareManager.subscribe()`.
2. **Framework request:** The framework sends the subscribe configuration to the supplicant through the `ISupplicantNanIface.startSubscribeRequest()` AIDL call.
3. **Supplicant configures:** `wpa_supplicant` (specifically `wpas_nan_subscribe`) processes the request locally. In this userspace-driven flow, the supplicant registers the subscribe session in its internal discovery table and maintains the match criteria

(service ID and match filters).

4. **Discovery window loop:**
    - **DW notification:** The firmware or driver periodically notifies `wpa_supplicant` of upcoming discovery window (DW) timings using `NL80211_CMD_NAN_NEXT_DW_NOTIFICATION`.
    - **Active path:** For active subscribers, the supplicant builds a Subscribe Service Discovery Frame (SDF) and sends it to the firmware for transmission using `NL80211_CMD_FRAME`.
    - **Matching logic (userspace):** Unlike the offloaded flow, the firmware doesn't filter incoming frames. It passes all received peer publish frames to the supplicant through `NL80211_CMD_FRAME` (RX).
    - **Supplicant filtering:** The supplicant evaluates each incoming frame against its active sessions:
        - **Match failed:** If the frame does not satisfy the match criteria, the supplicant drops the frame.
        - **Matched:** If a match is found, the supplicant triggers the `wpas_aidl_notify_nan_service_discovered` notification to the AIDL layer.
        - **Redundancy management:** The supplicant can optionally drop subsequent identical publish frames from the same peer to prevent redundant notifications to the Framework.
5. **Notification to app:** The AIDL layer propagates the discovery through the `ISupplicantNanIfaceEventCallback.eventMatch()` callback. The framework then notifies the application through `DiscoverySessionCallback.onServiceDiscovered()`, providing a `peerHandle` for further communication.

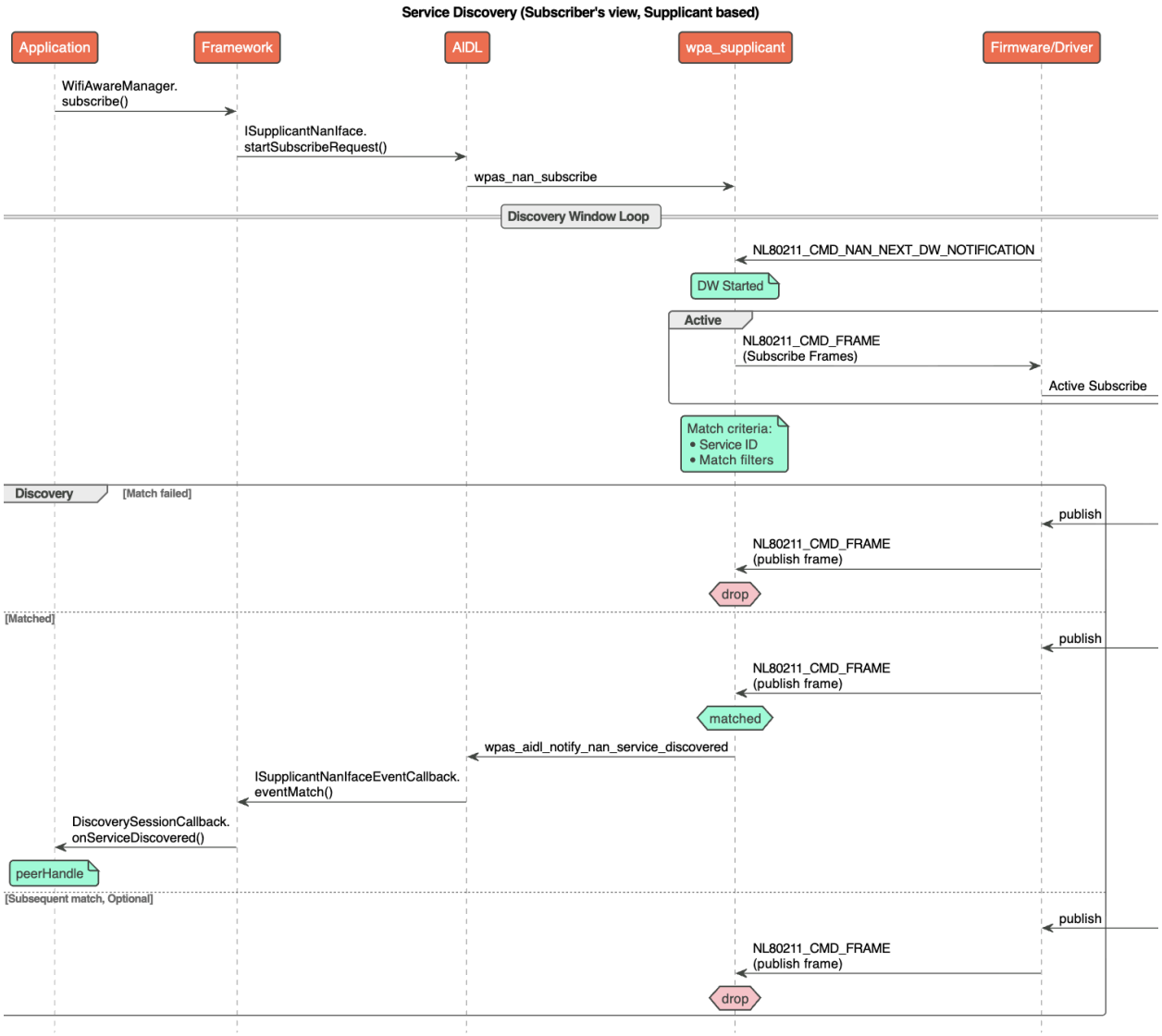

Figure 14. Wi-Fi Aware service discovery flow (subscriber view, supplicant-based).

Discovery expiry

Discovered peers maintain their service presence by periodically checking the publish messages from respective peers. If a peer stops sending these messages, its service is considered lost.

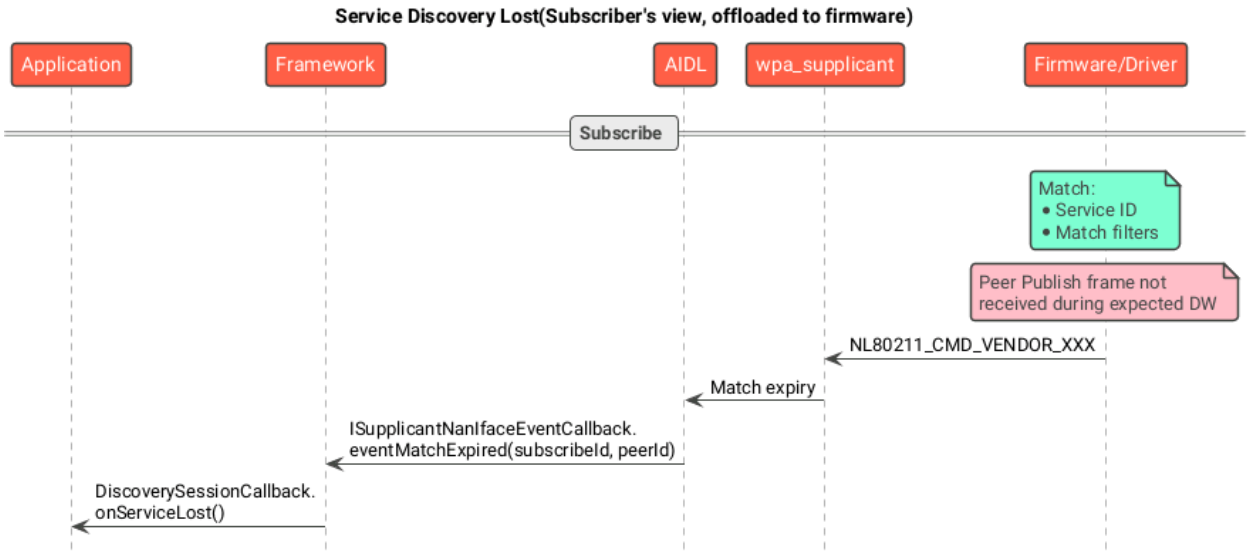

Figure 15. Wi-Fi Aware service discovery expiry flow.

AIDL

- ISupplicantNanIfaceEventCallback.aidl
- NanMatchInd.aidl

Follow-up message

Follow-up messages are a type of unicast action frame used in Wi-Fi Aware, primarily for further service discovery and related procedures like pairing and key exchange. The follow-up function provides a means for services and applications to transmit and receive service-specific information to and from a peer NAN device. They carry this detailed service information after the initial discovery of the device and service is complete. Follow-up messages can also be used for purposes like provisioning security keys without requiring a NAN Data Path setup, as well as for pairing bootstrapping and NIK exchange (see Pairing).

For API usage details, see Send a message .

When discovered, a device can send a follow-up message to a peer using `DiscoverySession.sendMessage()`. This allows the receiving app to obtain the sender's `peerHandle` for subsequent operations. The flow is handled as follows:

- **Initiating transmission:** The framework forwards the request to the supplicant through the `ISupplicantNanIface.transmitFollowupRequest()` AIDL call. The supplicant prepares a Follow-up Service Discovery Frame (SDF) and passes it to the firmware using the `NL80211_CMD_FRAME` Netlink command.
- **Transmission Status:** The supplicant receives the transmission status from the firmware through `NL80211_CMD_FRAME_TX_STATUS` and notifies the framework through the `notifyTransmitFollowupResponse()` callback, resulting in the sending application receiving either `onMessageSendSucceeded()` or `onMessageSendFailed()`.
- **Message reception:** The receiving device's firmware captures the frame during its Discovery Window and passes it to its local supplicant through `NL80211_CMD_FRAME`.
- **Reception notification:** The peer's supplicant processes the frame and triggers the `eventFollowupReceived()` AIDL callback. The framework then delivers the message payload and the sender's `peerHandle` to the target application through the `DiscoverySessionCallback.onMessageReceived()` callback.

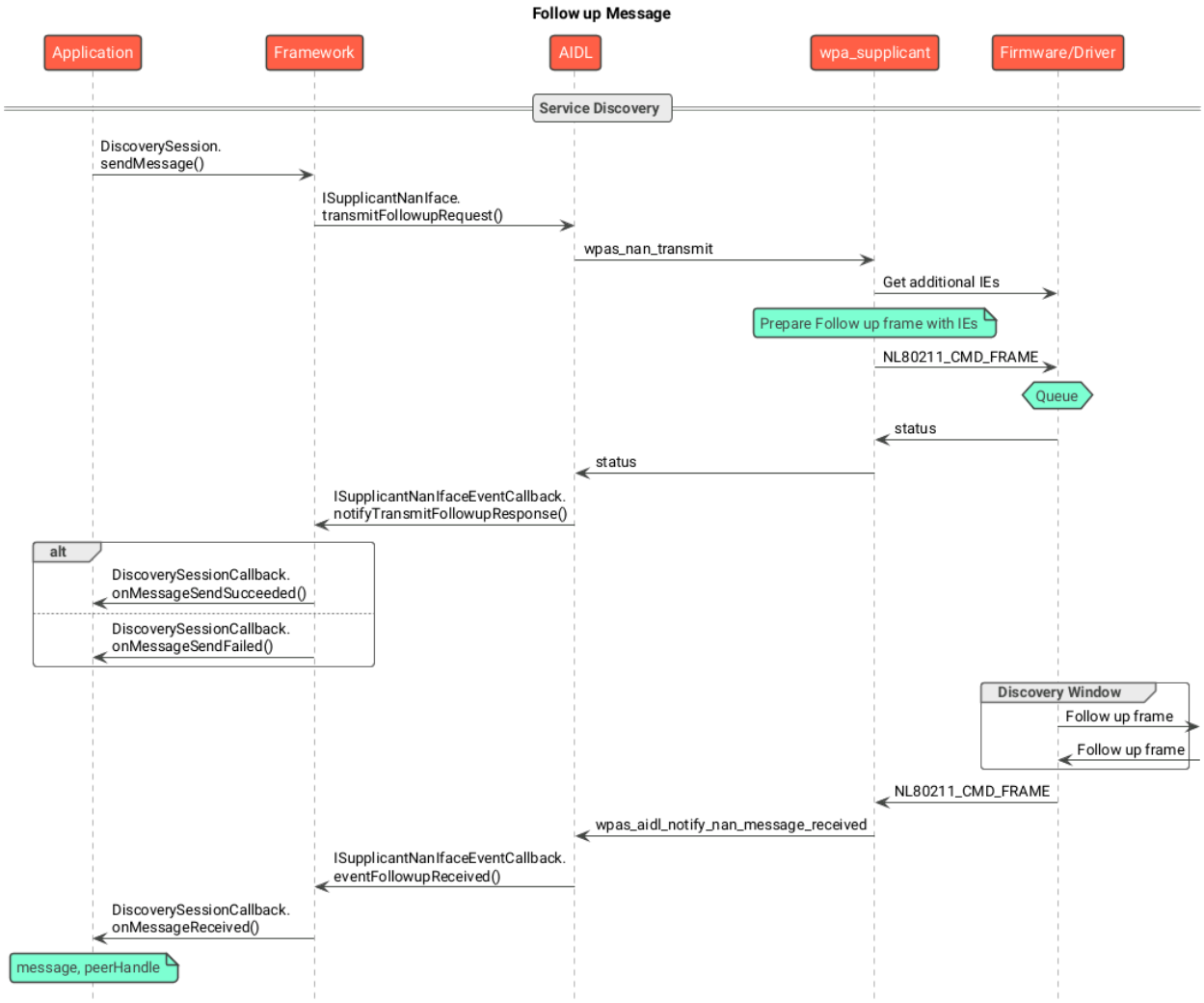

Figure 16. Wi-Fi Aware follow up message flow.

AIDL

- ISupplicantNanIface.aidl
- ISupplicantNanIfaceEventCallback.aidl
- NanFollowupReceivedInd.aidl
- NanTransmitFollowupRequest.aidl

Pairing

Wi-Fi Aware pairing, also referred to as NAN pairing, is a set of protocols and mechanisms that enable a pair of NAN devices to establish a trusted relationship, authenticate each other, and verify each other's identity.

The NAN pairing protocols include two main parts:

- NAN pairing setup protocol: Used to establish the initial trust and pairing relationship. Setup typically requires an Out-of-Band (OOB) bootstrapping mechanism to ensure both devices possess the same pairing credential.
- NAN pairing verification protocol: Used after initial setup to verify the identity of a paired peer, allowing for autonomous reconnection.

Bootstrapping

The NAN Pairing bootstrapping process allows two NAN devices to acquire a common pairing credential. This is a necessary step before performing a secure pairing setup, which then enables protection of subsequent NAN management frames and data path traffic.

In the example call flow provided, the service subscriber often acts as the bootstrapping initiator, and the service publisher acts as the bootstrapping responder.

The core of the pairing bootstrapping handshake is the exchange of follow-up messages between the bootstrapping initiator (typically the subscriber) and the bootstrapping responder (typically the publisher). This process establishes shared credentials (for example, a password) needed for security protocols like Password-based SAE or Opportunistic PASN during subsequent data path setup.

Publisher (responder) role:

1. **Configuration and advertising:** The application verifies support through `getCharacteristics().isAwarePairingSupported()` and configures `PublishConfig` to advertise supported methods (for example, PIN or QR code) using pairing-specific attributes sent to the firmware through `NL80211_CMD_FRAME`.
2. **Request handling:** Incoming requests are passed from firmware to supplicant through `NL80211_CMD_FRAME`, triggering the `eventBootstrappingRequest()` AIDL callback to the framework. The application then approves or rejects the attempt through `respondToBootstrappingIndicationRequest()`.
3. **Completion:** Once the handshake is confirmed, the framework notifies the application through the `DiscoverySessionCallback.onBootstrappingSucceeded()`

callback.

Subscriber (initiator) role:

1. **Discovery and initiation:** After verifying support and configuring `SubscribeConfig`, the application discovers a publisher advertising pairing support. The application selects a method and calls `initiateBootstrappingRequest()`.
2. **Handshake:** The supplicant builds a bootstrapping request carried within a follow-up frame and sends it through `NL80211_CMD_FRAME`. The firmware manages the over-the-air exchange and passes the response back to the supplicant through `NL80211_CMD_FRAME`.
3. **Completion:** The supplicant processes the confirmation and triggers the `eventBootstrappingConfirm()` AIDL callback. The framework then notifies the application through `onBootstrappingSucceeded()`.

This process can involve user interaction (such as entering a PIN or scanning a QR code) or other out-of-band means to acquire the common pairing credential.

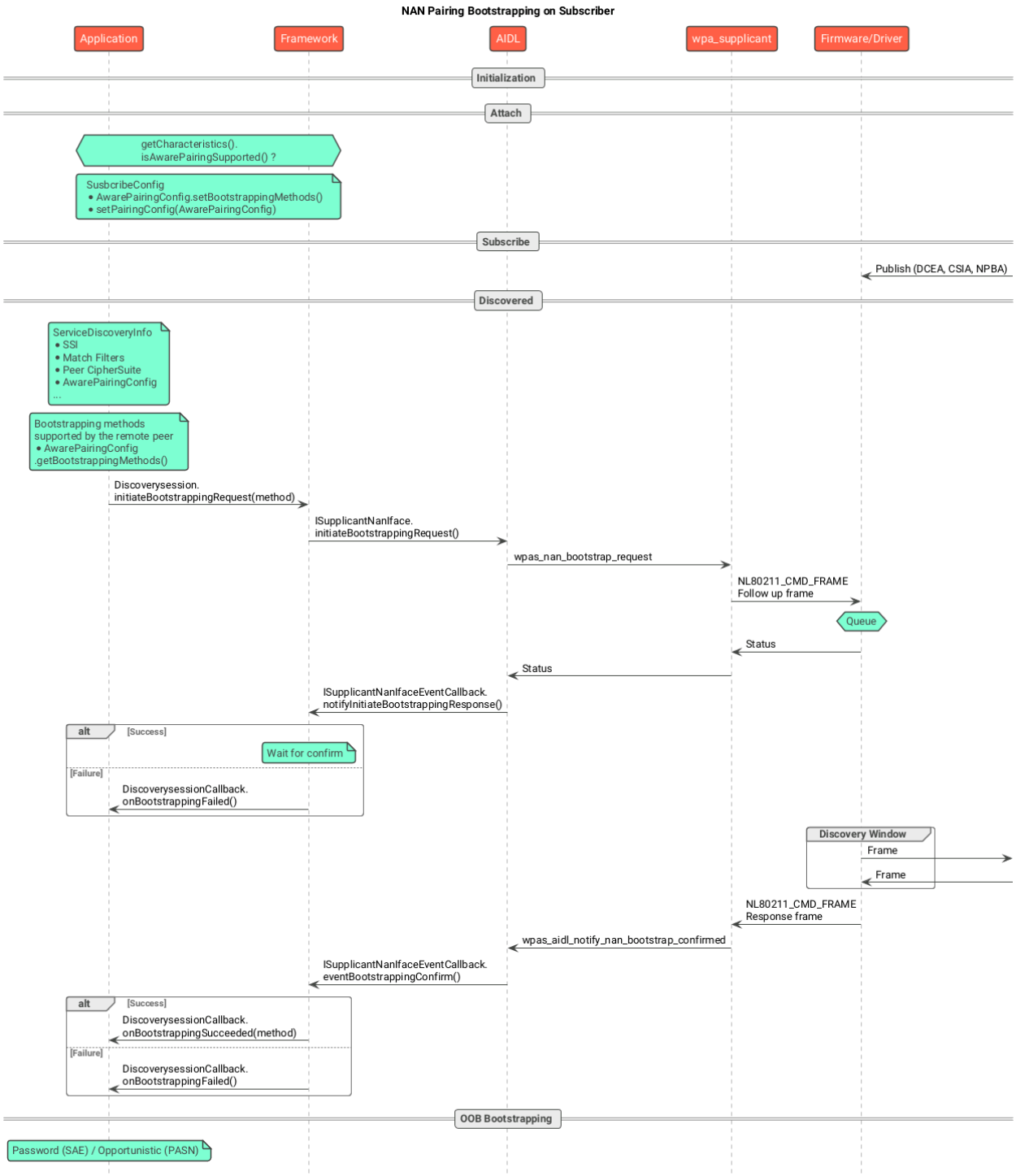

Figure 17. NAN pairing bootstrapping flow (subscriber).

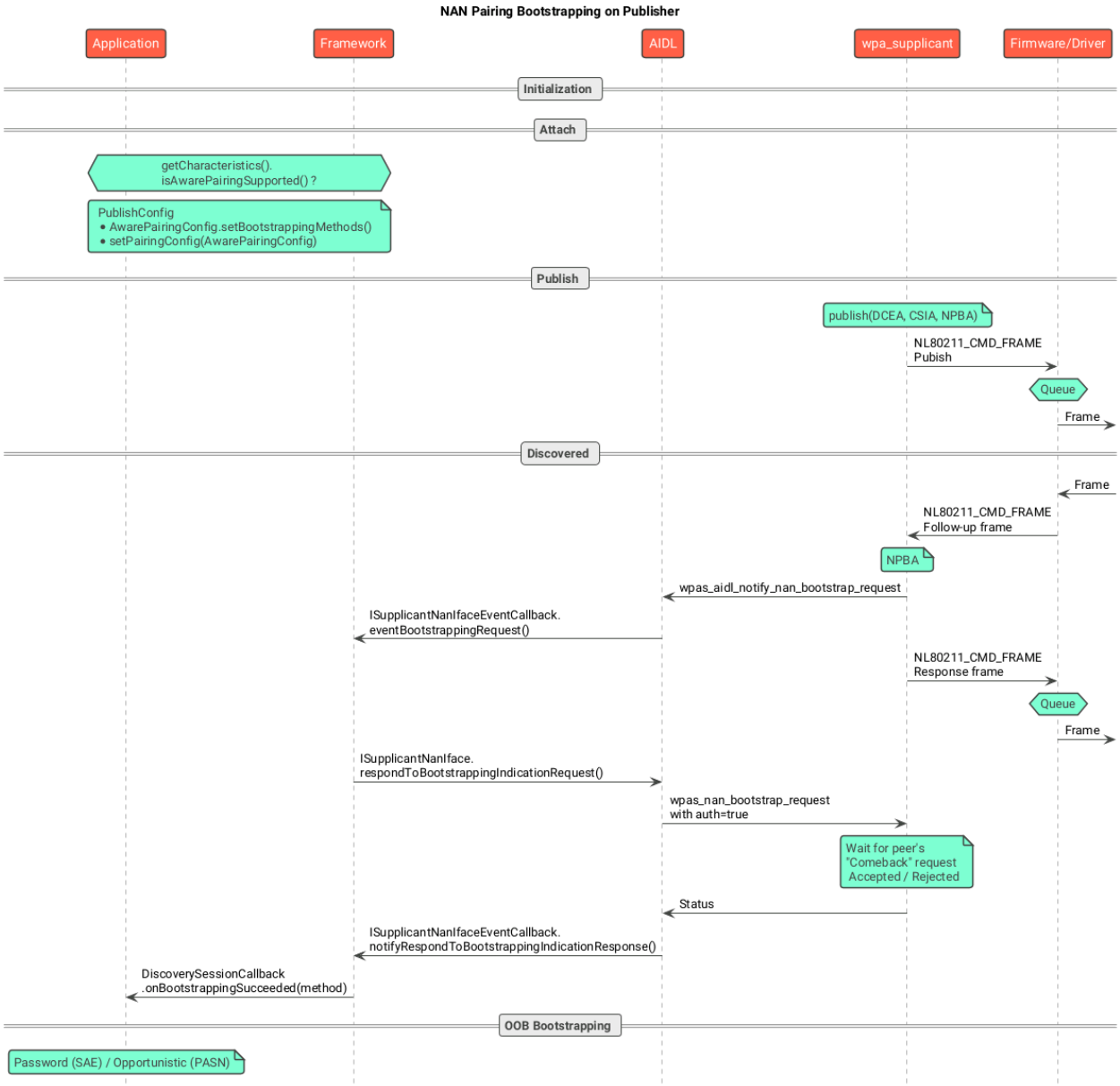

Figure 18. NAN pairing bootstrapping flow (publisher).

AIDL

- ISupplicantNanIface.aidl
- ISupplicantNanIfaceEventCallback.aidl

- NanBootstrappingRequest.aidl
- NanBootstrappingResponse.aidl
- NanBootstrappingConfirmInd.aidl
- NanBootstrappingMethod.aidl
- NanBootstrappingRequestInd.aidl

Pairing setup

The NAN pairing setup protocol enables a pair of NAN devices to authenticate and establish a secure communication relationship through Pre-Association Security Negotiation (PASN) when no prior pairing exists. This process derives management keys (NM-TK) to protect subsequent service discovery and follow-up message exchanges at the hardware level.

Subscriber (initiator) role:

1. **Initiation:** Following successful bootstrapping, the application calls `DiscoverySession.initiatePairingRequest()`. The framework generates a NAN Interface Key (NIK) and forwards the request to the supplicant through the `ISupplicantNanIface.initiatePairingRequest()` AIDL call.
2. **Handshake and key derivation:** The supplicant starts the PASN state machine and sends the PASN-M1 frame to the firmware through `NL80211_CMD_FRAME`. Upon receiving the PASN-M2 response, it establishes the NAN Pairing Key Security Association (NPKSA) and derives the management keys (NM-KCK, NM-TK, NM-KDK). It then sends the PASN-M3 frame to finalize authentication.
3. **Completion:** The supplicant triggers the `eventPairingConfirm()` AIDL callback, allowing the framework to notify the application through `onPairingSetupSucceeded()`.
4. **Key installation:** The supplicant installs the derived NM-TK into the firmware using the `NL80211_CMD_NEW_KEY` Netlink command to enable hardware-level encryption for Protected Unicast Management Frames.

Publisher (responder) role:

1. **Reception and approval:** The incoming PASN-M1 frame from the initiator is passed to the supplicant through `NL80211_CMD_FRAME (RX)`. This triggers the `onPairingSetupRequestReceived()` callback to the application. The application accepts the request by calling `acceptPairingRequest()`.
2. **Framework response:** The framework generates a NIK and issues the

`ISupplicantNanIface.respondToPairingIndicationRequest()` AIDL call to the supplicant.

3. **Handshake and derivation:** The supplicant establishes the NPKSA, derives the management keys, and sends the PASN-M2 response frame to the firmware through `NL80211_CMD_FRAME`.
4. **Completion:** The peer sends the final PASN-M3 frame. The supplicant validates the handshake and triggers the `eventPairingConfirm()` callback. The framework notifies the application of success through `onPairingSetupSucceeded()`.
5. **Key installation:** Similar to the initiator, the supplicant installs the derived NM-TK into the firmware using the `NL80211_CMD_NEW_KEY` command to secure the link for Protected Unicast Management Frames.

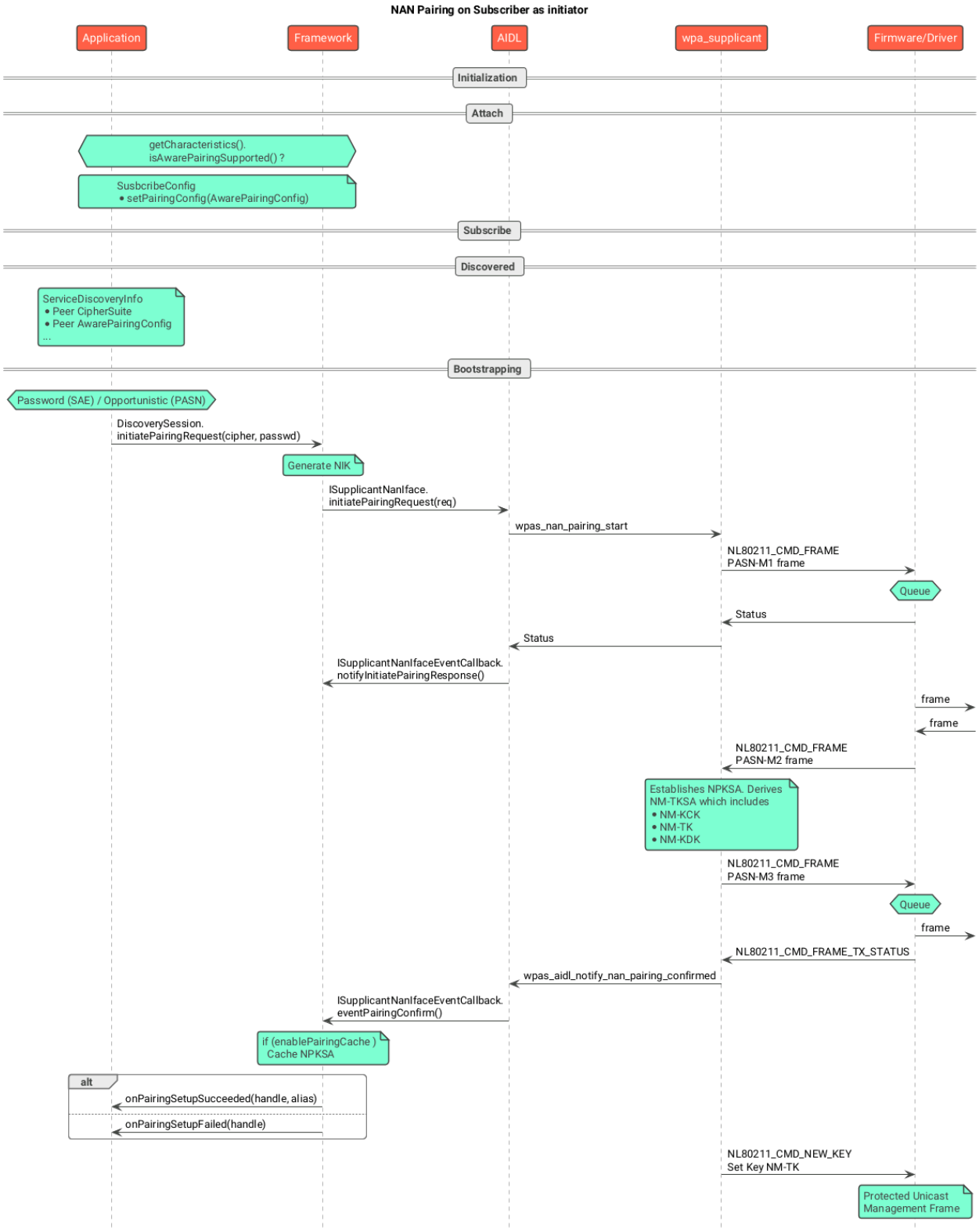

Figure 19. NAN pairing on subscriber as initiator flow.

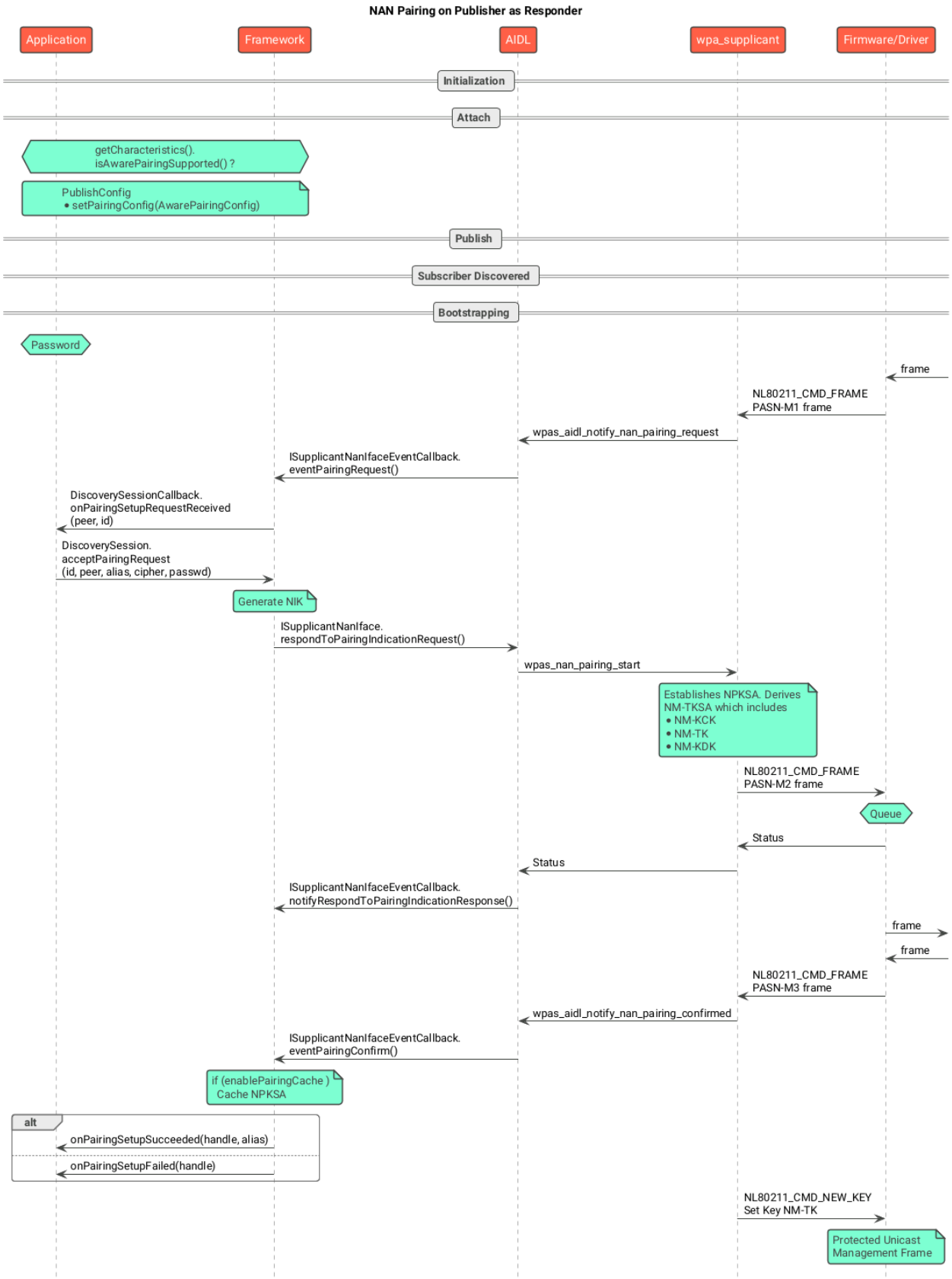

Figure 20. NAN pairing on publisher as responder flow.

When NIK or NPK caching is enabled, the exchange of NIK between paired devices uses protected follow-up messages, and the Key Data field of the NAN Shared Key Descriptor attribute in these messages is encrypted by the NM-KEK (A key encryption key for protecting the Key Data field of the NAN Shared Key Descriptor attribute included in the SDFs). The same mechanism can be used to exchange other keys like BIGTK and IGTK.

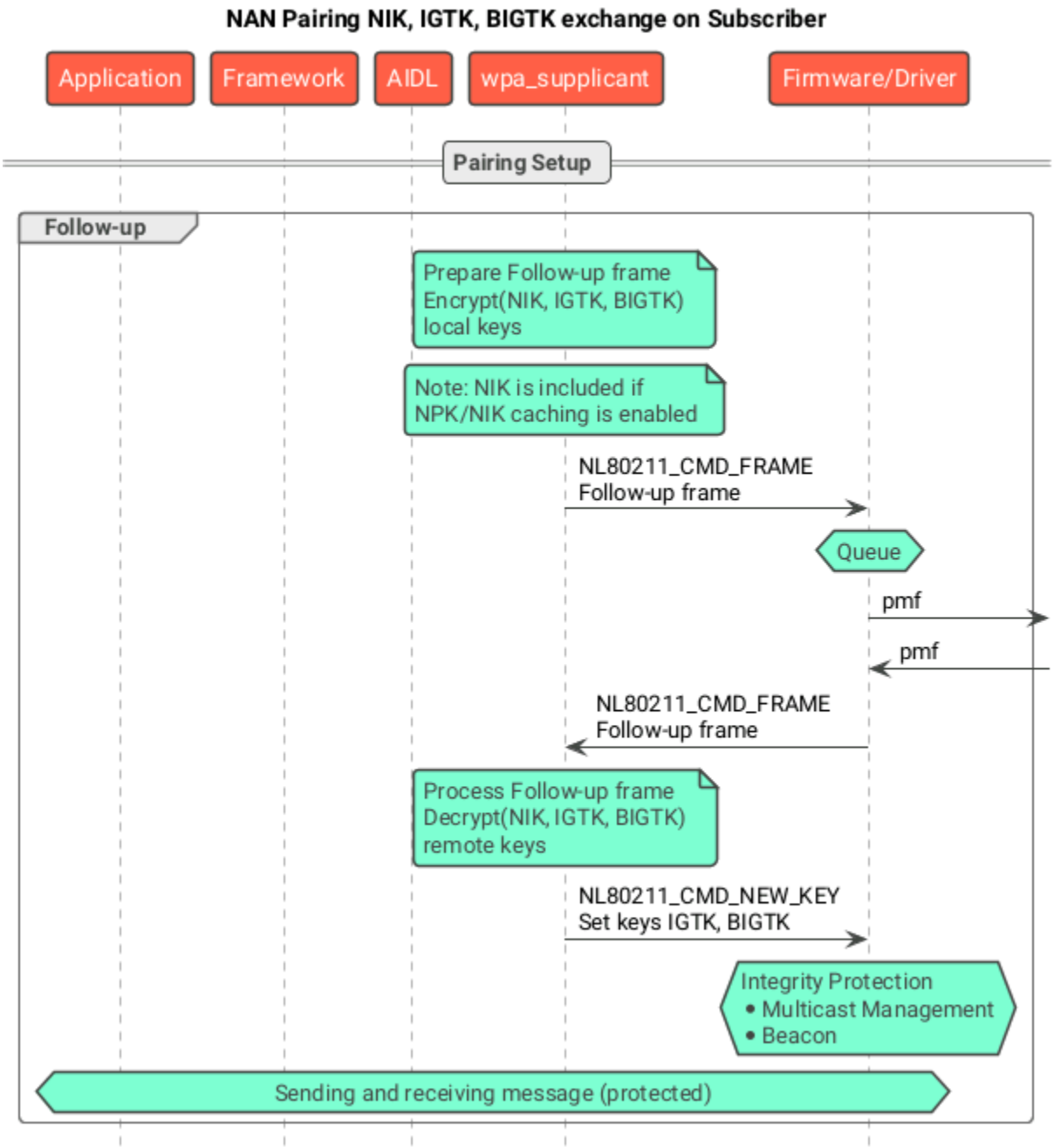

Figure 21. NAN pairing NIK, IGTK, BIGTK exchange on subscriber flow.

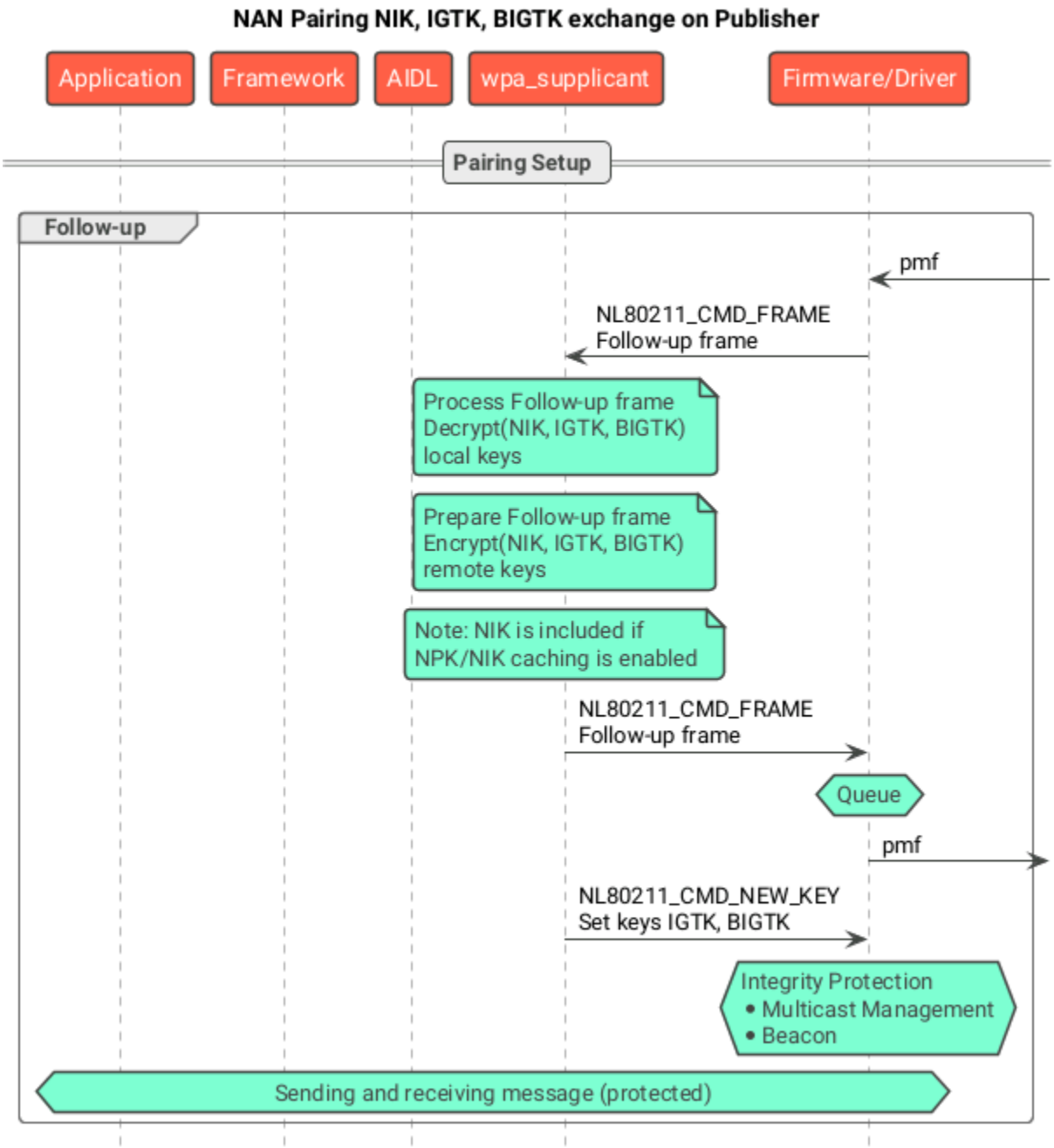

Figure 22. NAN pairing NIK, IGTK, BIGTK exchange on publisher flow.

All the following subsequent management stages are secured.

- Further service discovery
- Ranging Setup
- Data path setup

AIDL

The following resources constitute the core AIDL interfaces required for the pairing under the Wi-Fi Aware architecture:

- ISupplicantNanIface.aidl
- ISupplicantNanIfaceEventCallback.aidl
- NanIdentityResolutionAttribute.aidl
- NanPairingAkm.aidl
- NanPairingConfig.aidl
- NanPairingConfirmInd.aidl
- NanPairingRequest.aidl
- NanPairingRequestInd.aidl
- NanPairingRequestType.aidl
- NanPairingSecurityConfig.aidl
- NanRespondToPairingIndicationRequest .aidl

Pairing verification

The NAN pairing verification protocol enables a pair of previously paired NAN devices to mutually authenticate and re-establish a secure communication relationship without repeating the initial bootstrapping or setup phases. This mechanism uses a cached NAN Pairing Key Security Association (NPKSA) to perform an expedited PASN handshake and derive fresh management keys (NM-TK) for link protection.

Subscriber (initiator) role:

1. **Trigger and initiation:** Upon discovery of a previously paired publisher, the framework validates the local NPKSA cache. If a valid association exists, it initiates verification by issuing the `ISupplicantNanIface.initiatePairingRequest()` AIDL call to the supplicant.
2. **Handshake and re-derivation:** The supplicant dispatches a PASN-M1 frame to the firmware through `NL80211_CMD_FRAME`. Upon receipt of the PASN-M2 response, it

verifies key material, re-establishes the NPKSA, and derives fresh management keys (NM-KCK, NM-TK, and NM-KDK) before sending the final PASN-M3 frame.
3. **Completion and key installation:** The supplicant triggers the `eventPairingConfirm()` AIDL callback, allowing the framework to notify the application through `onPairingVerificationSucceeded()`. The fresh NM-TK is subsequently installed into the firmware through the `NL80211_CMD_NEW_KEY` command.

Publisher (responder) role:

1. **Reception and cache check:** The initiator's PASN-M1 frame is passed to the supplicant through `NL80211_CMD_FRAME (RX)`, triggering the `eventPairingRequest()` callback. The framework identifies the verification attempt and validates the NPKSA cache for the corresponding peer.
2. **Response:** Following authorization, the framework issues the `ISupplicantNanIface.respondToPairingIndicationRequest()` call. The supplicant re-establishes the association, derives fresh management keys, and sends the PASN-M2 response frame through `NL80211_CMD_FRAME`.
3. **Completion and key installation:** Upon receiving the final PASN-M3 frame, the supplicant confirms verification through the `eventPairingConfirm()` callback. The framework notifies the application through `onPairingVerificationSucceeded()` and installs the derived NM-TK using the `NL80211_CMD_NEW_KEY` Netlink command.

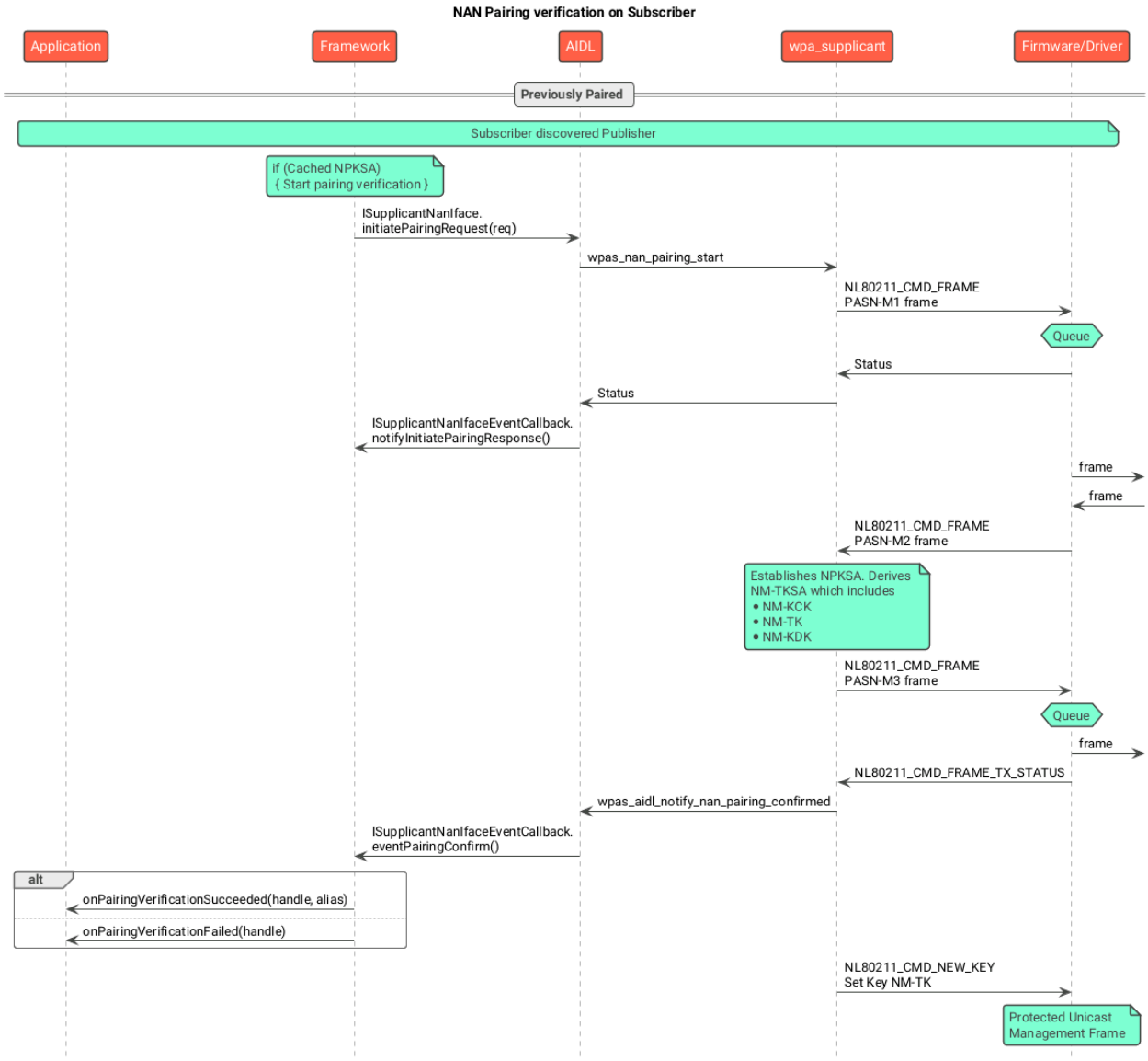

Figure 23. NAN pairing verification on subscriber flow.

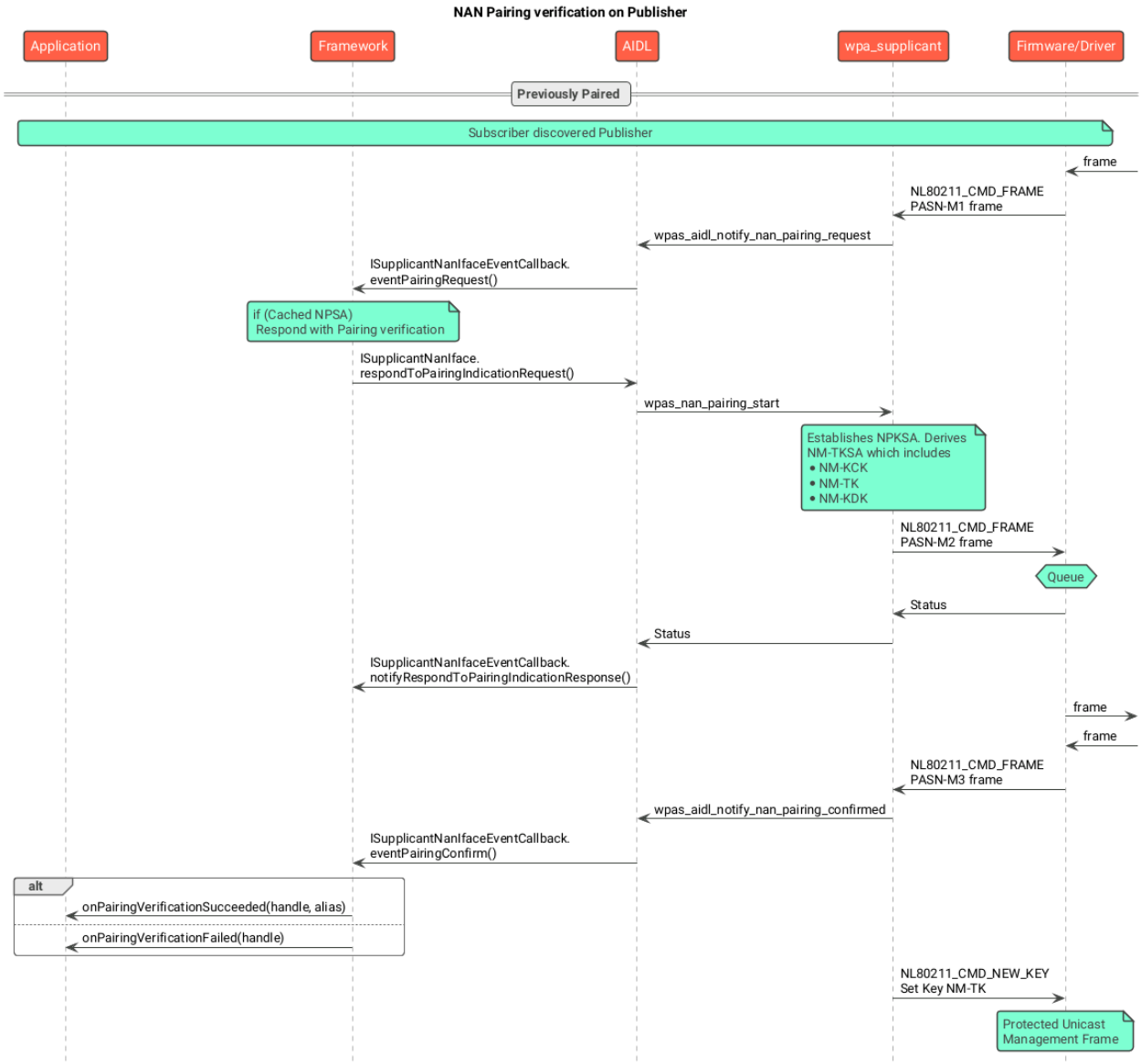

Figure 24. NAN pairing verification on publisher flow.

AIDL

See AIDL section in Pairing setup.

Data path

For Android API usage from an application perspective, see Create a connection.

In Wi-Fi Aware, support for pairing is a mandatory feature for enhanced security. However, in scenarios where a peer device does not support pairing (due to legacy implementation or operating system differences), a non-paired connection is established. This introduces a key constraint: a single NAN Data Interface (NDI) cannot simultaneously manage both paired and non-paired connections according to the Wi-Fi Aware specification.

Consequently, if a device supports only one NDI, the security level of the first connection established dictates the type for all subsequent connections on that interface. To address this limitation and support both security contexts (secure and open) concurrently, a device can implement two NDIs, dedicating one exclusively for paired connections and the other for non-paired connections.

Data path interface selection is documented in `WifiAwareDataPathStateManager.selectInterfaceForRequest()` (see `WifiAwareDataPathStateManager.java`).

- **Re-use for same peer (if security allows):** If the peer is already connected on an interface and a device setting permits it (based on the device overlay `config_wifiAwareNdpSecurityUpdateOnSameNdi`), that interface is chosen.
- **Unused interface:** If not, the framework looks for a completely free interface.
- **Re-use for different peer (if device allows):** If no free interfaces exist and a device setting permits (`config_wifiAllowMultipleNetworksOnSameAwareNdi`), it tries to use an interface already active with a different peer.
- If none of these conditions can be met, it returns no interface (`null`), indicating the request cannot be fulfilled with current resources.

Two device overlay configurations control this behavior:

- `config_wifiAwareNdpSecurityUpdateOnSameNdi`: Controls whether an existing NDI can be reused for a new NDP to the same peer if a security upgrade is being performed.
- `config_wifiAllowMultipleNetworksOnSameAwareNdi`: Controls whether an NDI can host multiple NDPs simultaneously (to different peers, or to the same peer if the security upgrade rule allows it).

Following interface selection, establishing a secure NAN Data Path (NDP) involves a handshake and cryptographic key derivation sequence:

Secure NDP as initiator:

- **Signaling and request:** The application initiates a network request through `ConnectivityManager.requestNetwork()`. The framework proposes a NAN Data Link (NDL) schedule and issues the `ISupplicantNanIface.initiateDataPathRequest()` AIDL call to the supplicant.
- **Security foundation:** The supplicant derives the ND-PMK (Data Path Pairwise Master Key) from the established NM-KDK, ensuring data path security is cryptographically bound to the initial pairing association.
- **Handshake execution (M1-M4):**
  - **M1/M2:** The supplicant dispatches a Data Path Request (M1) to the firmware through `NL80211_CMD_FRAME`. The peer provides a Data Path Response (M2) in return.
  - **M3/M4:** The supplicant sends a Data Path Confirm (M3) and derives the temporal ND-TK. The responder completes the exchange with a Data Path Install (M4) frame containing group keys (GTK, IGTK, BIGTK).
- **Key installation:** The supplicant installs the unicast ND-TK and multicast GTK into the firmware using `NL80211_CMD_NEW_KEY` for hardware-level encryption (including IGTK/BIGTK if required).
- **Network connectivity:** Upon the `eventDataPathConfirm()` callback, the framework finalizes the NDL schedule and notifies the application through `NetworkCallback.onAvailable()` with the peer's IPv6 details.

Secure NDP as responder:

- **Signaling and readiness:** The responder application prepares a `ServerSocket` and initiates a network request through `ConnectivityManager.requestNetwork()`.
- **Request reception (M1):** The firmware captures the incoming Data Path Request (M1) and forwards it to the supplicant through the `NL80211_CMD_FRAME` Netlink event.
- **Notification and decision:** The supplicant triggers the `eventDataPathRequest()` AIDL callback. The framework evaluates the NDL schedule and authorizes the session through `respondToDataPathIndicationRequest()`.
- **Security derivation:** The supplicant derives the ND-PMK from the NM-KDK and transmits the Data Path Response (M2) frame using the `NL80211_CMD_FRAME` command.
- **Handshake completion (M3-M4):** Following receipt of the initiator's Data Path Confirm

(M3), the supplicant prepares and sends the Data Path Install (M4) frame containing local group keys.

- **Key installation:** Similar to the initiator, the supplicant installs derived keys (for example; ND-TK, GTK.) into the firmware using the `NL80211_CMD_NEW_KEY` Netlink command.
- **Network available:** The supplicant confirms success through `eventDataPathConfirm()`, allowing the framework to finalize the link and alert the application through `NetworkCallback.onAvailable()`.

The framework participates in NDL schedule negotiation (see NDL schedule).

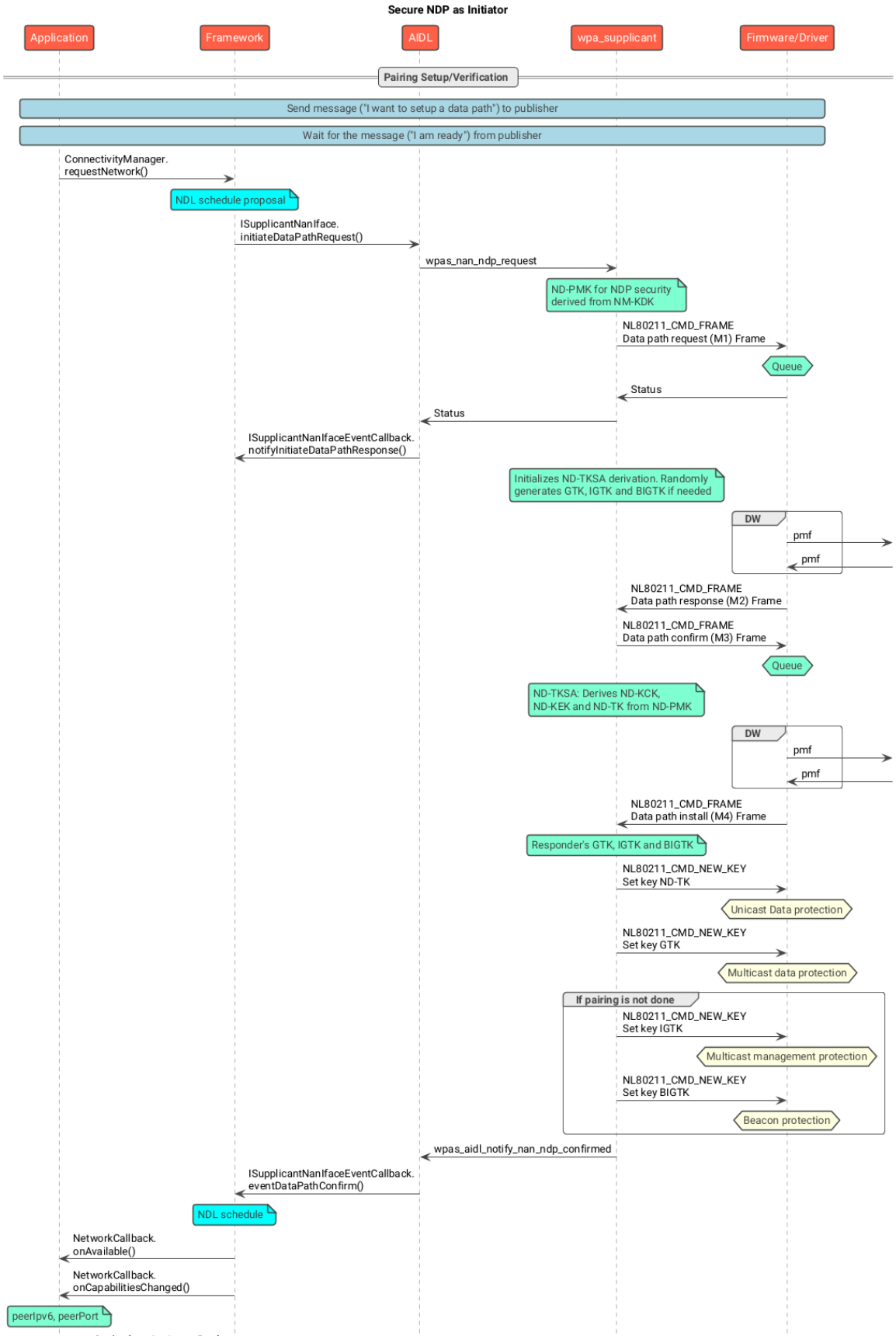
Secure NDP as Initiator
Application
Framework
AIDL
wpa_supplicant
Firmware/Driver
Pairing Setup/Verification
Send message ("I want to setup a data path") to publisher
Wait for the message ("I am ready") from publisher
ConnectivityManager.
requestNetwork()
NDL schedule proposal
ISupplicantNanIface.
initiateDataPathRequest()
wpas_nan_ndp_request
ND-PMK for NDP security
derived from NM-KDK
NL80211_CMD_FRAME
Data path request (M1) Frame
Queue
Status
Status
ISupplicantNanIfaceEventCallback.
notifyInitiateDataPathResponse()
Initializes ND-TKSA derivation. Randomly
generates GTK, IGTK and BIGTK if needed
DW
pmf
pmf
NL80211_CMD_FRAME
Data path response (M2) Frame
NL80211_CMD_FRAME
Data path confirm (M3) Frame
Queue
ND-TKSA: Derives ND-KCK,
ND-KEK and ND-TK from ND-PMK
DW
pmf
pmf
NL80211_CMD_FRAME
Data path install (M4) Frame
Responder's GTK, IGTK and BIGTK
NL80211_CMD_NEW_KEY
Set key ND-TK
Unicast Data protection
NL80211_CMD_NEW_KEY
Set key GTK
Multicast data protection
If pairing is not done
NL80211_CMD_NEW_KEY
Set key IGTK
Multicast management protection
NL80211_CMD_NEW_KEY
Set key BIGTK
Beacon protection
wpas_aidl_notify_nan_ndp_confirmed
ISupplicantNanIfaceEventCallback.
eventDataPathConfirm()
NDL schedule
NetworkCallback.
onAvailable()
NetworkCallback.
onCapabilitiesChanged()
peerIpv6, peerPort

Figure 25. Secure NDP as initiator flow.

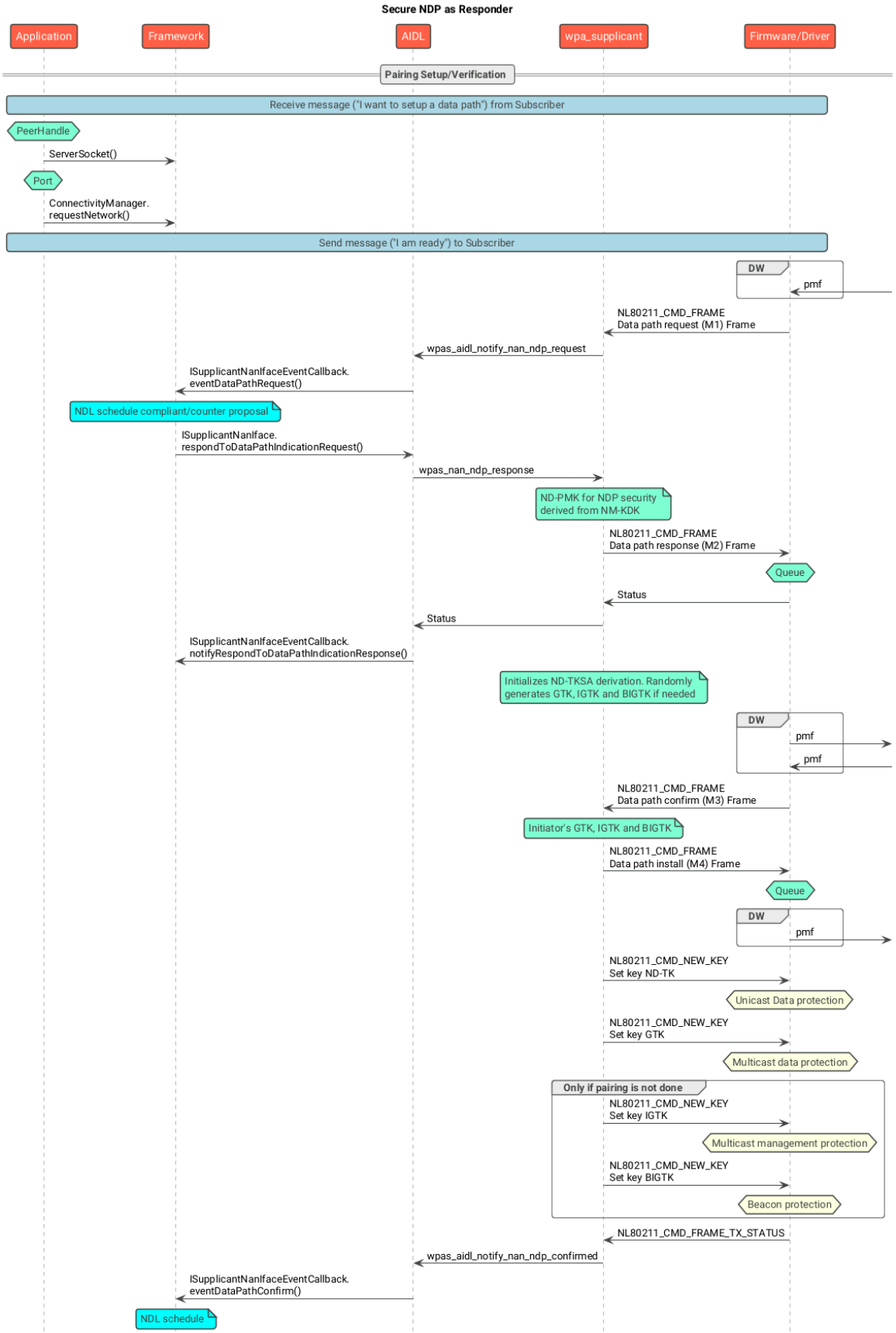
Secure NDP as Responder
Application
Framework
AIDL
wpa_supplicant
Firmware/Driver
Pairing Setup/Verification
Receive message ("I want to setup a data path") from Subscriber
PeerHandle
ServerSocket()
Port
ConnectivityManager.
requestNetwork()
Send message ("I am ready") to Subscriber
DW
pmf
NL80211_CMD_FRAME
Data path request (M1) Frame
wpas_aidl_notify_nan_ndp_request
ISupplicantNanIfaceEventCallback.
eventDataPathRequest()
NDL schedule compliant/counter proposal
ISupplicantNanIface.
respondToDataPathIndicationRequest()
wpas_nan_ndp_response
ND-PMK for NDP security
derived from NM-KDK
NL80211_CMD_FRAME
Data path response (M2) Frame
Queue
Status
Status
ISupplicantNanIfaceEventCallback.
notifyRespondToDataPathIndicationResponse()
Initializes ND-TKSA derivation. Randomly
generates GTK, IGTK and BIGTK if needed
DW
pmf
pmf
NL80211_CMD_FRAME
Data path confirm (M3) Frame
Initiator's GTK, IGTK and BIGTK
NL80211_CMD_FRAME
Data path install (M4) Frame
Queue
DW
pmf
NL80211_CMD_NEW_KEY
Set key ND-TK
Unicast Data protection
NL80211_CMD_NEW_KEY
Set key GTK
Multicast data protection
Only if pairing is not done
NL80211_CMD_NEW_KEY
Set key IGTK
Multicast management protection
NL80211_CMD_NEW_KEY
Set key BIGTK
Beacon protection
NL80211_CMD_FRAME_TX_STATUS
wpas_aidl_notify_nan_ndp_confirmed
ISupplicantNanIfaceEventCallback.
eventDataPathConfirm()
NDL schedule

Figure 26. Secure NDP as responder flow.

AIDL

- ISupplicantNanIface.aidl
- ISupplicantNanIfaceEventCallback.aidl
- NanCipherSuiteType.aidl
- NanDataPathChannelInfo.aidl
- NanDataPathConfirmInd.aidl
- NanDataPathRequestInd.aidl
- NanDataPathSecurityConfig.aidl
- NanInitiateDataPathRequest.aidl
- NanSchedule.aidl
- ScheduleBitmap.aidl

MAC address randomization

Wi-Fi Aware requires that MAC addresses used for operations (NMI and NDPs) be randomized, rather than using the device's permanent MAC address.

Randomization schedule:

- Randomization occurs whenever Wi-Fi Aware is enabled or re-enabled.
- A regular interval for re-randomization is configured with a default of 30 minutes, as defined by the framework. (According to the Wi-Fi Aware specification, the recommended change interval is 15 minutes.)

This randomization helps protect user privacy by making it harder to track a device based on its MAC address across Wi-Fi networks.

Constraints on changing MAC addresses:

- Management Interface and Data Interface addresses (NMI and NDI) shall not change for the lifetime of an NDP. This applies once a device sets up an NDP and selects an NDI for it.
- Once a NAN device changes its NAN Interface Address (referring to the NMI, which is part of the Master Rank), it shall not change the NAN Interface Address again until after 240 Discovery Windows (DWs). This provides a minimum stability period tied to NAN cluster synchronization.

- A NAN device shall not change its NMI address when it has active NM-TKSAs or ND-TKSAs.

The NMI is a component of the NAN Master Rank calculation. Changing the NMI affects the Master Rank and can trigger the Anchor Master selection algorithm. The NMI is also used in deriving keys during opportunistic pairing, and the NM-TK is associated with NMI addresses.

When setting up an NDP, a device selects an NDI for that specific NDP. This NDI is used as the transmitter address for data frames associated with that NDP. A device may use different NDIs for different NDPs based on security requirements. The ND-TK is associated with NDI addresses used in a connection.

In the Android 16 Wi-Fi Aware implementation, MAC address randomization is managed at the chip driver or firmware level.

For more information, see the following resources:

- Android 13 Compatibility Definition | Wi-Fi Aware
- Wi-Fi Aware | Android Open Source Project
- MAC randomization behavior | Android Open Source Project

In Android 17 and higher, the framework controls MAC address randomization for both NMI and NDI/NDP, setting them through the supplicant and firmware.

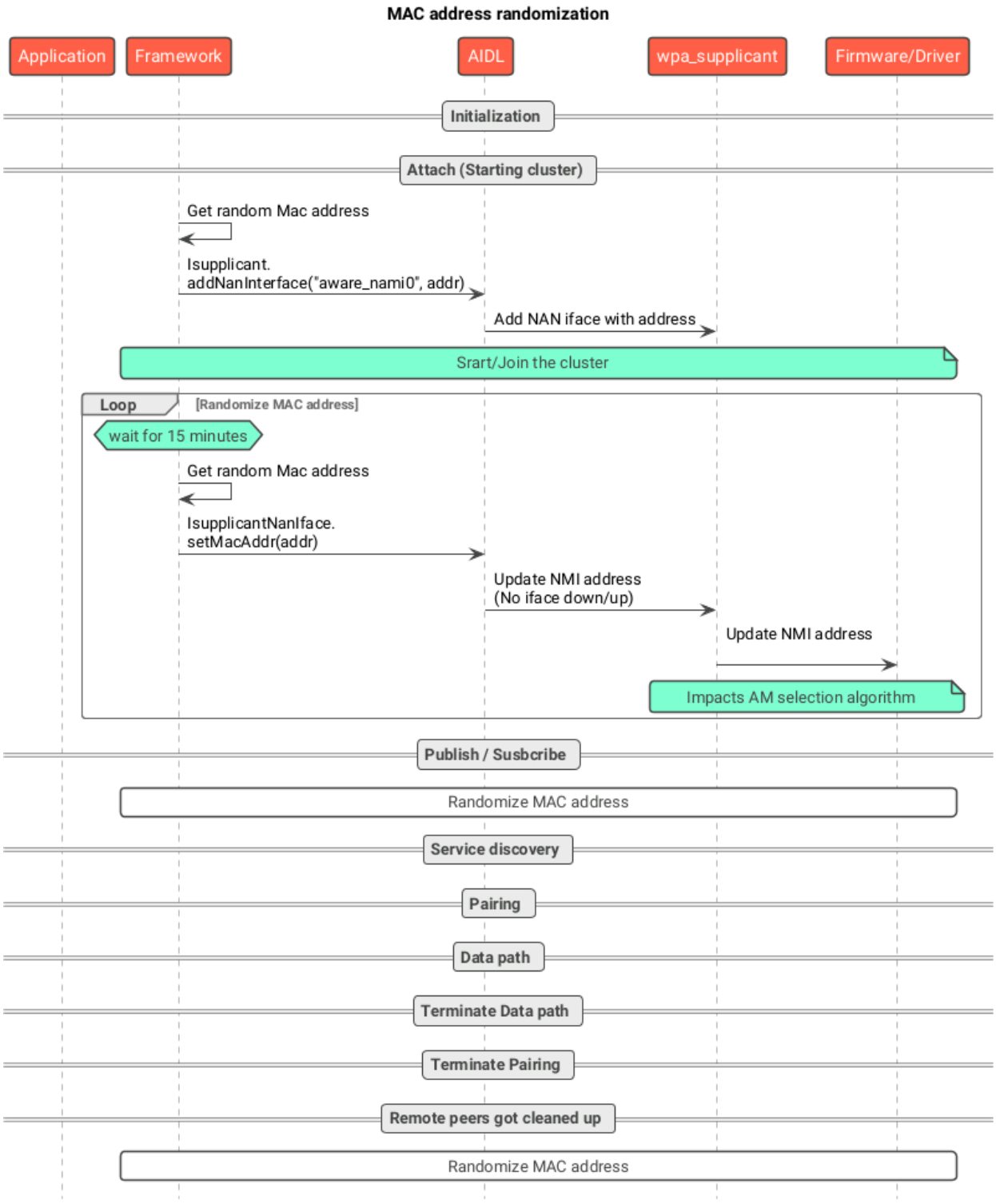
MAC address randomization
Application
Framework
AIDL
wpa_supplicant
Firmware/Driver
Initialization
Attach (Starting cluster)
Get random Mac address
Isupplicant.
addNanInterface("aware_nami0", addr)
Add NAN iface with address
Srart/Join the cluster
Loop
[Randomize MAC address]
wait for 15 minutes
Get random Mac address
IsupplicantNanIface.
setMacAddr(addr)
Update NMI address
(No iface down/up)
Update NMI address
Impacts AM selection algorithm
Publish / Susbcribe
Randomize MAC address
Service discovery
Pairing
Data path
Terminate Data path
Terminate Pairing
Remote peers got cleaned up
Randomize MAC address

Figure 27. MAC address randomization flow.

The framework includes logic to create random unicast MAC addresses with the locally administered bit set:

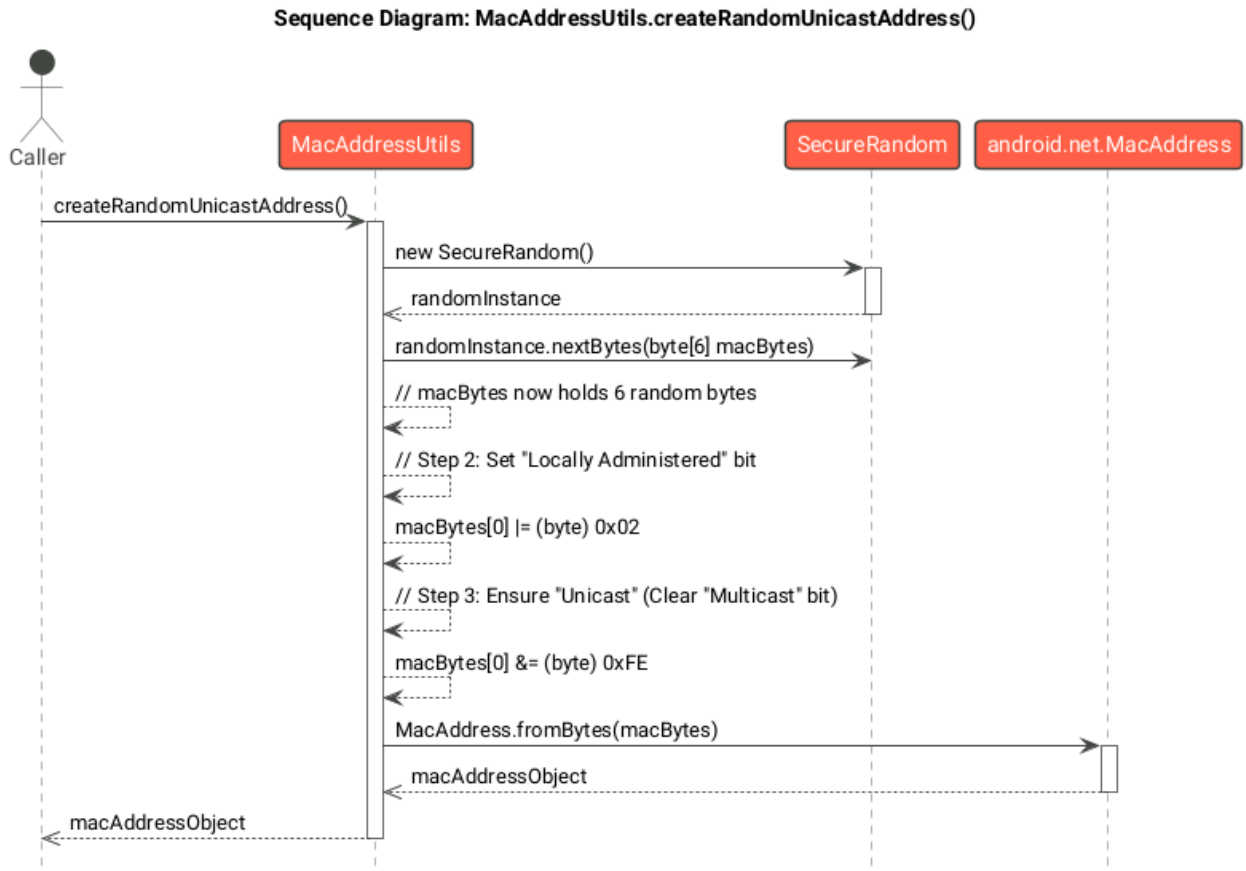

Figure 28. Create random unicast MAC address flow.

Instant communication mode

Instant communication mode in Wi-Fi Aware is a specialized operational mode designed to facilitate rapid service discovery and data communication between NAN devices. This mode can be activated when triggered by an out-of-band (OOB) mechanism, such as Near Field Communication (NFC) or Bluetooth Low Energy (BLE), or by user action.

Key characteristics:

- Continuous availability
- Defined availability schedules
- Accelerated beacon transmission
- **Publisher/subscribe behavior:** If the device acts as a publisher or subscriber for a service requiring instant communication, it includes the corresponding service ID in the service ID list attribute within its transmitted NAN beacons. If it receives a NAN beacon from a peer with a matching service ID in the subscribe service ID List, it sends a publish/subscribe message to that peer.
- **Interoperability:** A NAN device in Instant Communication mode continues to follow NAN synchronization and service discovery operations within DWs to ensure seamless interoperability with other NAN devices not operating in Instant Communication mode.

How a NAN device exits Instant Communication mode is implementation-specific.

See Attach (cluster building) and Publish and subscribe for enabling and disabling instant communication mode.

Ranging

NAN ranging is a key capability within Wi-Fi Aware that allows devices to determine the distance between two NAN devices in a NAN Cluster. It provides a mechanism for a service on a NAN device to initiate range measurements to or from another NAN device that supports this capability.

See Ranging peers and location-aware discovery for SDK API details. For further background, see the *NAN ranging overview* section in the Wi-Fi Aware specification version 4.0.

Android supports the following Aware ranging features:

- Ranging between discovered peers
- Periodic ranging once discovery is complete
- Geofencing-based discovery

Ranging, a time-critical operation, is fully offloaded to firmware. The framework and supplicant act as a passthrough, configuring necessary parameters.

Ranging between discovered peers

To request a range measurement, the framework creates `RangingRequest`, specifying a list of

Wi-Fi Aware peers. A single request can include multiple peers, and distances to all specified devices are measured and returned. Wi-Fi Aware peers can be added to a ranging request using either their MAC address or their `PeerHandle` through `addWifiAwarePeer(MacAddress peer)` and `addWifiAwarePeer(PeerHandle peer)`, respectively.

Note: In Android 16 or lower, most use cases require only a single Wi-Fi Aware peer. The accompanying diagram illustrates the call flow with one Aware peer; expansion to multiple peers is supported.

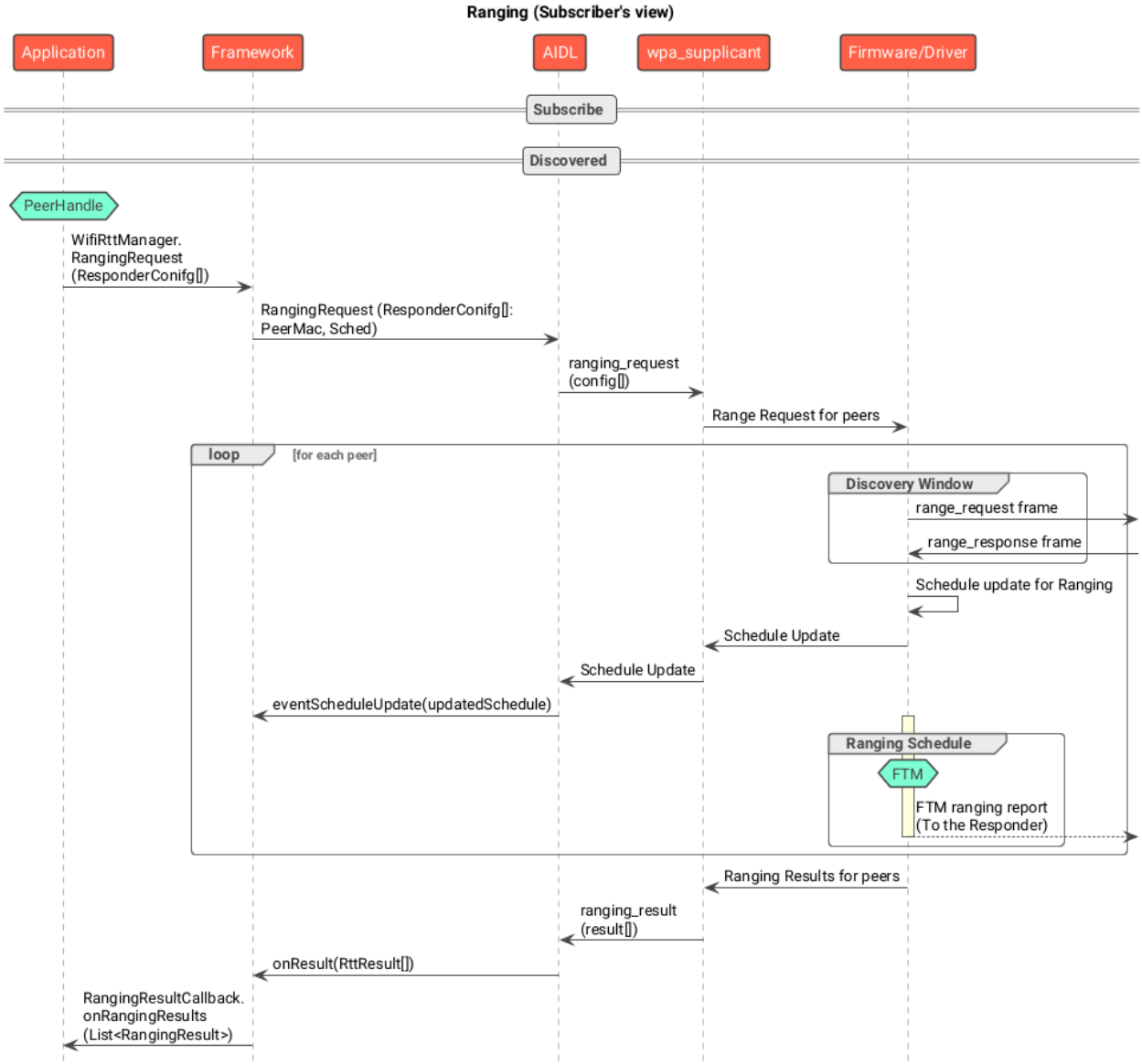

Figure 29. Ranging flow (subscriber).

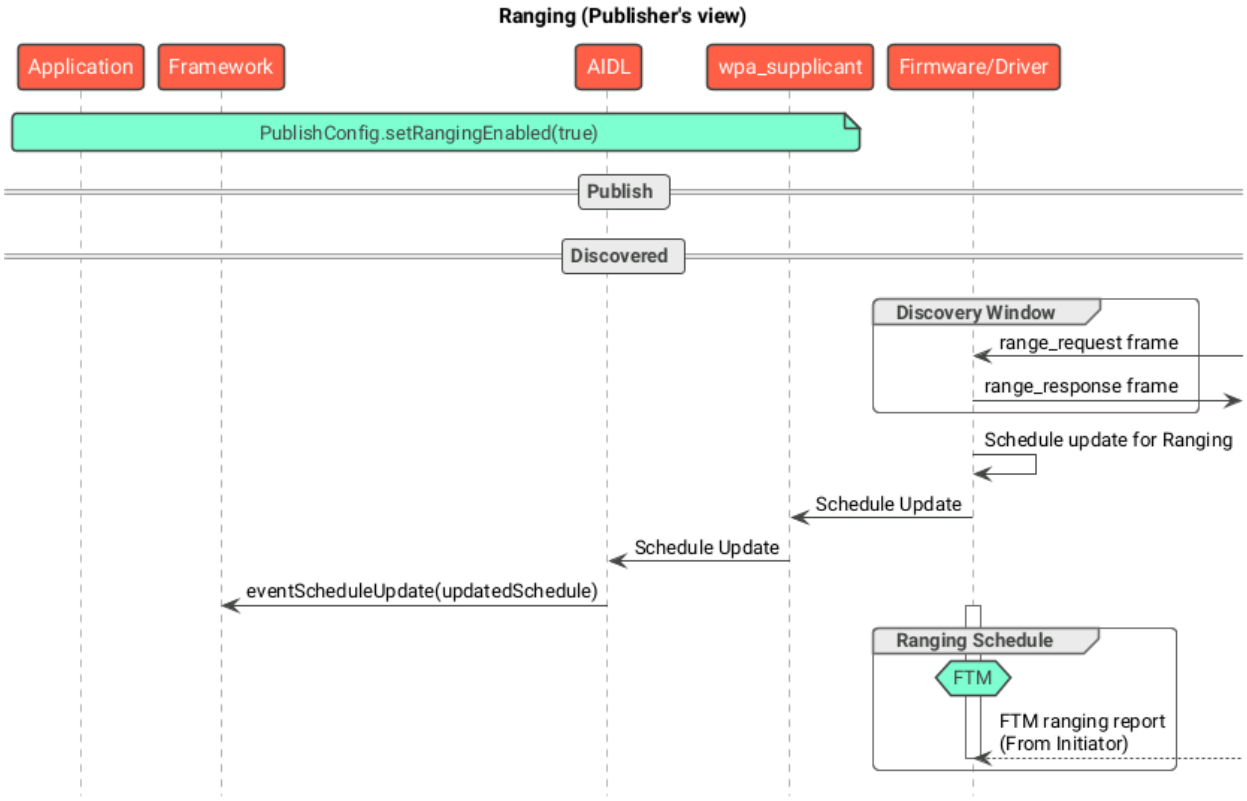

Figure 30. Ranging flow (publisher).

Geofencing-based discovery

Android provides the following parameters to enable geofencing:

- The publisher must enable ranging on the published service using `setRangingEnabled(true)`.
- The subscriber must specify a geofence using a combination of `setMinDistanceMm` and `setMaxDistanceMm`.

When a peer service is discovered within a geofence, the `onServiceDiscoveredWithinRange` callback is triggered, providing the measured distance to the peer. This callback is not periodic.

See Publish and subscribe and Service discovery (note: `onServiceDiscoveredWithinRange` is called instead of `onServiceDiscovered`).

The following illustration depicts the service discovery call flow, from subscriber and publisher viewpoints, utilizing geofencing. Firmware performs the geofence check along with other matching criteria.

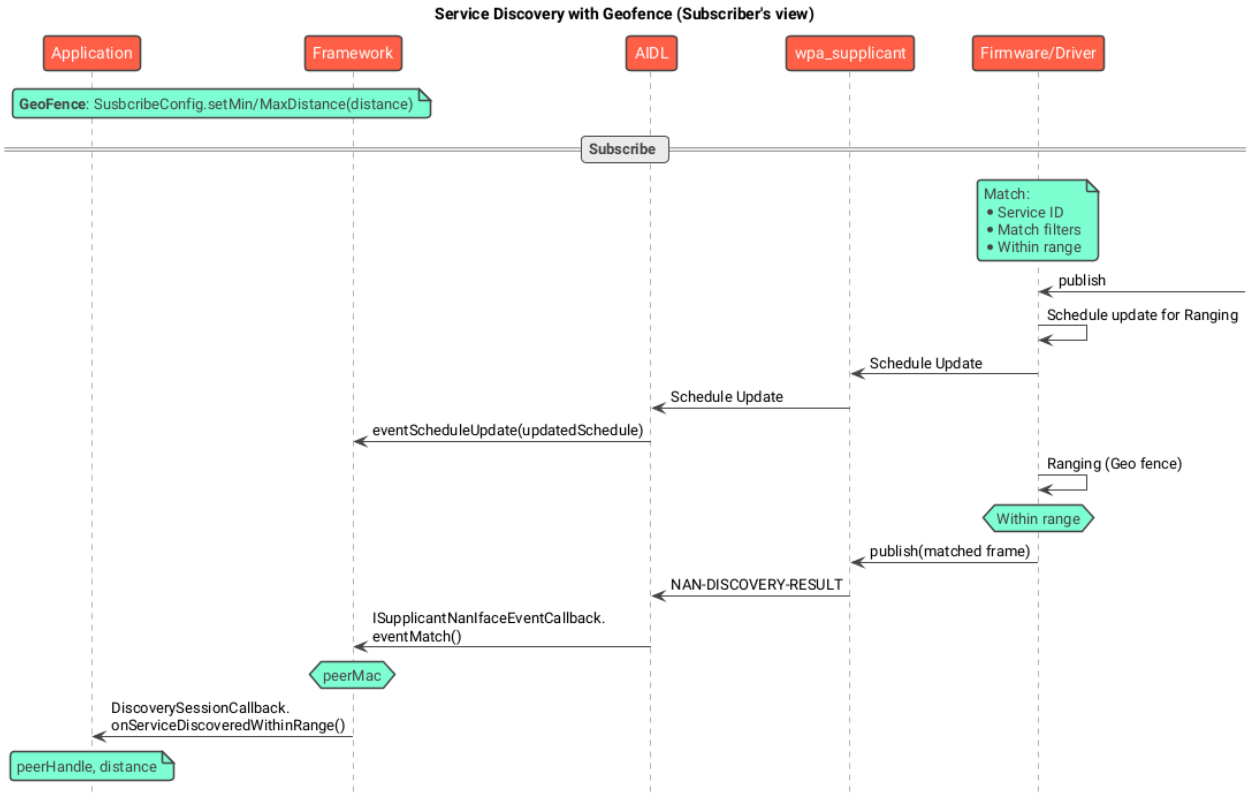

Figure 31. Service discovery with geofence flow (subscriber).

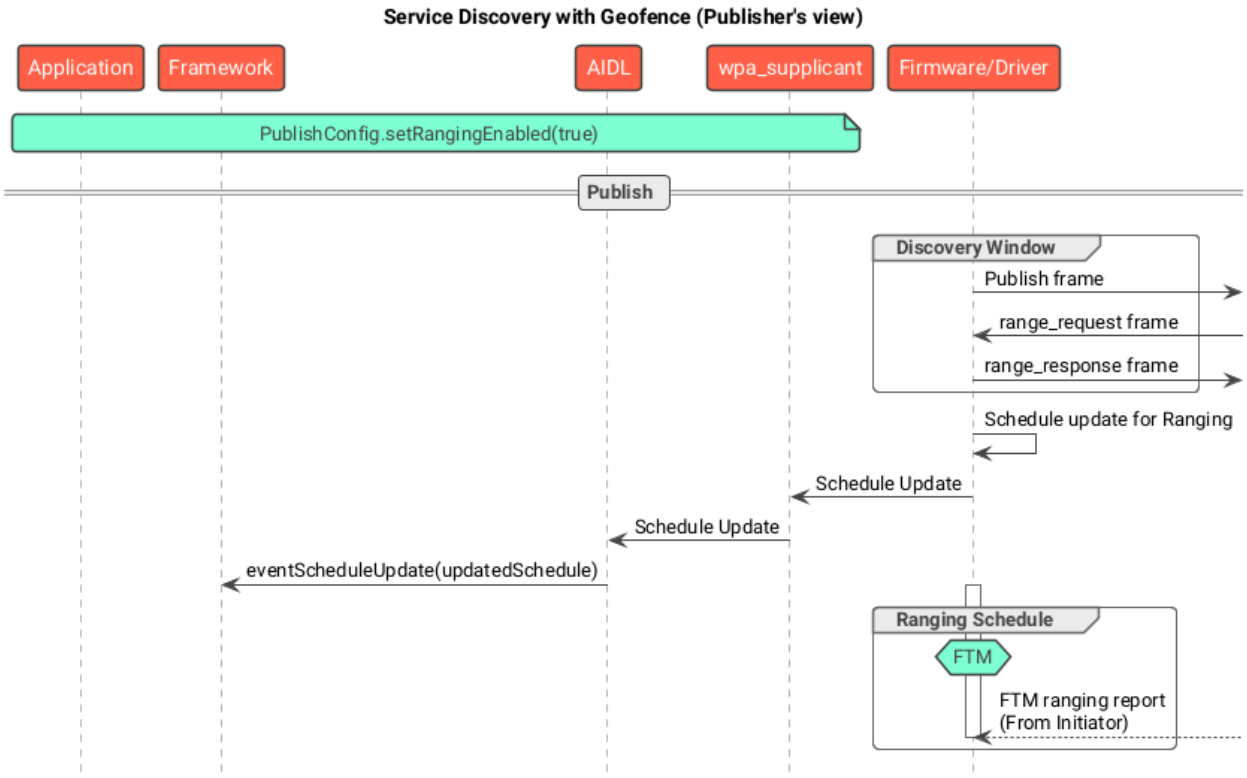

Figure 32. Service discovery with geofence flow (publisher).

Periodic ranging

If the subscriber device supports periodic ranging (indicated by `Characteristics.isPeriodicRangingSupported()`), it can enable this feature using `SubscribeConfig.setPeriodicRangingEnabled()` and define the interval with `SubscribeConfig.setPeriodicRangingInterval(int interval)`.

Publishers can optionally receive notifications for periodic ranging results by calling `PublishConfig.Builder.setPeriodicRangingResultsEnabled(boolean enable)`.

See Publish and subscribe and Service discovery. Once a service is discovered, the subscriber (and optionally publisher) is notified with `onRangingResultsReceived()`. This is periodic.

The following call flow outlines interactions between a subscriber and publisher for periodic ranging with discovery.

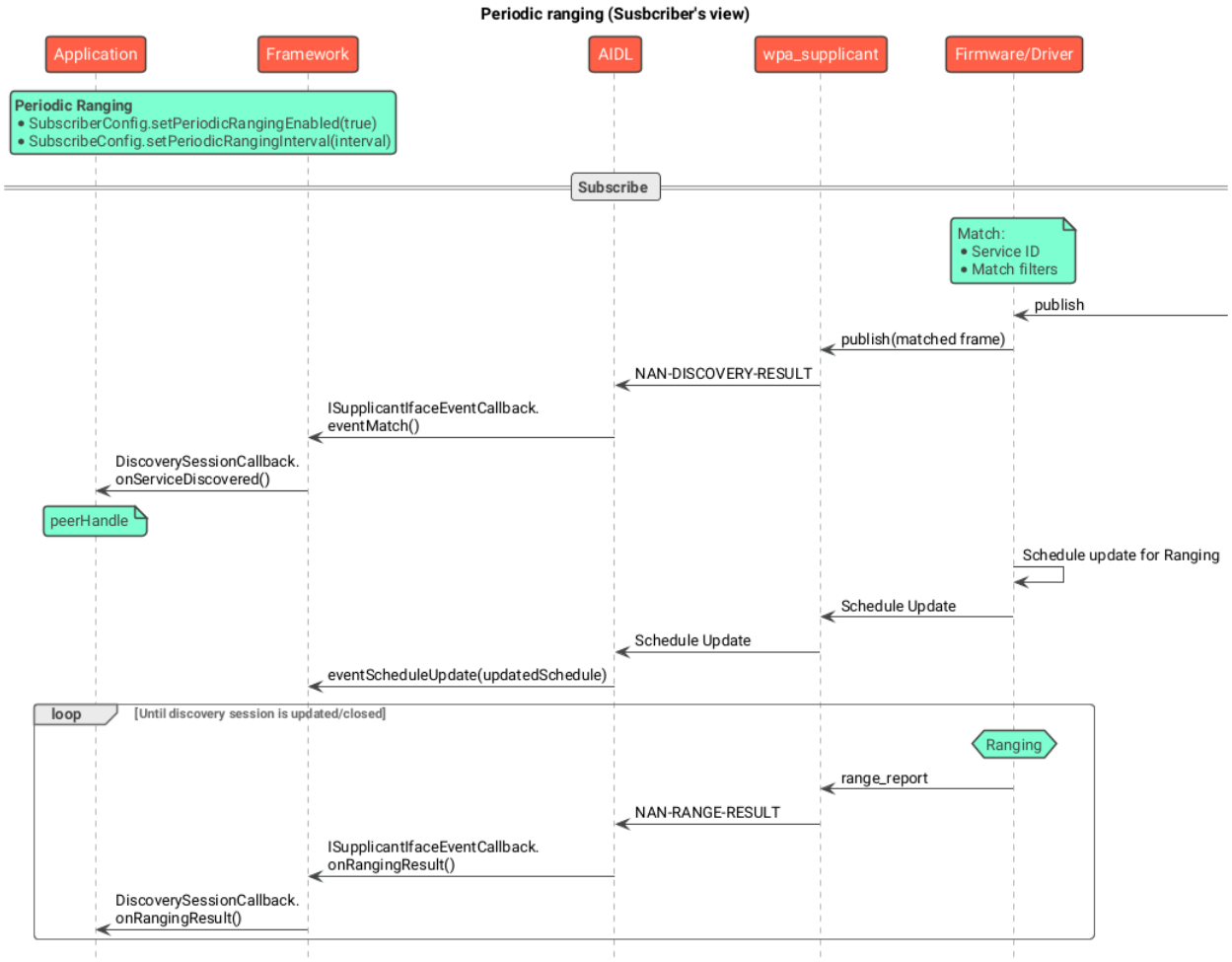

Figure 33. Periodic ranging flow (subscriber).

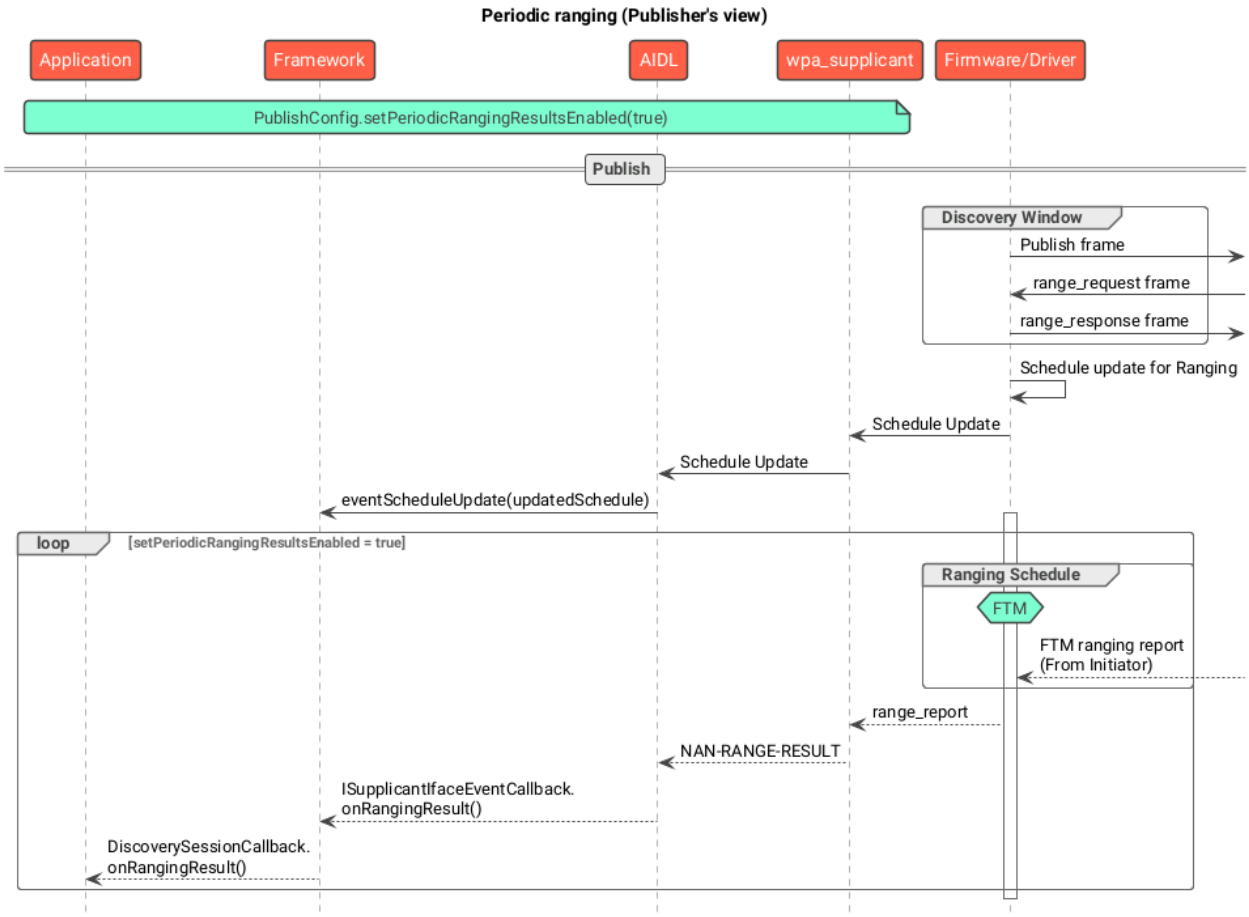

Figure 34. Periodic ranging flow (publisher).

NDL schedule

In Android 17 and higher, the Wi-Fi framework contributes to NDL schedule decisions.

Driving the schedule from the framework offers significant advantages:

- Improved throughput between devices with different chipsets or operating systems.
- Context-aware scheduling based on use case, desired QoS, application types, and user preferences.
- Leveraging framework knowledge of peer devices through out-of-band (OOB) channels, OOB agreements, and active STA and concurrency connections on peer devices.

However, in certain chipsets, driving the data path schedule entirely from the framework presents challenges:

- Chipset limitations (maximum channels, minimum block slot size, multi-band scheduling).
- Transient states like scanning that require temporary schedule adjustments.
- Peer device schedule proposals that require negotiation flexibility.
- Fine-tuning optimizations within firmware.

To address these considerations, Android 17 and higher incorporates the following objectives:

- The framework collects NDP scheduling capabilities as part of Aware capabilities to factor into proposed schedules.
- The chip can temporarily modify the schedule during transients (such as scanning) without consulting the framework (using Unscheduled Slot Availability (ULW), unsolicited schedule notifications, or schedule update requests).
- Based on chip capabilities, the framework operates in one of two modes:
    - **Full Framework mode:** For chipsets that accept framework schedules and don't implement chip-level scheduling offload, the schedule is fully driven by the framework.
    - **Chipset Offload mode:** For chipsets implementing chip-level schedule offload, the framework's schedule serves as a recommendation, except for mandatory requirements (indicated in the proposed schedule).

The schedule proposed by the framework takes into consideration:

1. Chip capabilities and scheduling limitations collected at initialization.
2. Regulatory limitations (using existing HAL APIs for available channels, including 6 GHz and DFS channel restrictions).
3. Concurrent activities on the device and active channel usage.
4. Peer device type and preferences (gathered through OOB channels).
5. Required QoS for Aware and concurrent interfaces (such as Infra STA).

General design guidelines for chipset-offloaded operation include:

- The framework-proposed schedule is a recommendation with a bitmask indicating mandatory parts (for example, minimum overlap with the concurrent Infra STA channel).
- The chip follows the proposed schedule unless a specific constraint prevents it. If the chip cannot fulfill mandatory constraints, it rejects NDP data path setup.

- After data path setup, the chip notifies the framework of the peer-proposed schedule and agreed schedule.
- If the chip alters the schedule later, it can do so autonomously as long as mandatory parts are preserved and the framework is notified.
- Following roaming events, the framework can issue an updated proposal using the same procedure.

Input parameters for deciding NDL schedule proposals:

Parameter	Description	Source
`MaxSchedChannels`	Maximum number of channels the chip can transition between in Aware and concurrent interfaces.	New chip capability
`MinSchedBlock16Tus`	Minimum block size for NAN scheduling (one block = 16 TU).	New chip capability
DBS vs. non-DBS	Dual-band simultaneous capability.	Existing HAL
Available channels	Regulatory channel availability.	Existing HAL
Concurrency	Active concurrency state.	Framework
Peer device type & preferences	Peer device type and preferences gathered out-of-band.	OOB
QoS for Aware	Input from the application depending on the use case (for example, average data rate and maximum latency for Aware), used to determine medium sharing between Aware and concurrent operations.	New API

Current NDL schedule	Active NDL and ranging schedule state maintained by the framework.	Framework
Peer proposed schedule	Schedule proposed by the peer device.	OOB
Recommended ranging schedule	Static ranging schedule recommendations.	Framework

Schedule update diagrams

Wi-Fi Aware schedule updates cover three main use cases during the connection lifecycle:

Local schedule changes

After a device starts publishing and subscribing, the local schedule may change due to use case or state changes (for example, allocating additional slots for fast setup or ranging, or STA connection changes). The framework uses the AIDL API to update the schedule, prompting the driver or firmware to adjust the local schedule and update subsequent beacons and SDFs.

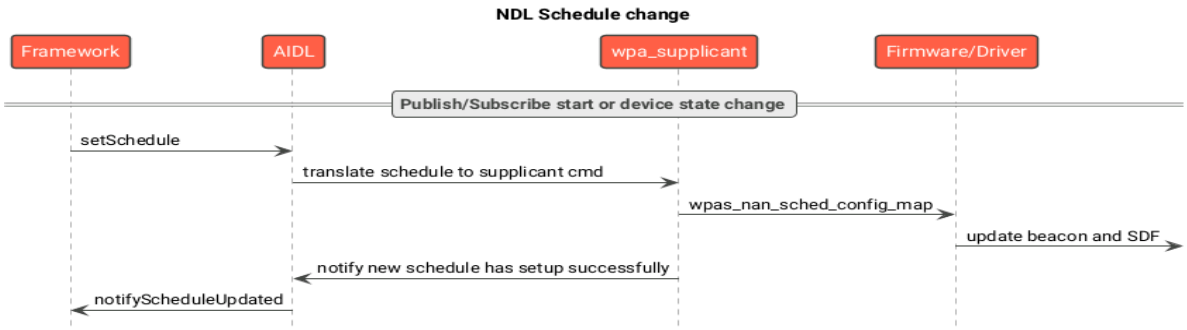

Figure 35. NDL schedule change flow.

Data path setup

To achieve optimal NDP performance, sufficient slots are allocated during data path setup to ensure high throughput and low latency.

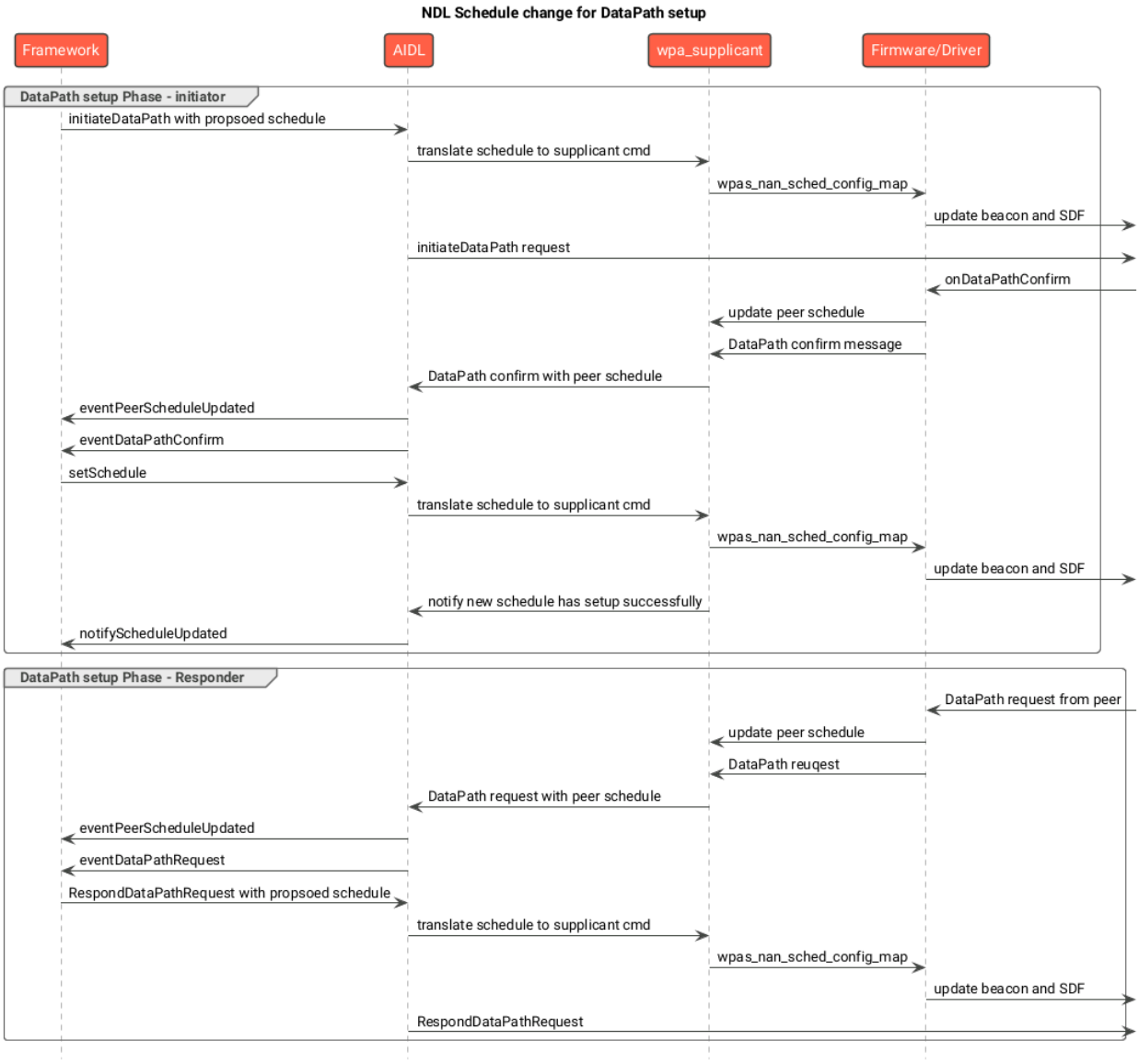

Figure 36. NDL schedule change for data path setup flow.

Peer schedule updates

When a peer device's status changes, its schedule might update. Local devices adjust accordingly to maintain connection quality.

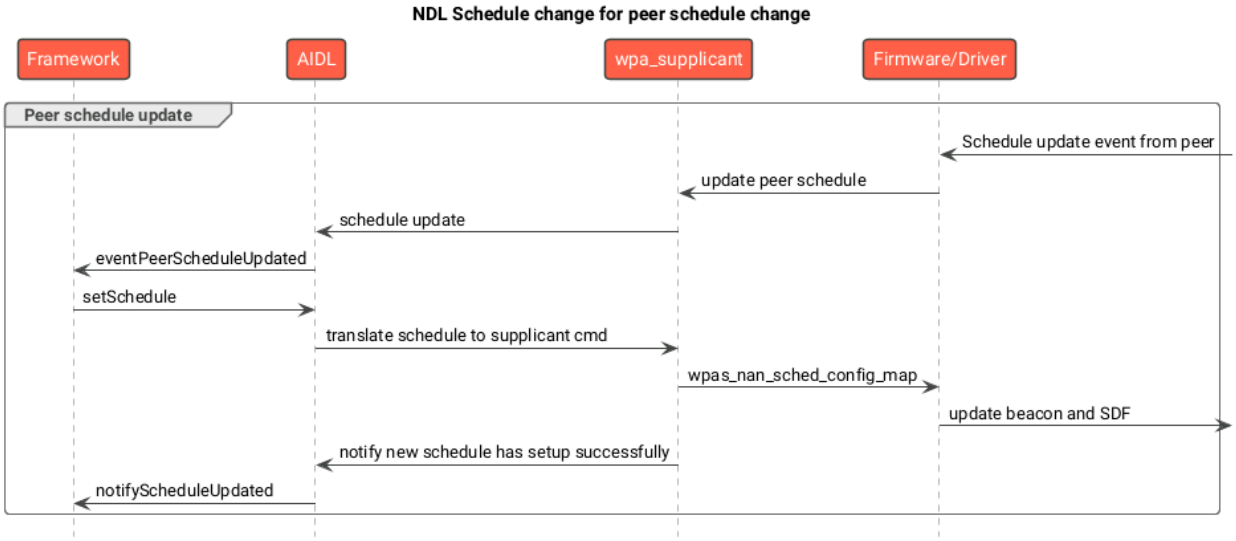

Figure 37. NDL schedule change for peer schedule change flow.

Schedule selection

The Wi-Fi framework provides `wpa_supplicant` with a traffic schedule containing committed and potential schedules adhering to these guidelines:

- At least 25% of available slots must be reserved for infrastructure connections.
- Priority is given to allocating slots for Aware connections.
- When connected to infrastructure, devices attempt to minimize channel changes. For example, slots are not upgraded to 6 GHz if doing so increases the total channel count.
- When no infrastructure is connected, high-throughput bands (5 GHz or 6 GHz) are prioritized.
- For single-radio devices, 5 GHz or 6 GHz is preferred even if 2.4 GHz is connected.

To simplify scheduling logic, availability is declared in blocks of 8 slots while remaining compatible with other peer formats.

Firmware controls the NDL schedule (scheduled one slot after the DW) and manages conditional slot usage based on system state.

Detailed algorithm

The framework uses the following sequence to generate optimal schedules:

1. Allocate local committed slots first.
2. Allocate peer committed slots that are not on social channels.
3. Fill remaining slots with local and peer infrastructure schedules from potential schedules:
    - If neither device has an infrastructure channel, no additional slots are allocated.
    - If both devices share the same infrastructure channel (or only one is connected), fill remaining slots with that channel.
    - If devices use different infrastructure channels, fill the first 8 slots with the social channel, slots 17 to 24 with the lower infrastructure channel number, and slots 25 to 32 with the remaining channel.
4. Fill any remaining unallocated slots with the social channel.

AIDL interfaces

The `NanSchedule` AIDL interface is passed to `wpa_supplicant` containing committed and potential schedules:

- For committed schedules, channel and bitmap definitions are provided.
- For potential schedules, preferred channels are provided (typically matching the infrastructure channel).

Both `NanInitiateDataPathRequest` and `NanRespondToDataPathIndicationRequest` include a `NanSchedule` field to reduce AIDL calls. For publish or subscribe start events or state changes, the framework calls `ISupplicantNanIface#setSchedule` to update local schedules.