Sample Code

Building peer-to-peer apps

Communicate with nearby devices over a secure, high-throughput, low-latency connection by using Wi-Fi Aware.

Download

iOS 26.0+ | iPadOS 26.0+ | Xcode 26.0+

Overview

This sample app uses the Wi-Fi Aware framework to build a peer-to-peer app. One device acts as a publisher by running a local simulation and advertising a service. Nearby devices connect to the publisher and subscribe to the simulation movements. The Wi-Fi Aware framework provides a secure, low-latency connection between the publisher and connected devices.

The sample app shows the interactions between the publisher and subscriber devices. The publisher simulates a satellite orbiting a planet. It sends the satellite's coordinates to connected subscriber devices. The subscribers receive the coordinates for each frame on the publisher simulation and update their local satellite's position, mirroring the publisher's.

Configure the sample code project

Because this sample app relies on using Wi-Fi Aware to make a network connection between devices, you can't run this sample in Simulator — you'll need to run it on physical devices.

Launch and run the app

1. Launch the app on two nearby devices.

2. Tap Host Simulation on one device to start it in publisher mode.

3. Tap View Simulation on the other device to start it in subscriber mode.

4. Pair the devices.

   - Tap **+** on both devices.

   - On the subscriber device, select the publisher device to pair with, and follow the on-screen steps to complete the pairing.

   - After the pairing is complete, each device shows the other device under the *Paired Devices* section. Dismiss the pairing views on both devices.

5. Connect the devices.

   - On the publisher device, tap Advertise.

   - On the subscriber device, tap Discover & Connect.

The devices then make a Wi-Fi Aware connection, and the satellite position on the subscriber device mirrors that of the one on the publisher. You can control the position of the satellite on the publisher device by tapping on it and moving it around, and you can observe the position of the satellite on the subscriber devices mirroring the one on the publisher device.

Authorize the app to publish and subscribe

The sample app uses the Wi-Fi Aware framework with the addition of the `com.apple.developer.wifi-aware` entitlement as a capability array. To perform both publish and subscribe operations, the sample app adds the `Publish` and `Subscribe` strings to the capability array:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/Prop
<plist version="1.0">
<dict>
    <key>com.apple.developer.wifi-aware</key>
    <array>
        <string>Publish</string>
        <string>Subscribe</string>
    </array>
</dict>
</plist>
```

Declare and access services

For an app to perform any Wi-Fi Aware operations, it needs to declare the services that it intends to use for publishing and subscribing. The sample app declares the *_sat-simulation._udp* service in its `Info.plist` with `Publishable` and `Subscribable` keys:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/Prop
<plist version="1.0">
<dict>
    <key>WiFiAwareServices</key>
    <dict>
        <key>_sat-simulation._udp</key>
        <dict>
            <key>Publishable</key>
            <dict/>
            <key>Subscribable</key>
            <dict/>
        </dict>
    </dict>
</dict>
</plist>
```

For convenience, the app creates extensions to the `WAPublishableService` and `WASubscribableService` classes for the service it uses to publish and subscribe:

```swift
let simulationServiceName = "_sat-simulation._udp"

extension WAPublishableService {
    public static var simulationService: WAPublishableService {
        allServices[simulationServiceName]!
    }
}

extension WASubscribableService {
    public static var simulationService: WASubscribableService {
        allServices[simulationServiceName]!
    }
}
```

Pair devices to connect

To set up a connection between devices, you need to pair the devices. Both the DeviceDiscoveryUI and AccessorySetupKit frameworks work for pairing. The sample app uses `DeviceDiscoveryUI` to pair devices.

To start a browser that can discover nearby devices, the app uses the `DevicePicker` view, providing it with `userSpecifiedDevices` as the list of allowed devices and `simulationService` as the service:

```
DevicePicker(.wifiAware(.connecting(to: .userSpecifiedDevices, from: .simulationServ
    logger.info("Paired Endpoint: \(endpoint)")
} label: {
    Image(systemName: "plus")
    Text("Add Device")
} fallback: {
    Image(systemName: "xmark.circle")
    Text("Unavailable")
}
```

To start a listener that allows nearby devices to discover and pair, the app uses `DevicePairingView` with `simulationService` as the service and `userSpecifiedDevices` as the list of allowed devices:

```
DevicePairingView(.wifiAware(.connecting(to: .simulationService, from: .userSpecifie
    Image(systemName: "plus")
    Text("Add Device")
} fallback: {
    Image(systemName: "xmark.circle")
    Text("Unavailable")
}
```

To perform pairing, tap the **+** button in the app on the two devices, one running in publisher mode and the other in subscriber mode.

Access paired devices

The sample app declares a `@State` variable to display the currently available paired devices (`WAPairedDevice` instances) using a List view:

```
@State var pairedDevices: [WAPairedDevice] = []
```

The app keeps track of currently paired devices that it has access to by iterating over the async sequence provided by `allDevices`.

```
do {
    for try await updatedDeviceList in WAPairedDevice.allDevices {
        pairedDevices = Array(updatedDeviceList.values)
    }
} catch {
    logger.error("Failed to get paired devices: \(error)")
}
```

Consider performance

Before using Wi-Fi Aware in an app, it's important to decide on the `WAPerformanceMode` to use for the connections. The first option is `WAPerformanceMode.bulk`, the recommended option for almost all use cases. The second option is `WAPerformanceMode.realtime`, which is for instances that require low latency. The sample app uses `realtime` as it needs to send position updates for every frame of the simulation scene and thus requires very low latency.

Changing the performance mode from `realtime` to `bulk` in the app demonstrates the difference between the two modes. To change the performance mode in the sample app, change the following line in `NetworkConfig.swift`:

```
let appPerformanceMode: WAPerformanceMode = .realtime
// Change the performance mode.
let appPerformanceMode: WAPerformanceMode = .bulk
```

Important

Using `realtime` performance mode can drain a device's battery, so use it only when it's required.

In addition to setting the performance mode to `realtime`, the app also sets the traffic service class to `interactiveVideo` to achieve better prioritization of traffic resuling in lower latency. Experiment with changing the traffic service class in `NetworkConfig.swift`.

```
let appAccessCategory: WAAccessCategory = .interactiveVideo
// Change the traffic service class.
let appAccessCategory: WAAccessCategory = .bestEffort
```

Connect using the Network framework

The sample app uses the Network framework to publish and subscribe. The Wi-Fi Aware framework implements the `ListenerProvider` and `BrowserProvider` protocols in the Network framework. The app creates a network listener and browser using `NetworkListener` and `NetworkBrowser`, respectively. Running these instances results in Wi-Fi Aware publish and subscribe operations. After the app discovers the network endpoints, it creates a Wi-Fi Aware connection using `NetworkConnection`.

Publish a service

To start publishing, the app creates a `NetworkListener` instance by providing it `simulation Service` as the service to advertise, and specifying the paired devices allowed to connect. In the sample app, `allPairedDevices` allows a connection from any paired device. Using `selected(_:)` limits connections to a smaller subset of devices, and using `matching(_:)` provides a predicate filter on `WAPairedDevice`.

```
try await NetworkListener(for:
    .wifiAware(.connecting(to: .simulationService, from: .allPairedDevices)),
using: .parameters {
    Coder(receiving: NetworkEvent.self, sending: NetworkEvent.self, using: NetworkJS
        UDP()
    }
}
.wifiAware { $0.performanceMode = appPerformanceMode }
.serviceClass(appServiceClass))
.onStateUpdate { listener, state in
    logger.info("\(String(describing: listener)) : \(String(describing: state))")

    // Process the listener state update.
}
.run { connection in
    logger.info("Incoming Connection: \(String(describing: connection))")

    // Handle the incoming connection.
}
```

Note

The sample app uses the User Datagram Protocol (UDP) as the transport protocol because the use case doesn't require reliable or in-order delivery. For most cases, using a protocol like the Transmission Control Protocol (TCP) or Quick UDP Internet Connections (QUIC) would likely be a better option.

Subscribe to a service

To start subscribing, the app creates a `NetworkBrowser` instance by providing it `simulationService` as the service, and specifying the paired devices that it can discover.

```swift
let browser = NetworkBrowser(for:
    .wifiAware(.connecting(to: .allPairedDevices, from: .simulationService))
)
.onStateUpdate { browser, state in
    logger.info("\(String(describing: browser)): \(String(describing: state))")

    // Process the browser state update.
}
```

To start the subscriber, the sample app uses the `run` method on the `NetworkBrowser` instance. The app then selects the first discovered endpoint advertising the service and stops the browse operation by returning `.finish(firstEndpoint)`. Typically, an app evaluates the list of discovered endpoints and decides to continue the browse operation until the desired endpoint is found.

```swift
let endpoint = try await browser.run { waEndpoints in
    logger.info("Discovered: \(waEndpoints)")
    if let firstEndpoint = waEndpoints.first {
        return .finish(firstEndpoint)
    } else {
        return .continue
    }
}
```

Make a connection

The sample app uses the Network framework to make a connection. For convenience, the app declares a type alias `WiFiAwareConnection` for a parameterized `NetworkConnection`, and a `Codable` and `Sendable` enumeration to encode and send satellite position coordinates over the connection.

```
typealias WiFiAwareConnection = NetworkConnection<Coder<NetworkEvent, NetworkEvent,

public enum NetworkEvent: Codable, Sendable {
    case startStreaming
    case satelliteMovedTo(position: CGPoint, dimensions: CGSize)
}
```

On the publisher side, the sample app receives incoming connections in the `run` closure attached to the `NetworkListener`. As it receives new connections, the app sets up the state update handler and holds a reference to the connection instance.

```
connection.onStateUpdate { connection, state in
    logger.info("\(String(describing: connection)) : \(String(describing: state))")

    // Process the connection state update.
}
```

On the subscriber side, the `WAEndpoint` instances the app receives from the `NetworkBrowser` are connectable when the app passes them to the `NetworkConnection`. To set up a connection to a discovered endpoint using `NetworkConnection`, the app provides the endpoint of interest and sets the Wi-Fi Aware performance mode and traffic service class. It also sets up a connection state update handler similar to the publisher.

The sample app connects to the first endpoint that's discovered, provides the *_sat-simulation._udp* service, and stops the browser after making the first connection. Although this particular design of connecting to the first available endpoint works for the sample app's use case, most apps typically review the discovered endpoint and make a connection only if required.

```
let connection = NetworkConnection(
    to:
        endpoint,
    using: .parameters {
        Coder(receiving: NetworkEvent.self, sending: NetworkEvent.self, using: Netwo
            UDP()
        }
    }
    .wifiAware { $0.performanceMode = appPerformanceMode }
    .serviceClass(appServiceClass)
)

connection.onStateUpdate { connection, state in
    logger.info("\(connection.debugDescription): \(String(describing: state))")

    // Process the connection state update.
}
```

> **Note**
>
> The `WAPerformanceMode` setting provided during connection should match the one the app provides when starting the listener. Mismatched performance modes lead to an undefined behavior.

Send data to connected devices

The publisher instance of the sample app sends the coordinates of the *satellite* in the local simulation to all connected devices, for every frame:

```
func send(_ event: NetworkEvent, to connection: WiFiAwareConnection) async {
    do {
        try await connection.send(event)
    } catch {
        logger.error("Failed to send to: \(connection.debugDescription): \(error)")
    }
}

func sendToAll(_ event: NetworkEvent) async {
    for connection in connections {
        await send(event, to: connection)
    }
}
```

Receive data from the connected device

To receive data over the connection, the app uses the `Network` framework APIs available in the `NetworkConnection` instance:

```
for try await (event, _) in connection.messages {
    // Process the incoming message and update the satellite position using the rece
}
```

The sample app instance running on the subscriber side receives the coordinates of the satellite as it moves in the publisher simulation. It then updates the position of the local satellite based on the coordinates it receives.

Monitor connection performance

The sample app uses the Wi-Fi Aware framework to monitor the performance of connections to peer devices. It gets the `WAPairedDevice` representing the remote endpoint and the current performance report of the connection by accessing the `currentPath` property of the connection. The app accesses the `WAPerformanceReport` through the `.wifiAware` extension to `currentPath`, which contains Wi-Fi Aware-specific connection performance information:

```
if let wifiAwarePath = try await connection.currentPath?.wifiAware {
    let connectedDevice = wifiAwarePath.endpoint.device
    let performanceReport = wifiAwarePath.performance

    // Use the performance report.
}
```

The app displays two performance attributes for each connected device: signal strength and transmit latency.

Get the signal strength

Signal strength, represented as a number between `0.0` and `1.0`, is present in the `WAPerformanceReport` instance. A number closer to `1.0` indicates a stronger signal strength.

```
performanceReport.signalStrength
```

Get the transmit latency

The Wi-Fi Aware framework reports the measured transmit latency of packets sent to a paired device, per access category (`WAAccessCategory`). Each `NetworkConnection` is bound to a specific `WAAccessCategory` based on the `serviceClass` configuration that was provided during the creation of the connection instance. In case of the sample app, the publisher specifies `interactiveVideo` as the service class to transmit packets to the subscribers. To access the average transmit latency for the corresponding access category, use the following code:

```
performanceReport.transmitLatency[appAccessCategory]?.average
```

Get Wi-Fi Aware errors

The Wi-Fi Aware framework extends `NWError` with a `wifiAware` property that provides Wi-Fi Aware with specific errors that occur on the `NetworkListener`, `NetworkBrowser` or `NetworkConnection` instances. The app gets the underlying Wi-Fi Aware error from the NWError the Network framework provides as part of the `NWListener.State.failed(_:)`, `NWBrowser.State.failed(_:)`, and `NWConnection.State.failed(_:)` states depending on whether the app is publishing, browsing, or connecting.

```
case .failed(let error): // Get the Wi-Fi Aware from the NWError as error.wifiAware
```

See Also

Essentials

Connecting devices for peer-to-peer Wi-Fi

Make outgoing and accept incoming secure connections with paired devices.

Adopting Wi-Fi Aware

Add entitlements and declare your app's services.

`com.apple.developer.wifi-aware`

The entitlement the system requires for an app to use the Wi-Fi Aware framework.

`WiFiAwareServices`

Dictionaries of Wi-Fi Aware services that the app can publish or subscribe to.